Quick and robust CLI creation with Click module

Ilia Meerovich

Red Hat

Why do we need CLI?

- Human readable interface
- Simplifies human interaction
- Simplifies automation development

Common CLI flow

```
>$ cowthink

^C
>$
```

No help message, need to press CTRL C in order to back to shell

'Self explaining' help message

```
>$ cowthink -h
cow{say,think} version 3.03, (c) 1999 Tony Monroe
Usage: cowthink [-bdgpstwy] [-h] [-e eyes] [-f cowfile]
          [-l] [-n] [-T tongue] [-W wrapcolumn] [message]
>$
```

That doesn't help

```
>$ cowthink -e foo
foo
;
^C
>$
```

after reading man page and stackoverflow

```
>$ cowthink -e YY -T ff moo
 _____
( moo )
 -----
        o   ^__^
         o  (YY)\_______
            (__)\       )\/\
             ff ||----w |
                ||     ||
```

Let's take a look on widely used CLI approaches

Implementing greeting tool

Direct usage of command line parameters

```
#! /usr/bin/env python
import sys

def greeter(command, name):
    print "{cmd} {name}!".format(cmd=command, name=name)

if __name__ == "__main__":
    greeter(sys.argv[1], sys.argv[2])
```

- Fast to develop
- Unreliable

getopt

```
......
    try:
        opts, args = getopt.getopt(
                        sys.argv[1:], "c:n:h", ["command=", "name=", "help"])
    except getopt.GetoptError, err:
        print(err)
        sys.exit(-1)

    for o, a in opts:
        if o in ("-c", "--command"):
            command = a
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-n", "--name"):
            name = a
        else:
            assert False, "unhandled option"
            sys.exit(-1)

    argc = len(sys.argv)
    if argc != 5:
        usage()
        sys.exit()

    greeter(command, name)
```

- Allows arguments parsing
- Doesn't contain inbox support for default parameters, validation, nesting, invocation

argparse

```python
#! /usr/bin/env python
import argparse

def hello(args):
    print('Hello, {0}!'.format(args.name))


def goodbye(args):
    print('Goodbye, {0}!'.format(args.name))

parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers()

hello_parser = subparsers.add_parser('hello')
hello_parser.add_argument('name')
hello_parser.set_defaults(func=hello)

goodbye_parser = subparsers.add_parser('goodbye')
goodbye_parser.add_argument('name')
goodbye_parser.set_defaults(func=goodbye)

if __name__ == '__main__':
    args = parser.parse_args()
    args.func(args)
```

- Allows default parameters,nesting, expansion and invocation
- Has built-in magic behavior to guess if something is an argument or an option.

Click

```
#! /usr/bin/env python
import sys

def greeter(command, name):
    print "{cmd} {name}!".format(cmd=command, name=name)

if __name__ == "__main__":
    greeter(sys.argv[1], sys.argv[2])
```

```
#! /usr/bin/env python
import click

@click.command()
@click.help_option('--help', '-h')
@click.option('-c', '--command')
@click.option('-n', '--name')
def greeter(command, name):
    click.echo("{cmd} {name}".format(cmd=command, name=name))

if __name__ == "__main__":
    greeter()
```

```
>$ ~/pycon/greeter_basic_click.py -h
Usage: greeter_basic_click.py [OPTIONS]

Options:
  -h, --help          Show this message and exit.
  -c, --command TEXT
  -n, --name TEXT
```

Click

```python
import click


@click.group()
def greeter():
    pass


@greeter.command()
@click.argument('name')
def hello(**kwargs):
    print('Hello, {0}!'.format(
        kwargs['name']))


@greeter.command()
@click.argument('name')
def goodbye(**kwargs):
    print('Goodbye, {0}!'.format(
        kwargs['name']))

if __name__ == '__main__':
    greeter()
```

```python
import argparse

def hello(args):
    print('Hello, {0}!'.format(args.name))


def goodbye(args):
    print('Goodbye, {0}!'.format(args.name))

parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers()

hello_parser = subparsers.add_parser('hello')
hello_parser.add_argument('name')
hello_parser.set_defaults(func=hello)

goodbye_parser = subparsers.add_parser('goodbye')
goodbye_parser.add_argument('name')
goodbye_parser.set_defaults(func=goodbye)

if __name__ == '__main__':
    args = parser.parse_args()
    args.func(args)
```

Why Click?

