Traderbot Documentation

Release 1

Jordan Dworaczyk

Oct 06, 2017

About Traderbot

All documentation for Traderbot is built using Sphinx and hosted on Read the Docs. The docs are kept in the `docs/` directory at the top of the source tree.

CHAPTER 1

What is Traderbot?

Traderbot is software designed to automatically trade cryptocurrency.

Automated trading, also known as algorithmic trading, can be defined as a system of computers programmed to follow specific instructions on how to place trade orders.[1] This system is used to generate profits at a speed and frequency that is impossible for human traders. Some advantages of algorithmic trading include the mitigation of emotional decision making, improved order entry speeds, consistency, and backtesting.[2]

Traderbot's software is open source and all documents relating to this project can be found here.

Contributing to Traderbot

The following is a set of instructions for contributing to Traderbot and its packages. These are mostly guidelines, not rules. Use your best judgment, and feel free to propose changes to this document in a pull request.

Note: This project and everyone participating in it is governed by the Traderbot Code of Conduct. By participating, you are expected to uphold this code. Please report unacceptable behavior to jordan.dwo@gmail.com.

How can I contribute?

You can contribute by reporting bugs, suggesting enhancements, and submitting your own code using Pull Requests.

[1] **Algorithmic trading** (automated trading, black-box trading, or simply algo-trading) is the process of using computers programmed to follow a defined set of instructions for placing a trade in order to generate profits at a speed and frequency that is impossible for a human trader. The defined sets of rules are based on timing, price, quantity or any mathematical model. Apart from profit opportunities for the trader, algo-trading makes markets more liquid and makes trading more systematic by ruling out emotional human impacts on trading activities. Basics of Algorithmic Trading: Concepts and Examples | Investopedia

[2] One of the big reasons that algorithmic trading has become so popular is because of the advantages that it holds over trading manually. The **advantages of algo trading** are related to speed, accuracy, and reduced costs. Advantages of Algorithmic Trading | Nasdaq.com Forex Education

Reporting Bugs

It is encouraged that you submit an issue if you find a bug. Do your best to provide as much information as you can so that people can recreate the bug that you experienced. Don't forget to mention the version of the bot that you are using, as well as, the system that you are using to run the bot. If there is an error message displayed, then include the error messagge in your submission.

> **Warning:** **Remove any sensitive information** from the screenshots in your bug report. Sensitive info may include things like your API Keys, passwords, or acount balances.

Suggesting Enhancements

Please submit an issue if you would like to suggest an enhancement. When submitting your suggestion, try to follow the *user story* convention often found in Agile software development.

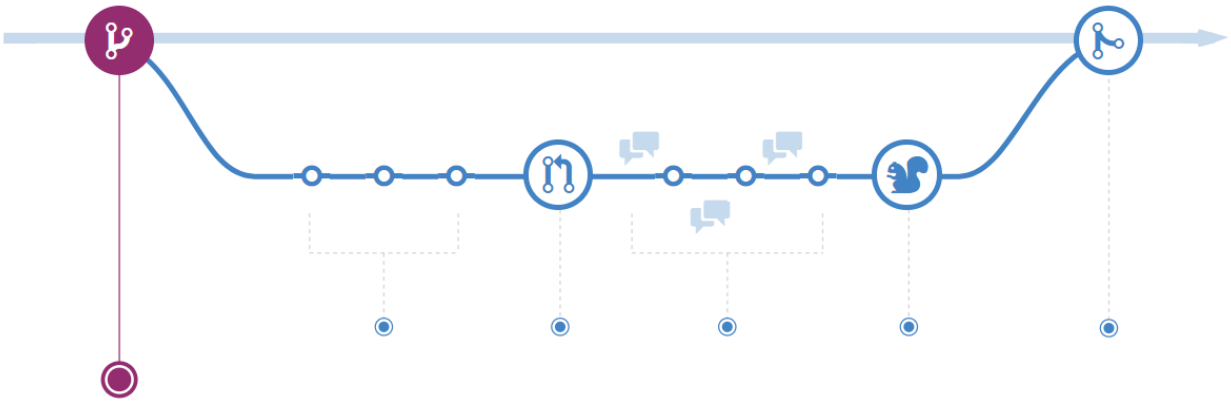
User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system. They typically follow a simple template:

```
As a <type of user>, I want <some goal> so that <some reason>.
```

User stories are often written on index cards or sticky notes, stored in a shoe box, and arranged on walls or tables to facilitate planning and discussion. As such, they strongly shift the focus from writing about features to discussing them. In fact, these discussions are more important than whatever text is written.[1]

Workflow

The Traderbot project implements a standard GitHub workflow, known as *GitHub Flow*, we hope to attract more contributors to the community due to its simple and easy to understand nature.[2]



See also:

GitHub Flow. Understanding the GitHub Flow | GitHubGuides

[1] User stories are part of an agile approach that helps shift the focus from writing about requirements to talking about them. All agile user stories include a written sentence or two and, more importantly, a series of conversations about the desired functionality. Read more about User Stories | MountainGoateSoftware

[2] GitHub Flow is a lightweight, branch-based workflow that supports teams and projects where deployments are made regularly. This guide explains how and why GitHub Flow works. Read more about GitHub Flow | GitHubGuides

Pull Requests

When submitting a Pull Request make sure you:

- Fill in the required template.
- Do not include issue numbers in the PR title
- Include screenshots and animated GIFs in your pull request whenever possible.
- Follow the *Styleguides*

Styleguides

Please mantain the following python and docsting styleguides in order to facilitate easy communication among contributers and to properly document source code.

