sphinx-click

Stephen Finucane

Aug 17, 2023

CONTENTS

`sphinx-click` is a Sphinx plugin that allows you to automatically extract documentation from a click-based application and include it in your docs.

CHAPTER

ONE

INSTALLATION

Install the plugin using *pip*:

```
$ pip install sphinx-click
```

Alternatively, install from source by cloning this repo then running *pip* locally:

```
$ pip install .
```

sphinx-click supports both click and asyncclick. If *asyncclick* is found, it will be preferred.

Important: Both the package you're referencing and any dependencies **must be installed**.

CHAPTER

TWO

USAGE

To enable the plugin, add the extension to the list of extensions in your Sphinx *conf.py* file:

```
extensions = ['sphinx_click']
```

Once enabled, *sphinx-click* enables automatic documentation for click-based applications by way of a Sphinx directive.

.. click:: module:parser

Automatically extract documentation from a click-based application and include it in your docs.

```
.. click:: module:parser
   :prog: hello-world
   :nested: full
```

The directive takes the import name of a *click* object as its sole argument. This should be a subclass of `click.core.BaseCommand`, such as `click.Command`, `click.Group`, `click.MultiCommand`, etc.

In addition, the following options are required:

:prog:
The name of your tool (or how it should appear in your documentation). For example, if you run your script as `./boo --opts args` then `:prog:` will be `boo`. If this is not given, the module name is used.

The following options are optional:

:nested:

Whether subcommands should also be shown. One of:

full
List sub-commands with full documentation.

short
List sub-commands with short documentation.

none

Do not list sub-commands.

Defaults to `short` unless `show-nested` (deprecated) is set.

:commands:
Document only listed commands.

:show-nested:
This option is deprecated; use `nested` instead.

The generated documentation includes anchors for the generated commands, their options and their environment variables using the Sphinx standard domain.

2.1 Cross-referencing

As discussed above, the documentation generated by *sphinx-click* includes anchors for the generated commands, their options and their environment variables using the Sphinx standard domain. Specifically, it uses the `program`, `option`, and `envvar` directives.

Options (e.g. `--param`)
The `option` directive can be cross-referenced using the `:option:` role.

Environment variables
The `envvar` directive can be cross-references using the `:ref:` role. *sphinx-click* generates labels in the format `{command_name}-{param_name}-{envvar}`. It is **not** currently possible to use the `:envvar:` role because the default labels generated by Sphinx are not namespaced and will generate conflicts if the same environment variable is used in multiple commands. See issue #26 for more information.

Programs
Sphinx currently does not allow you to cross-reference programs. See Sphinx issue #880 for more information.

2.2 Docstring processing

sphinx-click provides the following additional events:

sphinx-click-process-description(app, ctx, lines)

sphinx-click-process-usage(app, ctx, lines)

sphinx-click-process-options(app, ctx, lines)

sphinx-click-process-arguments(app, ctx, lines)

sphinx-click-process-envvars(app, ctx, lines)

sphinx-click-process-epilog(app, ctx, lines)

Parameters

- **app** – the Sphinx application object
- **ctx** – the `click.Context` object used to generate the description
- **lines** – the lines of the documentation, see below

Events are emitted when sphinx-click has read and processed part of a command's documentation. *lines* is a list of strings – the lines of the documentation that was processed – that the event handler can modify **in place** to change what Sphinx puts into the output.

```
def process_description(app, ctx, lines):
    """Append some text to the "example" command description."""
    if ctx.command.name == "example":
        lines.extend(["Hello, World!", ""])

def setup(app):
    app.connect("sphinx-click-process-description", process_description)
```

2.3 Example

Take the below `click` application, which is defined in the `hello_world` module:

```
import click

@click.group()
def greet():
    """A sample command group."""
    pass

@greet.command()
@click.argument('user', envvar='USER')
def hello(user):
    """Greet a user."""
    click.echo('Hello %s' % user)

@greet.command()
def world():
    """Greet the world."""
    click.echo('Hello world!')
```

To document this, use the following:

```
.. click:: hello_world:greet
  :prog: hello-world
```

By default, the subcommand, `hello`, is listed but no documentation provided. If you wish to include full documentation for the subcommand in the output, configure the `nested` flag to `full`.

```
.. click:: hello_world:greet
  :prog: hello-world
  :nested: full
```

Note: The `nested` flag replaces the deprecated `show-nested` flag.