- Easier to expand and maintain than Argparse.
- Allows to create CLI with much less code than Argparse.
- Similar or better end-user experience than with Argparse.
- No black magic that exists in Argparse.
- Supports both Python 2 and 3
- Code is much more readable
- Inbox support for autocompletion

Example from DevOps life

Consolidation of collection of scripts that responsible for Jenkins related stuff to one tool with powerful CLI. Requirements:

- nested help messages
- bash completion ready
- easy to maintain and expand
- back-end independent from front-end

CLI outline

ci-tool [OPS] COMMAND [OPS] OPCODE [OPS]

For example:

- ci-tool [ops] jslave [ops] setup/teardown [ops]
- ci-tool [ops] test-bed [ops] setup/teardown [ops]

CLI structure - CLI script

```
import click

from lib.cli_common import BaseCLI
# lib.cli_common.CLIMeta takes care for registration of bootstrap stuff
# so import here is in order to include cli sub groups
import cli_interfaces.jenkins_slave  # noqa
import cli_interfaces.jenkins_master  # noqa

class CLI(BaseCLI):

    SUB_GROUPS_REGISTRY = []

    @classmethod
    def register_cli(cls):
        for group in cls.SUB_GROUPS_REGISTRY:
            cls.cli.add_command(group)

    @staticmethod
    @click.group(subcommand_metavar='COMMAND [OPTIONS] OPCODE [OPTIONS]',
                 context_settings=BaseCLI.CONTEXT_SETTINGS)
    @click.option('--project_defaults',
                  type=click.Path(),
                  help='path to project defaults conf file')
    @BaseCLI.context()
    def cli(ctx, **kwargs):
        """
        ci-tool help message
        """
        ctx.config_dict.update(**kwargs)


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

CLI structure - CLI extension

```python
#!/usr/bin/env python
import click
from lib.cli_common import BaseCLI


class JenkinsMasterCLI(BaseCLI):

    SUB_GROUP_COMMANDS = ['setup']
    SUB_GROUP_NAME = 'jenkins_master'

    @staticmethod
    @click.group(subcommand_metavar='COMMAND [OPTIONS]',
                 context_settings=BaseCLI.CONTEXT_SETTINGS)
    @BaseCLI.context()
    def jenkins_master(ctx, **kwargs):
        """
        jenkins_master provisioner help message
        """
        ctx.config_dict.update(**kwargs)
        # DEBUG INFO
        click.echo(ctx.config_dict)

    @staticmethod
    @click.command()
    @BaseCLI.context()
    def setup(ctx, **kwargs):
        """
        setup jenkins_master help message
        """
        ctx.config_dict.update(**kwargs)
        # DEBUG INFO
        click.echo(ctx.config_dict)
```

CLI structure - BaseCLI

```python
class BaseCLI(object):
    """
    Base class for all CLI providers
    """

    __metaclass__ = CLIMeta

    CONTEXT_SETTINGS = dict(help_option_names=['-h', '--help'])

    @classmethod
    def register_cli(cls):
        """
        Registration logic of subgroups, should be overwritten in main CLI
        """
        for cmd in cls.SUB_GROUP_COMMANDS:
            getattr(cls, cls.SUB_GROUP_NAME).add_command(getattr(cls, cmd))

    @staticmethod
    def context():
        """
        Configuration context holder
        """
        return click.make_pass_decorator(Context, ensure=True)
```

CLI structure - CLIMeta

```python
class CLIMeta(type):

    SUB_GROUPS_REGISTRY = []

    def __new__(cls, *args, **kwargs):
        new_cls = super(CLIMeta, cls).__new__(cls, *args, **kwargs)

        # collect sub groups
        if new_cls.__name__ not in ['CLI', 'BaseCLI']:
            cls.SUB_GROUPS_REGISTRY.append(
                getattr(new_cls, new_cls.SUB_GROUP_NAME))
        # init context
        if new_cls.__name__ == 'BaseCLI':
            new_cls.context()
        # pass sub groups registry to main cli
        else:
            new_cls.SUB_GROUPS_REGISTRY = cls.SUB_GROUPS_REGISTRY

