sphinxcontrib-autoprogram Documentation

Release 0.1.6

Hong Minhee

Mar 29, 2020

Contents

This contrib extension, *sphinxcontrib.autoprogram*, provides an automated way to document CLI programs. It scans `argparse.ArgumentParser` object, and then expands it into a set of `.. program::` and `.. option::` directives.

In order to use it, add *sphinxcontrib.autoprogram* into `extensions` list of your Sphinx configuration file (`conf.py`):

```
extensions = ['sphinxcontrib.autoprogram']
```

See also:

Module `argparse` This extension assumes a program to document is made using `argparse` module which is a part of the Python standard library.

CHAPTER 1

.. autoprogram:: directive

Its only and simple way to use is `.. autoprogram::` directive. It's similar to `sphinx.ext.autodoc` extension's `.. automodule::` and other directives.

For example, given the following Python CLI program (say it's `cli.py`):

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='An integer for the accumulator.')
parser.add_argument('-i', '--identity', type=int, default=0,
                    help='the default result for no arguments '
                         '(default: 0)')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='Sum the integers (default: find the max).')

if __name__ == '__main__':
    args = parser.parse_args()
    print(args.accumulate(args.integers))
```

In order to document the above program:

```
.. autoprogram:: cli:parser
   :prog: cli.py
```

That's it. It will be rendered as:

1.1 cli.py

Process some integers.

```
usage: cli.py [-h] [-i IDENTITY] [--sum] N [N ...]
```

n
An integer for the accumulator.

-h, --help
show this help message and exit

-i <identity>, **--identity** <identity>
the default result for no arguments (default: 0)

--sum
Sum the integers (default: find the max).

If there are subcommands (subparsers), they are rendered to subsections. For example, given the following Python CLI program (say it's `subcmds.py`):

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
subparsers = parser.add_subparsers()

max_parser = subparsers.add_parser('max', description='Find the max.')
max_parser.set_defaults(accumulate=max)
max_parser.add_argument('integers', metavar='N', type=int, nargs='+',
                        help='An integer for the accumulator.')

sum_parser = subparsers.add_parser('sum', description='Sum the integers.')
sum_parser.set_defaults(accumulate=sum)
sum_parser.add_argument('integers', metavar='N', type=int, nargs='+',
                        help='An integer for the accumulator.')

if __name__ == '__main__':
    args = parser.parse_args()
    print(args.accumulate(args.integers))
```

```
.. autoprogram:: subcmds:parser
   :prog: subcmds.py
```

The above reStructuredText will render:

1.2 subcmds.py

Process some integers.

```
usage: subcmds.py [-h] {max,sum} ...
```

-h, --help
show this help message and exit

1.2.1 subcmds.py max

Find the max.

```
usage: subcmds.py max [-h] N [N ...]
```

n
An integer for the accumulator.

-h, --help
show this help message and exit

1.2.2 subcmds.py sum

Sum the integers.

```
usage: subcmds.py sum [-h] N [N ...]
```

n
An integer for the accumulator.

-h, --help
show this help message and exit

If there are argument groups, they can optionally be rendered as subsections, just like subcommands. For example:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='An integer for the accumulator.')

calculator_opts = parser.add_argument_group('Calculator Options')
calculator_opts.add_argument(
    '-i', '--identity', type=int, default=0,
    help='the default result for no arguments ' '(default: 0)')
calculator_opts.add_argument(
    '--sum', dest='accumulate', action='store_const',
    const=sum, default=max,
    help='Sum the integers (default: find the max).')

if __name__ == '__main__':
    args = parser.parse_args()
    print(args.accumulate(args.integers))
```

```
.. autoprogram:: cli_with_groups:parser
    :prog: cli_with_groups.py
    :groups:
```

The above reStructuredText Text will render:

1.3 cli_with_groups.py

Process some integers.