Conversely, if you do not wish to list these subcommands or wish to handle them separately, configure the `nested` flag to none.

```
.. click:: hello_world:greet
  :prog: hello-world
  :nested: none
```

You can also document only selected commands by using `:commands:` option.

```
.. click:: hello_world:greet
  :prog: hello-world
  :commands: hello
```

You can cross-reference the commands, option and environment variables using the roles provided by the Sphinx standard domain. See *Cross-referencing* for more information.

```
.. click:: hello_world:greet
   :prog: hello-world

The :program:`hello` command accepts a :option:`user` argument. If this is
not provided, the :envvar:`USER` environment variable will be used.
```

Note: Cross-referencing using the `:program:` directive is not currently supported by Sphinx. Refer to the Sphinx issue for more information.

2.3.1 Documenting `CommandCollection`

When building more complex CLI, one might need to bring together multiple groups of commands and make them accessible using a single client with `CommandCollection`. *sphinx-click* renders collection of commands with multiple sections, one for each group listed in the command `sources`. The group names are used as section titles and the help string from the description are used as section description. Thus, a client defined using a `CommandCollection` as `cli` can be rendered using *sphinx-click* and the following directive:

```
.. click:: cli:cli
   :prog: cli
   :nested: full
```

This will render the subcommands of each group in different sections, one for each group in `sources`. An example is provided in *Documenting command collections*.

2.4 Modifying `sys.path`

If the application or script you wish to document is not installed (i.e. you have not installed it with *pip* or run `python setup.py`), then you may need to modify `sys.path`. For example, given the following application:

```
git
  |- git
  |    |- __init__.py
  |    \- git.py
  \- docs
       |- git.rst
       |- index.rst
        \- conf.py
```

then it would be necessary to add the following to `git/docs/conf.py`:

```
import os
import sys
sys.path.insert(0, os.path.abspath('..'))
```

Once done, you could include the following in `git/docs/git.rst` to document the application:

```
.. click:: git.git:cli
   :prog: git
   :nested: full
```

assuming the group or command in `git.git` is named `cli`.

Refer to issue #2 for more information.

CHAPTER

THREE

CONTRIBUTION

We welcome all contributions to *sphinx-click*.

3.1 Support

Open and issue in the issue tracker for all support requests. StackOverflow is also worth considering.

3.2 Reporting Issues

Report all issues in the issue tracker. When doing so, please include version information for:

- Python
- *click*
- *sphinx-click*

3.3 Submitting Patches

All patches should be submitted as pull requests on the GitHub project.

- Include tests if fixing a bug
- Clearly explain what you're trying to accomplish
- Follow **PEP 8**. You can use the *pep8* tox target for this

3.4 Testing

sphinx-click uses *tox* and *unittest* for testing. To run all tests, run:

```
$ tox
```

We support a number of Python versions. To list available environments, run:

```
$ tox --list
```

To run one of these environments, such as *py27* which runs tests under Python 2.7, run:

```
$ tox -e py27
```

CHAPTER

FOUR

CHANGES

4.1 5.0.1

- Fix compatibility with Sphinx 7.2.x
- Add release note for events support
- Process events (#126)
- Add release note for end of Python 3.7 support
- setup.cfg: Add click keyword
- Drop Python 3.7 support
- tox: Use pre-commit for 'style' target
- mypy: Follow imports
- pre-commit: Enable black
- pre-commit: Version bumps
- examples: Add typing info
- Typo
- Add release note for Python 3.11 support
- actions: Test Python 3.11
- Points to alternative to complement Click-based CLI documentation
- Add Sphinx extension classifier
- Add pull request template
- Trivial setup.py, setup.cfg changes
- Render option defaults as literals
- Change target Python version used for black
- actions: Update to latest versions of actions
- Use deep clone for GitHub actions
- docs: Add more examples
- tests: Add more application tests
- Add our first "extension" test
- Switch to pytest for testing

- Run mypy as part of CI job
- Fix typo
- Add release note for auto_envvar_prefix support
- add functionality to also render env-variables that are created via the 'auto_envvar_prefix' option
- Extend pre-wrapped text support to options
- tests: Use Command tests, not Group tests
- tests: Add test for boolean options
- docs: Don't recommend using 'setup.py'
- docs: Indicate support for asyncclick
- asyncclick target
- docs: Remove 'language' configuration
- docs: Add root directory to PATH for docs build
- docs: Remove noise from conf.py
- tox: Enable nitpicky mode for docs builds
- Correct 'get_command' argument type error
- Correctly ignore empty 'commands' parameter
- docs: Correct typo
- Configure mypy
- Add typing hints
- Pass Context, not self, to MultiCommand
- Add test for truncation
- Add Python 3.10 support, drop Python 3.6 support
- Uncap coverage, remove test-requirements.txt
- Update pre-commit hooks

