Stackoverflow Watcher Documentation

Release 0.2.1

Matt Deacalion

Sep 27, 2017

Contents

Stackoverflow Watcher is a library and command line tool written in Python that notifies you of relevant questions when they're posted on Stack Overflow. It's very simple to customise. For example, you could customise the questions so that they are only considered relevant if have any of the following criteria:

- Questions tagged with *python*
- Questions tagged with *javascript* and not tagged with *jquery* or *underscore*
- Questions asked less than an *hour ago* with no answers
- Questions where the user's *score* is above 100
- Questions that have been *viewed* over 100 times
- Questions with the word "*monkey*" in the title

Note: Creating relevant filters such as these is accomplished by creating a custom class that inherits from `stack_watcher.Question`. You can see how to do this in the subclassing section.

CHAPTER 1

Getting started

Installation

Installing Stackoverflow Watcher can be done with pip, run this command in your terminal:

```
$ pip install stack-watcher
```

To verify it has been installed you can run the following command:

```
$ stack-watcher -V
0.3.4
```

Get the source code

If you would like to play with the example subclasses, contribute or just see the source code you can clone the repository from GitHub with the following command:

```
$ git clone git://github.com/Matt-Deacalion/Stackoverflow-Watcher.git
```

Usage

Once you've installed Stackoverflow Watcher you have two ways of using it:

1. Run it using the *"stack-watcher"* command.
2. Use the underlying library directly from your code.

Once you're comfortable with this you can begin customising how it behaves by subclassing the main components. This provides a lot of flexibility and isn't difficult. You can learn how to do this in the customising section and see examples of it in action in the examples directory.

Command Line

Stackoverflow Watcher comes with the *"stack-watcher"* command. You can use this from the shell to start watching and filtering questions. The arguments are:

--tag	What tag should we restrict the feed to?
--interval	How many seconds should we wait between feed requests?
--retriever	What Python class should we use for the `Retriever`?
--question	What Python class should we use for `Question` objects?

If you wanted to watch all of the latest questions, with no filtering at all. You would use the command on it's own like so:

```
$ stack-watcher
```

Let's say you wanted to watch only questions that have the *html* tag. You could use the **'tag'** argument like this:

```
$ stack-watcher --tag html
```

Library

Stackoverflow Watcher also has an API which you can use directly in your own code. Here's an example that does pretty much the same thing as the *"stack-watcher"* command with no arguments:

```
from stack_watcher import Retriever

retriever = Retriever()

for question in retriever.questions():
    print(question)
```

This will continue retrieving and displaying new questions indefinitely.

If you wanted to restrict the questions to a specific tag you could pass the **'tag'** argument to `Retriever` like this:

```
from stack_watcher import Retriever

retriever = Retriever(tag='html')

for question in retriever.questions():
    print(question)
```

Customising

Out of the box Stackoverflow Watcher does nothing special. It doesn't even perform any type of filtering. The components that make up the underlying library are intentionally very bare-bones. This is because any custom functionality is added by subclassing one of these components. This gives you a lot of flexibility while at the same time, keeps everything very simple.

Note: The **'tag'** argument to the *"stack-watcher"* command and `Retriever` class does **not** do any filtering, it simply grabs the feed from Stack Overflow for that specific tag. Real filtering is achieved through subclassing the

`Question` class.

Components and Subclassing

The two main components can be found in the `stack_watcher` package.

Question

A single `Question` (or `Question` subclass) instance represents an individual question on Stack Overflow. Question objects have the ability to **'verify'** that they are relevant by comparing themselves to a set of *rules*. If all of the rules return True, then the question object is considered relevant.

A *rule* is any Python property in a `Question` subclass that begins with *"is_"*. This is inspired by the same technique used in unit testing with Python.

To run all of the rules in a question object, you can check the `adheres_to_rules` property. This will only return True if there are no rules or all of the rules return True.

If we had this `Question` subclass:

```
from stack_watcher import Question

class MammothQuestion(Question):

    @property
    def is_asked_by_a_woolly_mammoth(self):
        return False
```

It would never be considered a relevant question:

```
>>> question = MammothQuestion()
>>> question.adheres_to_rules
False
```

Retriever

This class is responsible for retrieving the questions from Stack Overflow. One of the reasons to subclass this would be to add functionality that helps make the retrieval more reliable, for instance: handling throttling or authentication.

Subclassing examples

Here are a few examples that implement useful features, see the unit tests and the examples directory for more advanced scenarios.

Question subclass

In this example the `MyQuestion` class will only consider a question relevant if it's unanswered and has 25 or more views.

```
from stack_watcher import Question

class MyQuestion(Question):

    @property
    def is_not_answered(self):
        return not self.answered

    @property
    def is_eye_catching(self):
        return self.view_count >= 25
```

If I save this code into a file called `questions.py` I'll be able to run the following command from the terminal to begin fetching questions that match this criteria:

```
$ stack-watcher --question questions.MyQuestion
```

Retriever subclass

A useful reason to subclass `Retriever` is to override the `throttled()` method and do something useful. For instance, instead of just waiting for the time out period to end - we could change the IP address and try again.

```
class ThrottleRetriever(Retriever):
    def throttled(self):
        self.change_ip_address()
        time.sleep(5)

    def change_ip_address(self):
        call(['sudo', 'systemctl', 'restart', 'openvpn@random'])
```

The `change_ip_address()` method could implement anything you need to do on your system to change the IP address. You could even add proxy server support here.

Contributing

Any help is very welcome, thank you! Here's a few ways you could help out:

- Write code along with unit tests and submit a pull request
- Improve the documentation
- Submit ideas for new features
- Report bugs