Click utils Documentation

Release 0.2.2.dev0

Sławek Ehlert

November 02, 2016

Contents

Contents:

CHAPTER 1

Click utils

a set of utilites for writing command line programs with Click

- Free software: BSD license
- Documentation: https://click-utils.readthedocs.org.

1.1 Features

- Couple of useful `logging` options and types
- A custom command and option class for printing env vars names (WIP)
- More to come...

CHAPTER 2

Installation

At the command line:

```
$ easy_install click-utils
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv click-utils
$ pip install click-utils
```

CHAPTER 3

Usage

To use Click utils in a project:

```
import click_utils
```

CHAPTER 4

Click Utils API

4.1 click_utils package

4.1.1 Submodules

4.1.2 click_utils.defaults module

4.1.3 click_utils.help module

class click_utils.help.**EnvHelpCommand**(**args*, ***kwargs*)
Bases: click.core.Command

class click_utils.help.**EnvHelpOption**(*param_decls=None*, *show_default=False*, *prompt=False*, *confirmation_prompt=False*, *hide_input=False*, *is_flag=None*, *flag_value=None*, *multiple=False*, *count=False*, *allow_from_autoenv=True*, *type=None*, *help=None*, ***attrs*)
Bases: click.core.Option

get_envvar_names(*ctx*)

get_help_record(*ctx*)

4.1.4 click_utils.logging module

class click_utils.logging.**LogLevelChoice**(**extra_levels*)
Bases: click.types.Choice

A subclass of click.Choice class for specifying a logging level.

Example usage:

```
import click
import click_utils

@click.command()
@click.option('--loglevel', type=click_utils.LogLevelChoice())
def cli(loglevel):
    click.echo(loglevel)
```

This option type returns an integer representation of a logging level:

```
$ cli --loglevel=error
40
```

By default available choices are the lowercased standard `logging` level names (i.e `notset`, `debug`, `info`, `warning`, `error` and `critical`):

```
$ cli --help

Usage: cli [OPTIONS]

Options:
  --loglevel [notset|debug|info|warning|error|critical]
...
```

But you can pass your additional logging levels like this:

```
logging.addLevelName(102, 'My custom level')

@click.command()
@click.option('--loglevel', type=click_utils.LogLevelChoice(101, 102))
def cli(loglevel):
    click.echo(loglevel)

...

$ cli --help

Usage: cli [OPTIONS]

Options:
  --loglevel [notset|debug|info|warning|error|critical|level101|mycustomlevel]
...

$ cli --loglevel=level101
101
```

You can also pass level name in the uppercased form:

```
$ cli --loglevel=WARNING
30
```

Finally, as a special case, user can provide any integer as a logging level:

```
$ cli --loglevel=123
123
```

LOGGING_LEVELS = (('notset', 0), ('debug', 10), ('info', 20), ('warning', 30), ('error', 40), ('critical', 50))

convert(*value*, *param*, *ctx*)

click_utils.logging.**logconfig_callback**(*ctx*, *param*, *value*)

a default callback for invoking:

```
logging.config.fileConfig(value, defaults=None, disable_existing_loggers=False)
```

click_utils.logging.**logconfig_callback_factory**(*defaults=None*, *disable_existing_loggers=False*)

a factory for creating parametrized callbacks to invoke `logging.config.fileConfig`

click_utils.logging.**logconfig_option**(**param_decls*, ***attrs*)

An option to easily configure logging via `logging.config.fileConfig`. This one allows to specify a

file that will be passed as an argument to `logging.config.fileConfig` function. Using this option like this:

```
@click.command()
@click_utils.logconfig_option()
def cli():
    pass
```

is equivalent to:

```
def mycallback(ctx, param, value):
    if not value or ctx.resilient_parsing:
        return
    logging.config.fileConfig(value,
                              defaults=None,
                              disable_existing_loggers=False)

@click.command()
@click.option('--logconfig',
              expose_value=False,
              type=click.Path(exists=True, file_okay=True, dir_okay=False,
                              writable=False, readable=True, resolve_path=True)
              callback=mycallback
              )
def cli():
    pass
```

This option accepts all the usual arguments and keyword arguments as `click.option` (be careful with passing a different `callback` though). Additionally it accepts two extra keyword arguments which are passed to `logging.config.fileConfig`:

- `fileconfig_defaults` is passed as `defaults` argument (default: `None`)
- `disable_existing_loggers` is passed as `disable_existing_loggers` argument (default: `False`)

So you can add logconfig option like this:

```
@click.command()
@click_utils.logconfig_option(disable_existing_loggers=True)
def cli():
    pass
```

`click_utils.logging.`**`logfile_option`**(**param_decls*, ***attrs*)

a specific type of `logger_option` that configures a logging file handler (by default it's a `logging.handlers.RotatingFileHandler`) on a logger (root logger by default).