```
usage: cli_with_groups.py [-h] [-i IDENTITY] [--sum] N [N ...]
```

1.3.1 positional arguments

n
An integer for the accumulator.

1.3.2 optional arguments

-h, --help
show this help message and exit

1.3.3 Calculator Options

-i <identity>, **--identity** <identity>
the default result for no arguments (default: 0)

--sum
Sum the integers (default: find the max).

`.. autoprogram:: module:parser`
It takes an import name of the parser object. For example, if `xyz` variable in `abcd.efgh` module refers an `argparse.ArgumentParser` object:

```
.. autoprogram:: abcd.efgh:xyz
```

The import name also can evaluate other any Python expressions. For example, if `get_parser()` function in `abcd.efgh` module creates an `argparse.ArgumentParser` and returns it:

```
.. autoprogram:: abcd.efgh:get_parser()
```

It also optionally takes an option named `prog`. If it's not present `prog` option uses `ArgumentParser` object's prog value.

CHAPTER 2

Additional Options for `.. autoprogram::`

`:groups:` Render argument groups as subsections.

New in version 0.1.5.

`:maxdepth: ##` Only show subcommands to a depth of `##`.

New in version 0.1.3.

`:no_usage_codeblock:` Don't put the usage text in a `.. codeblock:: console` directive.

New in version 0.1.3.

`:start_command: subcommand` Render document for the given subcommand. `subcommand` can be a space separated list to render a sub-sub-...-command.

New in version 0.1.3.

`:strip_usage:` Removes all but the last word in the usage string before the first option, replaces it with '...', and removes an amount of whitespace to realign subsequent lines.

New in version 0.1.3.

CHAPTER 3

Author and license

The *sphinxcontrib.autoprogram* is written by Hong Minhee and distributed under BSD license.

The source code is maintained under the GitHub repository:

```
$ git clone git://github.com/sphinx-contrib/autoprogram.git
```

CHAPTER 4

Changelog

4.1 Version 0.1.6

To be released.

4.2 Version 0.1.5

Released on May 15, 2018.

- New `:groups:` option to render argument groups. [by Lukas Atkinson]

4.3 Version 0.1.4

Released on February 27, 2018.

- Fixed a :rst:dir‘.. autoprogram::‘ bug that raises `AttributeError` during build without `:no_usage_codeblock:` option on Python 2. [Bitbucket issue #168, Bitbucket issue #169]
- Fixed an issue with Sphinx 1.7 which removed `sphinx.util.compat`. [#1, #2 by Zach Riggle]

4.4 Version 0.1.3

Released on October 7, 2016.

- Fixed a bug that descriptions with `RawTextHelpFormatter` had been incorrectly formatted. [Bitbucket PR #123 by Aaron Meurer]
- Fixed crash when metavars is a tuple (i.e. for `nargs > 1`). [Bitbucket PR #112 by Alex Honeywell]

- Fixed usage string for subcommands (subcommands were previously showing the top-level command usage). [Bitbucket PR #112 by Alex Honeywell]
- Added *new options* to `.. autoprogram::` directive: [Bitbucket PR #112 by Alex Honeywell]
  - `maxdepth`
  - `no_usage_codeblock`
  - `start_command`
  - `strip_usage`
- Fixed suppressed arguments (using `argparse.SUPPRESS` flag) to become ignored. [Bitbucket issue #166]

4.5 Version 0.1.2

Released on August 18, 2015.

- Fixed crash with empty fields. [Bitbucket issue #110]
- Fixed `ImportError` with non-module Python scripts (i.e. files not ending with `.py`). [Bitbucket PR #101 by Matteo Bachetti]

4.6 Version 0.1.1

Released on April 22, 2014.

- Omit metavars of `store_const`/`store_true`/`store_false` options.
- Sort subcommands in alphabetical order if Python 2.6 which doesn't have `collections.OrderedDict`.

4.7 Version 0.1.0

Released on March 2, 2014. The first release.

Python Module Index

S

Index

Symbols

A

C

N

S