clout

Release 0.1.14

Dec 17, 2019

Contents

docs	
code	
tests	
package	

Convert dataclasses into a command-line interface.

CHAPTER 1

Quickstart

To install, use

```
pip install clout
```

Define some dataclasses and convert them into a command-line interface.

```
import attr
import click

import clout


@attr.dataclass
class DB:
    host: str
    port: int


@attr.dataclass
class Config:
    db: DB
    dry_run: bool


cli = clout.Command(Config)

print(cli.build())
```

```
$ myapp --dry-run db --host example.com --port 9999
Config(db=DB(host='example.com', port=9999), dry_run=True)
```

See the full docs for more information: https://clout.readthedocs.io/

CHAPTER 2

Contents

2.1 Clout: Command-Line Object Utility Tool

docs	
code	
tests	
package	

Convert dataclasses into a command-line interface.

2.1.1 Quickstart

To install, use

```
pip install clout
```

Define some dataclasses and convert them into a command-line interface.

```
import attr
import click
```

(continues on next page)

(continued from previous page)

```
import clout


@attr.dataclass
class DB:
    host: str
    port: int


@attr.dataclass
class Config:
    db: DB
    dry_run: bool


cli = clout.Command(Config)

print(cli.build())
```

```
$ myapp --dry-run db --host example.com --port 9999
Config(db=DB(host='example.com', port=9999), dry_run=True)
```

See the full docs for more information: https://clout.readthedocs.io/

2.2 Installation

At the command line:

```
pip install clout
```

2.3 Usage

Convert dataclasses into a command-line interface.

2.3.1 Quickstart

To install, use

```
pip install clout
```

Define some dataclasses and convert them into a command-line interface.

```
import attr
import click

import clout


@attr.dataclass
class DB:
```

(continues on next page)

(continued from previous page)

```
    host: str
    port: int


@attr.dataclass
class Config:
    db: DB
    dry_run: bool


cli = clout.Command(Config)

print(cli.build())
```

```
$ myapp --dry-run db --host example.com --port 9999
Config(db=DB(host='example.com', port=9999), dry_run=True)
```

2.3.2 A decorator

To run a function with the built object, you can use a decorator, `clout.command()`. Using *clout.Command.main()*, the program will exit immediately afterwards.

```
import attr

import clout


@attr.dataclass
class Person:
    name: str = attr.ib(metadata={"clout": {"cli": dict(envvar="EXAMPLE_NAME")}})
    age: int


@clout.command(Person)
def greet(person):
    print(f"Hello, {person.name}!")


if __name__ == "__main__":
    print(greet.main())
```

```
$ EXAMPLE_NAME=Alice python examples/decorator.py --age 21
Hello, Alice!
```

2.3.3 A callback

If you don't want to use a decorator, you can use *clout.Command()* directly, and pass the `callback=` argument.

```
import attr
import click

import clout
```

(continues on next page)

(continued from previous page)

```
@attr.dataclass
class Person:
    name: str
    age: int


def greet(person):
    print(f"Hello, {person.name}!")


cli = clout.Command(Person, callback=greet)

if __name__ == "__main__":
    cli.main()
```

2.3.4 Extended example

Now for a longer example.

- Define a nested configuration object.
- Load the object from the CLI, getting missing values from environment variables, configuration file, and dataclass `default=` values.

Here we define the config file in the Freedesktop standard directory.

```
# ~/.config/myapp/config.toml
[config]
dry_run=true
```

Set an environment variable and run the app.

```
% MYAPP_PRIORITY=2 python examples/long.py --debug  user --name Alice db --host␣
↪example.com --port 9999 user --name Bob
Config(db=DB(host='example.com', port=9999, user=User(name='Bob')), debug=True, dry_
↪run=True, user=User(name='Alice'), priority=2.0, logging=True)
```

The code:

```
import pathlib

import appdirs
import attr
import click
import toml

import clout


@attr.dataclass
class User:
    name: str
```

(continues on next page)

