flagman Documentation

Release 0.1.0-dev

Scott Colby

Aug 15, 2018

Contents

1 Features **3**

2 Use Cases **5**

3 The Anatomy of an Action **7**
3.1 The Anatomy of an Action 7

4 Overlapping Signals **11**
4.1 Overlapping Signals 11

5 A Real-World Use **13**
5.1 A Real-World Use 13

6 CLI Reference **15**
6.1 CLI Reference 15

7 API Reference **17**
7.1 API Reference 17

8 Installation **23**

9 Changelog **25**

10 Contributing **27**

11 License **29**

12 Indices and tables **31**

Python Module Index **33**

Warning: This is the documentation for a development version of flagman.

Documentation for the Most Recent Stable Version

Perform arbitrary actions on signals.

```
$ flagman --usr1 print 'a fun message' --usr1 print 'another message' --usr2 print_
↪once 'will be printed once' &
INFO:flagman.cli:PID: 49220
INFO:flagman.cli:Setting loglevel to WARNING
init  # the set_up phase of the three actions
init
init
$ kill -usr1 49220  # actions are called in the order they're passed in the arguments
a fun message
another message
$ kill -usr2 49220  # actions can remove themselves when no longer useful
will be printed once
cleanup  # the tear_down phase of the `print_once` action
WARNING:flagman.core:Received `ActionClosed`; removing action `PrintOnceAction`
# *snip* traceback
flagman.exceptions.ActionClosed: Only print once
$ kill -usr1 49220  # other actions are still here, though
a fun message
another message
$ kill 49220  # responds gracefully to shutdown requests
cleanup  # the tear_down phase of the two remaining actions
cleanup
```

On this page:

- *Features*
- *Use Cases*
- *The Anatomy of an Action*
- *Overlapping Signals*
- *A Real-World Use*
- *CLI Reference*
- *API Reference*
- *Installation*
- *Changelog*
- *Contributing*
- *License*
- *Indices and tables*

CHAPTER 1

Features

- Safe execution of code upon receiving `SIGHUP`, `SIGUSR1`, or `SIGUSR2`
- Optional systemd integration–sends `READY=1` message when startup is complete
- Complete mypy type annotations

CHAPTER 2

Use Cases

The use cases are endless! But specifically, `flagman` is useful to adapt services that do not handle signals in a convenient way for your infrastructure.

I wrote `flagman` to solve a specific problem, examined in *A Real-World Use*.

CHAPTER 3

The Anatomy of an Action

Learn how to create your own Actions!

Warning: This is the documentation for a development version of flagman.

Documentation for the Most Recent Stable Version

3.1 The Anatomy of an Action

Actions are the primary workhorse of *flagman*. Writing your own actions allows for infinite possible uses of the tool!

3.1.1 The Action Class

Actions are instances of the abstract base class *flagman.Action*. Let's look at the included `PrintAction` as an illustrative example.

```
class PrintAction(Action):
    """A simple Action that prints messages at the various stages of execution.

    (message: str)
    """

    def set_up(self, msg: str) -> None:  # type: ignore
        """Store the message to be printed and print the "init" message.

        :param msg: the message
        """
        self_msg = msg
        print('init')
```

(continues on next page)

(continued from previous page)

```
    def run(self) -> None:
        """Print the message."""
        print(self._msg)

    def tear_down(self) -> None:
        """Print "cleanup" message."""
        print('cleanup')
```

We start with a standard class definition and docstring:

```
class PrintAction(Action):
    """A simple Action that prints messages at the various stages of execution.

    (message: str)
    """
```

We inherit from `Action`. The docstring is parsed and becomes the documentation for the action in the CLI output:

```
$ flagman --list
name  - description [(argument: type, ...)]
------------------------------------------------------------------------------
print - A simple Action that prints messages at the various stages of execution.
        (message: str)
```

If the `Action` takes arguments, it is wise to document them here. The name of the action is defined in an entry point–see *Registering an Action* below.