Python

All python code must adhere to the PEP 8 – Style Guide for Python Code.

You may use the following `pep8_cheatsheet.py` as a guide:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
"""This module's docstring summary line.

This is a multi-line docstring. Paragraphs are separated with blank lines.
Lines conform to 79-column limit.

Module and packages names should be short, lower_case_with_underscores.
Notice that this in not PEP8-cheatsheet.py

Seriously, use flake8. Atom.io with https://atom.io/packages/linter-flake8
is awesome!

See http://www.python.org/dev/peps/pep-0008/ for more PEP-8 details
"""


import os  # STD lib imports first
import sys  # alphabetical

import some_third_party_lib  # 3rd party stuff next
import some_third_party_other_lib  # alphabetical

import local_stuff  # local stuff last
import more_local_stuff
import dont_import_two, modules_in_one_line  # IMPORTANT!
from pyflakes_cannot_handle import *  # and there are other reasons it should be␣
↪avoided # noqa
# Using # noqa in the line above avoids flake8 warnings about line length!


_a_global_var = 2  # so it won't get imported by 'from foo import *'
_b_global_var = 3
```

```
A_CONSTANT = 'ugh.'


# 2 empty lines between top-level funcs + classes
def naming_convention():
    """Write docstrings for ALL public classes, funcs and methods.
    Functions use snake_case.
    """
    if x == 4:  # x is blue <== USEFUL 1-liner comment (2 spaces before #)
        x, y = y, x  # inverse x and y <== USELESS COMMENT (1 space after #)
    c = (a + b) * (a - b)  # operator spacing should improve readability.
    dict['key'] = dict[0] = {'x': 2, 'cat': 'not a dog'}


class NamingConvention(object):
    """First line of a docstring is short and next to the quotes.
    Class and exception names are CapWords.
    Closing quotes are on their own line
    """

    a = 2
    b = 4
    _internal_variable = 3
    class_ = 'foo'  # trailing underscore to avoid conflict with builtin

    # this will trigger name mangling to further discourage use from outside
    # this is also very useful if you intend your class to be subclassed, and
    # the children might also use the same var name for something else; e.g.
    # for simple variables like 'a' above. Name mangling will ensure that
    # *your* a and the children's a will not collide.
    __internal_var = 4

    # NEVER use double leading and trailing underscores for your own names
    __nooooooodontdoit__ = 0

    # don't call anything (because some fonts are hard to distiguish):
    l = 1
    O = 2
    I = 3

    # some examples of how to wrap code to conform to 79-columns limit:
    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if width == 0 and height == 0 and \
           color == 'red' and emphasis == 'strong' or \
           highlight > 100:
            raise ValueError('sorry, you lose')
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)

    # empty lines within method to enhance readability; no set rule
    short_foo_dict = {'looooooooooooooooooong_element_name': 'cat',
                      'other_element': 'dog'}
```

```
    long_foo_dict_with_many_elements = {
        'foo': 'cat',
        'bar': 'dog'
    }

    # 1 empty line between in-class def'ns
    def foo_method(self, x, y=None):
        """Method and function names are lower_case_with_underscores.
        Always use self as first arg.
        """
        pass

    @classmethod
    def bar(cls):
        """Use cls!"""
        pass

# a 79-char ruler:
# 3456789112345678921234567893123456789412345678951234567896123456789712345678 9

"""
Common naming convention names:
snake_case
MACRO_CASE
camelCase
CapWords
"""

# Newline at end of file
```

Note: `pep8_cheatsheet.py` is a GitHub Gist. See full example | by Richard Bronosky

Docstrings

Documentation is automatically generated from Python docstrings using Read the Docs, Sphinx, and Napoleon. Therefore, to properly document code please adhere to either the Google Style or NumPy Style of writing Python docstrings.

Google Style:

```
def func(arg1, arg2):
  """Summary line.

  Extended description of function.