(continued from previous page)

```
@attr.dataclass
class DB:
    host: str
    port: int
    user: User


@attr.dataclass
class Config:
    db: DB
    debug: bool
    dry_run: bool
    user: User
    priority: float = attr.ib(
        default=0,
        metadata={
            "clout": {
                "cli": dict(param_decls=["--priority"], help="App priority value")
            }
        },
    )
    logging: bool = True


APP_NAME = "myapp"


# Read config file.
with open("examples/config.toml") as f:
    CONFIG_FILE_DATA = toml.load(f)


# Read from environment_variables prefixed `MYAPP_`,
# such as MYAPP_PRIORITY=10.
ENVVAR_DATA = clout.load_env(Config, prefix=APP_NAME)


# Combine config file and envvars to set CLI defaults.
CONTEXT_SETTINGS = dict(default_map=clout.DeepChainMap(ENVVAR_DATA, CONFIG_FILE_DATA))

# Define the CLI.
commands = [
    clout.Command(
        name="config",
        type=Config,
        context_settings=CONTEXT_SETTINGS,
        help="Run the app with given configuration object.",
    )
]
cli = click.Group(commands={c.name: c for c in commands})


if __name__ == "__main__":
    # Run the CLI.

    print(cli.main(standalone_mode=False))
```

2.4 Reference

2.4.1 clout package

Submodules

clout.exceptions module

exception clout.exceptions.**CloutException**
Bases: Exception

Base exception for Clout exceptions.

exception clout.exceptions.**MissingInput**(*group*, *string*, *found*)
Bases: *clout.exceptions.CloutException*

Raised for missing command-line arguments.

exception clout.exceptions.**ValidationError**
Bases: *clout.exceptions.CloutException*

Raised for invalid command-line arguments.

Module contents

class clout.**Command**(*type*, **args*, *name=None*, *app_name=None*, *callback=<function Command.<lambda>>*, *params=None*, *context_settings=None*, *epilog=None*, ***kwargs*)
Bases: click.core.Command

A click.Command built from an attr.dataclass() or dataclasses.dataclass().

build()
Return an instance of *self.type*, built from the command line arguments.

main(***kwargs*)
Run the command and exit the program afterwards.

Upcalls directly to click.MultiCommand.main().

clout.**load_env**(*type: Type[CT_co]*, *prefix: str = ''*)
Load environment variables for a class into a dict.

For example, define a class

```
import attr


@attr.dataclass
class Database:
    host: str
    port: int
```

Load the environment variables into a dict, setting the prefix for our app.

```
d = clout.load_env(Database, prefix='MYAPP')
assert d == {'host': 'example.com', 'port': 1234}
```

Run the app with environment variables set.

```
export MYAPP_HOST=example.com
export MYAPP_PORT=1234
```

class clout.**DeepChainMap**(**maps*)

Bases: collections.ChainMap

Combine multiple dicts into a deep mapping.

Lookups that fail in the first dict will be checked in the next one.

```
>>> import clout
>>> maps = [{"a": {}}, {"a": {"b": {}}}, {"a": {"b": {"c": 1337}}}]
>>> dcm = clout.DeepChainMap(*maps)
>>> dcm["a"]["b"]["c"]
1337
```

2.5 Contributing

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

2.5.1 Bug reports

When 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.

2.5.2 Documentation improvements

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

2.5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/python-clout/clout/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 code contributions are welcome :)

2.5.4 Development

To set up *python-clout* for local development:

1. Fork python-clout (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/python-clout.git
```

3. Create a branch for local development:

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

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. 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
```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a file in `changelog.d/` describing the changes. The filename should be `{id}.{type}.rst`, where `{id}` is the number of the GitHub issue or pull request and `{type}` is one of `breaking` (for breaking changes), `deprecation` (for deprecations), or `change` (for non-breaking changes). For example, to add a new feature requested in GitHub issue #1234, add a file called `changelog.d/1234.change.rst` describing the change.

Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

2.6 Authors

A full list of constributors can be found in GitHub's overview.

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

2.7 Changelog

2.7.1 0.1.2 (2019-08-26)

Changes

- First release on PyPI.

—

- genindex
- modindex
- search

Python Module Index

C

Index