Next is the `set_up()` method.

```
def set_up(self, msg: str) -> None:  # type: ignore
    """Store the message to be printed and print the "init" message.

    :param msg: the message
    """
    self_msg = msg
    print('init')
```

All arguments will be passed to this method as strings. If other types are expected, do the conversion in `set_up()` and raise errors as necessary. If mypy is being used, the `# type: ignore` comment is required since the parent implementation takes `*args`.

Do any required set up in this method: parsing arguments, reading external data, etc. If you want values from the environment (e.g. if API tokens or other values that should not be passed on the command line are needed), you can get them here. *flagman* itself does not provide facilities for parsing the environment, configuration files, etc.

Next we have the most important method, `run()`. This is the only abstract method on `Action` and as such it must be implemented.

```
def run(self) -> None:
    """Print the message."""
    print(self._msg)
```

Perform whatever action you wish here. This method is called once for each time **flagman** is signaled with the proper signal, assuming low enough rates of incoming signals. See below in the *Overlapping Signals* section for more information.

Because of *flagman*'s architecture, it is safe to do *anything* inside the `run()` method. It is not actually called from the signal handler, but in the main execution loop of the program. Therefore, normally "risky" things to do in signal handlers involving locks, etc. (including using the `logging` module, for example) are completely safe.

Finally, there is the `tear_down()` method.

```
def tear_down(self) -> None:
    """Print "cleanup" message."""
    print('cleanup')
```

Here you can perform any needed cleanup for your action like closing connections, writing out statistics, etc.

This method will be called when the action is "closed" (see below), during garbage collection of the action, and before *flagman* shuts down.

3.1.2 "Closing" an Action

If an Action has fulfilled its purpose or otherwise no longer needs to be called, it can be "closed" by calling its `_close()` method. This method takes no arguments and always returns `None`.

Calling this method does two things: it calls the action's `tear_down()` method and it sets a flag that prevents further calls to the internal `_run()` method that *flagman* uses to actually run Actions.

Further calls to `_run()` will raise a *flagman.ActionClosed* exception and will cause the removal of the action from the internal list of actions to be run. If there are no longer any non-closed actions, **flagman** will exit with code `1`, unless it was originally called with the `--successful-empty` option, in which case it will exit with `0`.

If you want to close your own action in its `run()` method, a construction like so is advised:

```
def run(self) -> None:
    if some_condition:
        self._close()
        raise ActionClosed('Closing because of some_condition')
    else:
        ...
```

This will print your argument to `ActionClosed` to the log and will result in the immediate removal of the action from the list of actions to be run. If `ActionClosed` is not raised, **flagman** will not realize the action has been closed and will not remove it from the list of actions to be run until the next time `run()` would be called, i.e. the next time the signal is delivered for the action.

3.1.3 Registering an Action

flagman detects available actions in the `flagman.action` entry point group. Actions must be distributed in packages with this entry point defined. For instance, here is how the built-in actions are referenced in *flagman*'s `setup.cfg`:

```
[options.entry_points]
flagman.action =
    print = flagman.actions:PrintAction
    delay_print = flagman.actions:DelayedPrintAction
    print_once = flagman.actions:PrintOnceAction
```

The name to the left of the = is how the action will be referenced in the CLI. The entry point specifier to the right of the = points to the class implementing the action. See the Setuptools documentation for more information about using entry points.

CHAPTER 4

Overlapping Signals

flagman attempts to handle overlapping signals in an intelligent manner. This algorithm is explained here:

Warning: This is the documentation for a development version of flagman.

Documentation for the Most Recent Stable Version

4.1 Overlapping Signals

flagman attempts to handle overlapping signals in an intelligent manner. A signal is "overlapping" if it arrives while actions for previously-arrived signals are still running.

flagman handles overlapping signals of the same identity by coalescing and of different identities by handling them serially but in a non-guaranteed order.

For example, take the following sequence of events.