        # register cli subgroup methods and cli subgroups
        if new_cls.__name__ != 'BaseCLI' and hasattr(new_cls, 'register_cli'):
            new_cls.register_cli()

        return new_cls
```

CLI structure - Context

```
class Context(object):
    """
    Configuration Context
    """
    def __init__(self):
        self.config_dict = {}
```

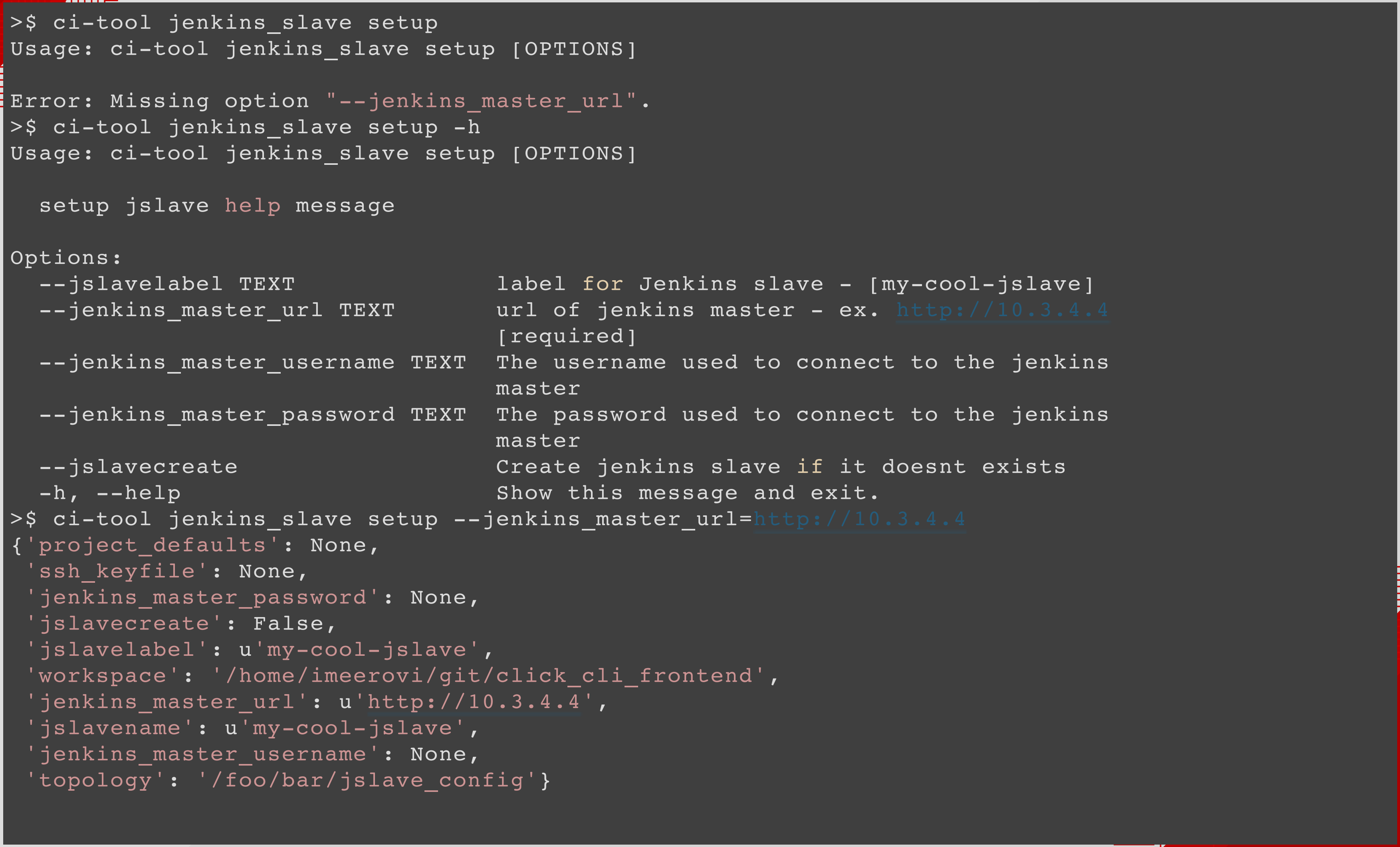
CLI structure - setup.py