CHAPTER

FIVE

EXAMPLES

5.1 Documenting commands

Consider the following sample application, using `Command`:

```
# file: cli.py
import click


@click.command()
@click.option('--param', envvar='PARAM', help='A sample option')
@click.option('--another', metavar='[FOO]', help='Another option')
@click.option(
    '--choice',
    help='A sample option with choices',
    type=click.Choice(['Option1', 'Option2']),
)
@click.option(
    '--numeric-choice',
    metavar='<choice>',
    help='A sample option with numeric choices',
    type=click.Choice([1, 2, 3]),
)
@click.option(
    '--flag',
    is_flag=True,
    help='A boolean flag',
)
@click.argument('ARG', envvar='ARG')
def cli(
    param: str,
    another: str,
    choice: str,
    numeric_choice: int,
    flag: bool,
) -> None:
    """A sample command."""
    pass
```

This can be documented using *sphinx-click* like so:

```
.. click:: commands.cli:cli
  :prog: cli
  :nested: full
```

The rendered example is shown below.

5.1.1 cli

A sample command.

```
cli [OPTIONS] ARG
```

Options

--param <param>

A sample option

--another <FOO>

Another option

--choice <choice>

A sample option with choices

Options
Option1 | Option2

--numeric-choice <choice>

A sample option with numeric choices

Options
1 | 2 | 3

--flag

A boolean flag

Arguments

ARG

Required argument

Environment variables

PARAM

Provide a default for *--param*

ARG

Provide a default for *ARG*

5.2 Documenting groups

Consider the following sample application, using `Groups`:

```
# file: cli.py
import click


@click.group()
@click.option(
    '--debug',
    default=False,
    is_flag=True,
    help="Output more information about what's going on.",
)
def cli(debug: bool) -> None:
    """A sample command group."""
    pass


@cli.command()
@click.option('--param', envvar='PARAM', help='A sample option')
@click.option('--another', metavar='[FOO]', help='Another option')
def hello(param: str, another: str) -> None:
    """A sample command."""
    pass
```

This can be documented using *sphinx-click* like so:

```
.. click:: groups.cli:cli
  :prog: cli
  :nested: full
```

The rendered example is shown below.

5.2.1 cli

A sample command group.

```
cli [OPTIONS] COMMAND [ARGS]...
```

Options

--debug

Output more information about what's going on.

hello

A sample command.

```
cli hello [OPTIONS]
```

Options

--param <param>
: A sample option

--another <FOO>
: Another option

Environment variables

PARAM

: Provide a default for *--param*

5.3 Documenting command collections

Consider the following sample application, using `CommandCollection`:

```
# file: cli.py
import click


main = click.Group(
    name='Principal Commands',
    help=(
        "Principal commands that are used in ``cli``.\n\n"
        "The section name and description are obtained using the name and "
        "description of the group passed as sources for |CommandCollection|_."
    ),
)


@main.command(help='CMD 1')
def cmd1() -> None:
    print('call cmd 1')


helpers = click.Group(
    name='Helper Commands',
    help="Helper commands for ``cli``.",
)


@helpers.command()
```

(continues on next page)

(continued from previous page)

```
def cmd2() -> None:
    "Helper command that has no option."
    pass


@helpers.command()
@click.option('--user', type=str)
def cmd3(user: str) -> None:
    "Helper command with an option."
    pass


cli = click.CommandCollection(
    name='cli',
    sources=[main, helpers],
    help='Some general info on ``cli``.',
)
```

This can be documented using *sphinx-click* like so:

```
.. click:: commandcollections.cli:cli
  :prog: cli
  :nested: full
```

The rendered example is shown below.

5.3.1 cli

Some general info on `cli`.

```
cli [OPTIONS] COMMAND [ARGS]...
```

Principal Commands

Principal commands that are used in `cli`.

The section name and description are obtained using the name and description of the group passed as sources for `CommandCollection`.

cmd1

CMD 1

```
cli cmd1 [OPTIONS]
```

Helper Commands

Helper commands for `cli`.

cmd2

Helper command that has no option.

```
cli cmd2 [OPTIONS]
```

cmd3

Helper command with an option.

```
cli cmd3 [OPTIONS]
```

Options

--user <user>

See also:

Module click
: This extension assumes you are using `click` to create your command line application.

Module sphinxcontrib.autoprogram
: An equivalent library for use with `argparse`.

INDEX

Symbols

A

C

P