1. **flagman** is sleeping awaiting a signal to arrive
2. `SIGUSR1` arrives
3. a long-running action for `SIGUSR1` starts
4. `SIGUSR2` arrives
5. the long-running action for `SIGUSR1` finishes
6. a long-running action for `SIGUSR2` starts
7. `SIGUSR1` arrives
8. `SIGUSR2` arrives; it is ignored since the `SIGUSR2` actions are currently running
9. `SIGHUP` arrives
10. the long-running action for `SIGUSR2` finishes

11. a short-running action for `SIGUSR2` starts and finishes
12. a short-running action for `SIGHUP` starts and finishes; note that `SIGHUP` arrived after the most recent `SIGUSR1`– only intra-signal action ordering is guaranteed
13. a long-running action for `SIGUSR1` starts
14. the long-running action for `SIGUSR1` finishes
15. **`flagman`** returns to sleep until the next handled signal arrives

CHAPTER 5

A Real-World Use

An examination of the problem I built *flagman* to solve.

Warning: This is the documentation for a development version of flagman.

Documentation for the Most Recent Stable Version

5.1 A Real-World Use

I have a multi-layered DNS setup that involves ALIAS records that are only resolved on a hidden master and are passed as A or AAAA records to the authoritative slaves.

I wanted to check if the resolved value of the ALIAS records have changed and send out DNS NOTIFYs to the slaves when they do, but I didn't want to store state in a file on disk.

Enter *flagman*. I wrote an action that queries the hidden master and saves the values of the records I'm interested in as member variables. If the values have changed since the last run, the hidden master's REST API is called for force the sending of a NOTIFY out to its slaves.

This is integrated with three systemd units:

```
# flagman.service
[Unit]
Description=Run flagman

[Service]
Type=notify
NotifyAccess=main
ExecStart=/path/to/flagman --usr1 dnscheck
```

```
# flagman-notify.service
[Unit]
```

(continues on next page)

(continued from previous page)

```
Description=Send SIGUSR1 to flagman

[Service]
Type=oneshot
ExecStart=/bin/systemctl kill -s SIGUSR1 flagman.service
```

```
# flagman-notify.timer
[Unit]
Description=Run flagman-notify hourly

[Timer]
OnCalendar=hourly
RandomizedDelaySec=300
Persistent=true

[Install]
WantedBy=timers.target
```

Simple? Not quite. But quite extensible and useful in a variety of situations.

CHAPTER 6

CLI Reference

The CLI options for **`flagman`** are documented here.

> **Warning:** This is the documentation for a development version of flagman.
>
> Documentation for the Most Recent Stable Version

6.1 CLI Reference

-h, --help	show this help message and exit
--list, -l	list known actions and exit
--hup ACTION	add an action for SIGHUP
--usr1 ACTION	add an action for SIGUSR1
--usr2 ACTION	add an action for SIGUSR2
--successful-empty	if all actions are removed, exit with 0 instead of the default 1
--no-systemd	do not notify systemd about status
--quiet, -q	only output critial messages; overrides *–verbose*
--verbose, -v	increase the loglevel; pass multiple times for more verbosity

6.1.1 Notes

- Options to add actions take the argument *ACTION*, the action name as shown in `flagman --list`, followed by an action-defined number of arguments, which are also documented in `flagman --list`. See the output of `flagman --help` for a more complete view of this.
- All options to add actions for signals may be passed multiple times.

- When a signal with multiple actions is handled, the actions are guaranteed to be taken in the order they were passed on the command line.
- Calling with no actions set is a critical error and will cause an immediate exit with code 2.

CHAPTER 7

API Reference

Information about the interfaces *flagman* exposes are here.

Warning: This is the documentation for a development version of flagman.

Documentation for the Most Recent Stable Version

7.1 API Reference

This part of the documentation covers all of the interfaces exposed by *flagman*.

7.1.1 The Action Class

This class is the abstract class you should inherit from to write your own actions.

class flagman.**Action**(**args*)

The base Action class.

_close()

Close the action, preventing future runs and executing tear down logic.

Return type None

set_up(**args*)

Perform any required set up for the Action.

Return type None

run()

Run the Action.

Return type None

tear_down()
Perform any required clean up for the Action.

Return type None

7.1.2 The Core Module

These functions and members are imported from the flagman.core module to be used if using *flagman* as a library instead of a standalone tool.

flagman.**HANDLED_SIGNALS List[signal.Signals]**
Signals in this list are handled by flagman. The CLI module auto-generates the appropriate CLI option for each signal.

flagman.**KNOWN_ACTIONS Mapping[ActionName, Type[Action]]**
Mapping of action entry point names to Action classes. Populated from the pkg_resources *flagman.action* entry point group.

flagman.**create_action_bundles**(*args_dict*)
Parse the enabled actions and insert them into the global ACTION_BUNDLES mapping.