```python
import os
from setuptools import setup

setup(
    name='ci-tool',
    version='0.1',
    py_modules=['ci_tool',
                'lib.cli_common',
                'cli_interfaces.jenkins_slave',
                'cli_interfaces.jenkins_master'],
    include_package_data=True,
    install_requires=[
        'click',
    ],
    entry_points='''
        [console_scripts]
        ci-tool=ci_tool:CLI.cli
    ''',
)
# ugly bash completion hack according to http://click.pocoo.org/6/bashcomplete/
with open('%s/.bashrc' % os.path.expanduser('~'), 'a') as f:
    f.write('eval "$(_CI_TOOL_COMPLETE=source ci-tool)"\n')
```

CLI example - user experience

```
>$ ci-tool
Usage: ci-tool [OPTIONS] COMMAND [OPTIONS] OPCODE [OPTIONS]

  ci-tool help message

Options:
  --project_defaults PATH  path to project defaults conf file
  -h, --help               Show this message and exit.

Commands:
  jenkins_master  jenkins_master provisioner help message
  jenkins_slave   jenkins_slave provisioner help message
>$ ci-tool jenkins_
jenkins_master  jenkins_slave
>$ ci-tool jenkins_slave
Usage: ci-tool jenkins_slave [OPTIONS] COMMAND [OPTIONS]

  jenkins_slave provisioner help message

Options:
  --topology PATH     path/to/file - [/foo/bar/jslave_config]
  --ssh_keyfile PATH  path to keyfile
  --jslavename TEXT   name of Jenkins slave - [my-cool-jslave]
  --workspace PATH    /path/to/workspace - ex. /var/lib/jenkins
  -h, --help          Show this message and exit.

Commands:
  setup     setup jslave help message
  teardown  teardown jslave help message
```

CLI example - user experience



```
>$ ci-tool jenkins_slave setup
Usage: ci-tool jenkins_slave setup [OPTIONS]

Error: Missing option "--jenkins_master_url".
>$ ci-tool jenkins_slave setup -h
Usage: ci-tool jenkins_slave setup [OPTIONS]

  setup jslave help message

Options:
  --jslavelabel TEXT              label for Jenkins slave - [my-cool-jslave]
  --jenkins_master_url TEXT       url of jenkins master - ex. http://10.3.4.4
                                  [required]
  --jenkins_master_username TEXT  The username used to connect to the jenkins
                                  master
  --jenkins_master_password TEXT  The password used to connect to the jenkins
                                  master
  --jslavecreate                  Create jenkins slave if it doesnt exists
  -h, --help                      Show this message and exit.
>$ ci-tool jenkins_slave setup --jenkins_master_url=http://10.3.4.4
{'project_defaults': None,
 'ssh_keyfile': None,
 'jenkins_master_password': None,
 'jslavecreate': False,
 'jslavelabel': u'my-cool-jslave',
 'workspace': '/home/imeerovi/git/click_cli_frontend',
 'jenkins_master_url': u'http://10.3.4.4',
 'jslavename': u'my-cool-jslave',
 'jenkins_master_username': None,
 'topology': '/foo/bar/jslave_config'}
```

CLI example - summary

- Provides interface that allows expanding of CLI without knowledge of the whole CLI
- Shows how we can create back-end independent CLI with Click
- Shows that complex CLI code could be readable

Click - Summary

- From developer point of view:
  - Readable and extendable code
  - Function help messages are CLI help messages
  - Function parameters could be CLI parameters
  - Nesting with ease
  - Bash completion built in
  - No black magic
- From user point of view:
  - Formatted help messages
  - Bash completion
  - Expected behaviour

Useful links

- http://click.pocoo.org/6/
- https://realpython.com/blog/python/comparing-python-command-line-parsing-libraries-argparse-docopt-click/
- https://github.com/iluxame/click_cli_frontend

We're Looking for you!

https://www.redhat.com/en/jobs

Ilia Meerovich

imeerovi@redhat.com

Q&A