In addition to all `click_utils.logger_option` and `click.option` arguments it accepts four keyword arguments to control handler creation and format settings.

Parameters

- **`fmt`** – a format (`fmt`) argument for `logging.Formatter` class
- **`datefmt`** – a date format (`datefmt`) argument for `logging.Formatter` class
- **`maxBytes`** – a `maxBytes` argument for `RotatingFileHandler` class
- **`backupCount`** – a `backupCount` argument for `RotatingFileHandler` class

click_utils.logging.**logger_callback_factory**(*logger_name=''*, *handler_cls=None*, *handler_attrs=None*, *formatter_cls=None*, *formatter_attrs=None*, *filters=None*, *level=None*, *ctx_loglevel_attr=None*, *ctx_key=None*)

click_utils.logging.**logger_option**(**param_decls*, ***attrs*)

An abstract option for configuring a specific handler to a given logger (root logger by default). This logger is then attached to a context for further usage. This option accepts all the

Parameters

- **handler_cls** – **required** a logging Handler class. A value of this option will be passed as a first positional argument while creating an instance of the `handler_cls`.
- **handler_attrs** – a dictionary of keyword arguments passed to `handler_cls` while instantiating
- **formatter_cls** – a Formatter class which instance will be added to a handler
- **formatter_attrs** – a dictionary of keyword arguments passed to `formatter_cls` while instantiating
- **filters** – an iterable of filters to add to the logger
- **level** – enforce a minimum logging level on a logger and handler. If this is `None` then a callback will try to retrieve the level from context (e.g. that was stored by `loglevel_option`). If it fails to find it a default level is `logging.WARNING`
- **ctx_loglevel_attr** – an attribute name of the context to **find** a stored logging level (by `loglevel_option` for example)
- **ctx_key** – an attribute name of the context to **store** a logger. Pass a *falsy* value to disable storing (default: `None`).
- **logger_name** – a name of a logger to configure (default: `''` i.e. a root logger)

click_utils.logging.**loglevel_callback_factory**(*ctx_loglevel_attr=None*)

click_utils.logging.**loglevel_option**(**param_decls*, ***attrs*)

Shortcut for logging level option type.

This is equivalent to decorating a function with `option()` with the following parameters:

```
@click.command()
@click.option('--loglevel', type=click_utils.LogLevelChoice())
def cli(loglevel):
    pass
```

It also has a callback that stores a chosen loglevel on the context object. By default the loglevel is stored in attribute called `click_utils_loglevel`

To change the attribute name you can pass an keyword arg to this option called `ctx_loglevel_attr`. This is useful for other options to know what loglevel was set. If you want to disable storing the loglevel on context just pass:

```
@click.command()
@click_utils.loglevel_option(ctx_loglevel_attr=False)
def cli(loglevel):
    click.echo(loglevel)
```

Also this option is eager by default.

4.1.5 Module contents

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/slafs/click-utils/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

5.1.4 Write Documentation

Click Utils could always use more documentation, whether as part of the official Click Utils docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/slafs/click-utils/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *click-utils* for local development.

1. Fork the *click-utils* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/click-utils.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv click-utils
$ cd click-utils/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ tox
```

   To get tox, just pip install it into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/slafs/click-utils/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ tox -- -k <pattern>
```

CHAPTER 6

Credits

6.1 Development Lead

- Sławek Ehlert <slafs.e@gmail.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

CHAPTER 8

0.2.0 (2015-01-30)

- modify `loglevel_option` (is_eager, callback with storing value on context)
- add an abstract `logger_option` and a `logfile_option`

CHAPTER 9

0.1.0 (2015-01-26)

- First release on PyPI.
- initial project structure
- Add LogLevelChoice type
- Add loglevel_option
- Add logconfig_option

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

C

C

E

G

L