The input dictionary should be like:

```
{'usr1': [['action1', 'arg1a', 'arg2a'], ['action2', 'arg2a']],
 'usr2': [['action3'], ['action4', 'arg4a', 'arg4b']]}
```

Parameters **args_dict** (Mapping[str, Iterable[Sequence[str]]]) – a mapping of strings to an Iterable of Action names

Return type int

Returns The number of configured actions

flagman.**run**()
Run the flagman "event loop".

Waits for a signal to be raised and dispatches to the user-defined handlers as appropriate.

Return type None

flagman.**set_handlers**()
Register handlers for the signals we're interested in.

Uses the global HANDLED_SIGNALS to decide what signals to register for.

Danger starts here!

Return type None

7.1.3 Errors and Exceptions

exception flagman.**ActionClosed**
The Action is closed and no longer will do anything on a call to *run()*.

7.1.4 Built-in Actions

Print Actions

Print actions for flagman.

Most likely only useful for debugging.

class flagman.actions.**PrintAction**(**args*)
Bases: flagman.actions.action.Action

A simple Action that prints messages at the various stages of execution.

(message: str)

set_up(*msg*)
Store the message to be printed and print "init" message.

Parameters **msg** (str) – the message

Return type None

run()
Print the message.

Return type None

tear_down()
Print "cleanup" message.

Return type None

class flagman.actions.**DelayedPrintAction**(**args*)
Bases: flagman.actions.print.PrintAction

An Action that prints messages at the various stages of execution and has a configurable delay in the run stage.

(message: str, delay: int)

set_up(*msg*, *delay*)
Store the message and the delay.

Parameters

- **msg** (str) – the message
- **delay** (str) – the delay in seconds

Return type None

run()
Print the message, delay, and print a finished message.

Return type None

class flagman.actions.**PrintOnceAction**(**args*)
Bases: flagman.actions.print.PrintAction

An Action that prints a message once and then cleans up after itself.

(message: str)

set_up(*msg*)
Store the message.

Parameters **msg** (str) – the message

Return type None

run()
Print the message and close the action.

Return type None

7.1.5 Types

Type aliases used throughout flagman.

flagman.types.**ActionName**
Type alias for the name of an Action.

alias of builtins.str

flagman.types.**ActionArgument**
Type alias for the the argument to an Action.

alias of builtins.str

flagman.types.**SignalNumber**
Type alias for a signal number.

alias of builtins.int

7.1.6 The CLI Module

Module that contains the command line for flagman.

Why does this file exist, and why not put this in __main__? You might be tempted to import things from __main__ later, but that will cause problems–the code will get executed twice:

- When you run *python -m flagman* python will execute *__main__.py* as a script. That means there won't be any *flagman.__main__* in *sys.modules*.
- When you import __main__ it will get executed again (as a module) because there's no *flagman.__main__* in *sys.modules*.

Also see (1) from http://click.pocoo.org/5/setuptools/#setuptools-integration

class flagman.cli.**AllAttrEmptyString**
Return '' for any attribute.

flagman.cli.**_sigterm_handler**(*signum*, *_frame*)
Raise SystemExit on SIGTERM.

Return type None

flagman.cli.**parse_args**(*argv*)
Parse the arguments for the flagman CLI.

Parameters **argv** (Sequence[str]) – a Squence of argument strings

Return type Namespace

Returns the parsed arguments as an argparse Namespace

flagman.cli.**list_actions**()
Pretty-print the list of available actions to stdout.

Return type None

flagman.cli.**main**()
The main function of the flagman CLI.

Don't call this from library code, use your own version implenting analogous logic.

Return type Optional[int]

Returns An exit code or None

flagman.cli.**main_wrapper**()
Main wrapper that handles graceful exiting on KeyboardInterrupt.