  Args:
      arg1 (int): Description of arg1
      arg2 (str): Description of arg2

  Returns:
      bool: Description of return value

  """
  return True
```

Numpy style:

```
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of arg1
    arg2 : str
        Description of arg2

    Returns
    -------
    bool
        Description of return value

    """
    return True
```

See also:

- Complete Example of Google Style Docstrings
- Complete Example of NumPy Style Docstrings

Traderbot Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment

- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at jordan.dwo@gmail.com. The project team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant homepage, version 1.4

Traderbot package

Submodules

Traderbot.test module

Example Google style docstrings.

This module demonstrates documentation as specified by the Google Python Style Guide. Docstrings may extend over multiple lines. Sections are created with a section header and a colon followed by a block of indented text.

Example

Examples can be given using either the `Example` or `Examples` sections. Sections support any reStructuredText formatting, including literal blocks:

```
$ python example_google.py
```

Section breaks are created by resuming unindented text. Section breaks are also implicitly created anytime a new section starts.

Traderbot.test.**module_level_variable1**
int – Module level variables may be documented in either the Attributes section of the module docstring, or in an inline docstring immediately following the variable.

Either form is acceptable, but the two should not be mixed. Choose one convention to document module level variables and be consistent with it.

Todo

- For module TODOs
- You have to also use sphinx.ext.todo extension

---

class Traderbot.test.**ExampleClass**(*param1*, *param2*, *param3*)
Bases: object

The summary line for a class docstring should fit on one line.

If the class has public attributes, they may be documented here in an Attributes section and follow the same formatting as a function's Args section. Alternatively, attributes may be documented inline with the attribute's declaration (see __init__ method below).

Properties created with the @property decorator should be documented in the property's getter method.

attr1
str – Description of *attr1*.

attr2
int, optional – Description of *attr2*.

attr3 = None
Doc comment *inline* with attribute

attr4 = None
list of str – Doc comment *before* attribute, with type specified

attr5 = None
str – Docstring *after* attribute, with type specified.

example_method(*param1*, *param2*)
Class methods are similar to regular functions.

Note: Do not include the *self* parameter in the Args section.

Parameters

- **param1** – The first parameter.
- **param2** – The second parameter.

Returns True if successful, False otherwise.

readonly_property
str – Properties should be documented in their getter method.

readwrite_property
list of str – Properties with both a getter and setter should only be documented in their getter method.

If the setter method contains notable behavior, it should be mentioned here.

exception Traderbot.test.**ExampleError**(*msg*, *code*)
Bases: Exception

Exceptions are documented in the same way as classes.

The __init__ method may be documented in either the class level docstring, or as a docstring on the __init__ method itself.

Either form is acceptable, but the two should not be mixed. Choose one convention to document the __init__ method and be consistent with it.

Note: Do not include the *self* parameter in the Args section.

Parameters

- **msg** (*str*) – Human readable string describing the exception.
- **code** (int, optional) – Error code.

msg
str – Human readable string describing the exception.

code
int – Exception error code.

Traderbot.test.**example_generator**(*n*)
Generators have a Yields section instead of a Returns section.

Parameters **n** (*int*) – The upper limit of the range to generate, from 0 to *n* - 1.

Yields *int* – The next number in the range of 0 to *n* - 1.

Examples

Examples should be written in doctest format, and should illustrate how to use the function.

```
>>> print([i for i in example_generator(4)])
[0, 1, 2, 3]
```

Traderbot.test.**function_with_pep484_type_annotations**(*param1: int*, *param2: str*) → bool
Example function with PEP 484 type annotations.

Parameters

- **param1** – The first parameter.
- **param2** – The second parameter.

Returns The return value. True for success, False otherwise.

Traderbot.test.**function_with_types_in_docstring**(*param1*, *param2*)
Example function with types documented in the docstring.

PEP 484 type annotations are supported. If attribute, parameter, and return types are annotated according to PEP 484, they do not need to be included in the docstring:

Parameters

- **param1** (*int*) – The first parameter.
- **param2** (*str*) – The second parameter.

Returns The return value. True for success, False otherwise.

Return type bool

Traderbot.test.**module_level_function**(*param1*, *param2=None*, **args*, ***kwargs*)

This is an example of a module level function.

Function parameters should be documented in the `Args` section. The name of each parameter is required. The type and description of each parameter is optional, but should be included if not obvious.

If *args or **kwargs are accepted, they should be listed as `*args` and `**kwargs`.

The format for a parameter is:

```
name (type): description
    The description may span multiple lines. Following
    lines should be indented. The "(type)" is optional.

    Multiple paragraphs are supported in parameter
    descriptions.
```

Parameters

- **param1** (*int*) – The first parameter.
- **param2** (`str`, optional) – The second parameter. Defaults to None. Second line of description should be indented.
- ***args** – Variable length argument list.
- ****kwargs** – Arbitrary keyword arguments.

Returns

True if successful, False otherwise.

The return type is optional and may be specified at the beginning of the `Returns` section followed by a colon.

The `Returns` section may span multiple lines and paragraphs. Following lines should be indented to match the first line.

The `Returns` section supports any reStructuredText formatting, including literal blocks:

```
{
    'param1': param1,
    'param2': param2
}
```

Return type bool

Raises

- `AttributeError` – The `Raises` section is a list of all exceptions that are relevant to the interface.
- `ValueError` – If *param2* is equal to *param1*.

Traderbot.test.**module_level_variable2 = 98765**
int – Module level variable documented inline.

The docstring may span multiple lines. The type may optionally be specified on the first line, separated by a colon.

Module contents

Python Module Index

t

Index