Return type Optional[int]

Returns An exit code or None

7.1.7 Systemd Notify Utilities

Systemd notification protocol implementation.

class flagman.sd_notify.**SystemdNotifier**(*debug=False*)
This class holds a connection to the systemd notification socket.

It can be used to send messages to systemd using its notify method.

__init__(*debug=False*)
Instantiate a new notifier object.

This will initiate a connection to the systemd notification socket.

Normally this method silently ignores exceptions (for example, if the systemd notification socket is not available) to allow applications to function on non-systemd based systems. However, setting debug=True will cause this method to raise any exceptions generated to the caller, to aid in debugging.

Return type None

notify(*state*)
Send a notification to systemd.

State is a string; see the man page of sd_notify (http://www.freedesktop.org/software/systemd/man/sd_notify.html) for a description of the allowable values.

Normally this method silently ignores exceptions (for example, if the systemd notification socket is not available) to allow applications to function on non-systemd based systems. However, setting debug=True will cause this method to raise any exceptions generated to the caller, to aid in debugging.

Return type None

CHAPTER 8

Installation

flagman has no required dependencies outside the Python Standard Library.

At the moment, installation must be performed via GitHub:

```
$ pip install git+https://github.com/scolby33/flagman.git
```

For prettier output for `flagman --list`, install the `color` extra:

```
$ pip install git+https://github.com/scolby33/flagman.git[color]
```

flagman targets Python 3 and tests with Python 3.7. Versions earlier than 3.7 are not guaranteed to work.

CHAPTER 9

Changelog

flagman adheres to the Semantic Versioning ("Semver") 2.0.0 versioning standard. Details about this versioning scheme can be found on the Semver website. Versions postfixed with '-dev' are currently under development and those without a postfix are stable releases.

You are reading the documents for version 0.1.0-dev of *flagman*.

Changes as of 18 July 2018

- Initial implementation of the flagman functionality.

CHAPTER 10

Contributing

There are many ways to contribute to an open-source project, but the two most common are reporting bugs and contributing code.

If you have a bug or issue to report, please visit the issues page on GitHub and open an issue there.

If you want to make a code contribution, feel free to open a pull request!

CHAPTER 11

License

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

flagman, 17
flagman.actions.print, 19
flagman.cli, 20
flagman.sd_notify, 21
flagman.types, 20

Index

Symbols

__init__() (flagman.sd_notify.SystemdNotifier method), 21
_close() (flagman.Action method), 17
_sigterm_handler() (in module flagman.cli), 20

A

Action (class in flagman), 17
ActionArgument (in module flagman.types), 20
ActionClosed, 18
ActionName (in module flagman.types), 20
AllAttrEmptyString (class in flagman.cli), 20

C

create_action_bundles() (in module flagman), 18

D

DelayedPrintAction (class in flagman.actions), 19

F

flagman (module), 17
flagman.actions.print (module), 19
flagman.cli (module), 20
flagman.sd_notify (module), 21
flagman.types (module), 20

H

HANDLED_SIGNALS (in module flagman), 18

K

KNOWN_ACTIONS (in module flagman), 18

L

list_actions() (in module flagman.cli), 20

M

main() (in module flagman.cli), 20
main_wrapper() (in module flagman.cli), 21

N

notify() (flagman.sd_notify.SystemdNotifier method), 21

P

parse_args() (in module flagman.cli), 20
PrintAction (class in flagman.actions), 19
PrintOnceAction (class in flagman.actions), 19

R

run() (flagman.Action method), 17
run() (flagman.actions.DelayedPrintAction method), 19
run() (flagman.actions.PrintAction method), 19
run() (flagman.actions.PrintOnceAction method), 20
run() (in module flagman), 18

S

set_handlers() (in module flagman), 18
set_up() (flagman.Action method), 17
set_up() (flagman.actions.DelayedPrintAction method), 19
set_up() (flagman.actions.PrintAction method), 19
set_up() (flagman.actions.PrintOnceAction method), 19
SignalNumber (in module flagman.types), 20
SystemdNotifier (class in flagman.sd_notify), 21

T

tear_down() (flagman.Action method), 17
tear_down() (flagman.actions.PrintAction method), 19