Command Tree Documentation

Release 0.7.0

Deverto Systems LTD

Jan 22, 2018

Contents

CHAPTER 1

Summary

The command-tree is a lightweight framework to build multi level command line interfaces by using the python builtin argparse module.

The main objectives are:

- Full `argparse.ArgumentParser` compatibility.
- Use as litle as possible code to build the tree, but preserve the argparse flexibiltiy.
- The class and function structure must looks as the cli tree.
- Decorators, decorators everywhere. We love decorators, so we use as often as possible.

CHAPTER 2

Usage

To understand how to use it see this example first: *Basic example*. This page contains an example implemented in 2 ways: one with argparse and one with command-tree.

There is a page where we dissected the basic command-tree example and add a lot of comment to explains the code: *Basic example with a lot of comments*

CHAPTER 3

Config

Some very cool extra features can be configured via the Config class. For further information see `command_tree.config.Config`. For a very simple example see *Basic config example*.

CHAPTER 4

Examples

4.1 Basic example

4.1.1 argparse

Listing 4.1: basic-argparse.py

```
from argparse import ArgumentParser

parser = ArgumentParser()

subparsers1 = parser.add_subparsers(dest = 'subcommand')

command1parser = subparsers1.add_parser('command1')

command1parser.add_argument("arg1")

def command1_handler(args):
    return int(args.arg1) / 2

command1parser.set_defaults(func = command1_handler)

command2parser = subparsers1.add_parser('command2')

command2parser.add_argument("arg1")

def command2_handler(args):
    return int(args.arg1) * 2

command2parser.set_defaults(func = command2_handler)

args = parser.parse_args()

print(args.func(args))
```

4.1.2 command-tree

Listing 4.2: basic.py

```
from command_tree import CommandTree

tree = CommandTree()

@tree.root()
class Root(object):

    @tree.leaf()
    @tree.argument()
    def command1(self, arg1):
        return int(arg1) / 2

    @tree.leaf()
    @tree.argument()
    def command2(self, arg1):
        return int(arg1) * 2

print(tree.execute())
```

4.2 Basic example with a lot of comments

```
# import the CommandTree. The import is important.
from command_tree import CommandTree

# Create the CommandTree instance. This is mandatory. Every decorator must be used from
# this instance.
tree = CommandTree()

# The root decorator is mandatory. Use in the top of the tree, at the root class.
# Only one root allowed per CommandTree instance. This a special, the top-level node.
@tree.root()
# This is a handler class. Must be derived from object, but no other ancestor is
# neccessary.
class Root(object):

    # Constructor is not neccessary if there is no argument for the node,
    # but may have if you want to initialize your personal stuffs.

    # Mark this function as a leaf. Must be used under a node. By default the function
    # name used as parser name. Every parameter in the leaf arguments are passed
    # to the ArgumentParser ctor.
    @tree.leaf()
    # We have an argument here! IMPORTANT: you have to use the argument decoator
    # as many as argument has the handler fuction. (this case: command1)
    # All positional and keyword arguments are passed to ArgumentParser.add_argument
    # function
    @tree.argument()
    # The leaf's handler function. When the user execute the `script.py command1 42` the
    # command-tree will call this function (after instantiate the parent node classes)
```

```
    def command1(self, arg1):
        # this return value will be returned by the CommandTree.execute
        return int(arg1) / 2

    @tree.leaf()
    @tree.argument()
    def command2(self, arg1):
        return int(arg1) * 2

# After you built the tree try to execute. The CommandTree will build the argparse tree,
# call the ArgumentParser.parse_args and search for the selected handler.
print(tree.execute())
```

4.3 Commands in files

This example shows how to distribute your code if you want to separate the node handler classes to external files.

Listing 4.3: tree.py: Instantiate the CommandTree class in an extra file

```
from command_tree import CommandTree

tree = CommandTree()
```

Listing 4.4: node1.py: Define the node1

```
from tree import tree

@tree.node()
class Node1(object):

    @tree.leaf()
    @tree.argument()
    def divide(self, arg1):
        return int(arg1) / 2
```

Listing 4.5: node2.py: Define the node2

```
from tree import tree

@tree.node()
class Node2(object):

    @tree.leaf()
    @tree.argument()
    def multiply(self, arg1):
        return int(arg1) * 2
```

Listing 4.6: power.py: If you are brave enough, you can define leafs in separated files. But beware of the 'self' argument!

```
from tree import tree

@tree.leaf()
@tree.argument()
```

```
def power(self, arg1):
    return int(arg1) * int(arg1)
```

Listing 4.7: cli.py: This is the 'root' file where you collect the nodes and leafs from other files.

```
from tree import tree
from node1 import Node1
from node2 import Node2
from power import power as Power

@tree.root()
class Root(object):

    node1 = Node1
    node2 = Node2

    power = Power

print(tree.execute())
```

4.4 Basic config example

Listing 4.8: config.py

```
from command_tree import CommandTree, Config

config = Config(change_underscores_to_hyphens_in_names = True)

tree = CommandTree(config)

@tree.root()
class Root(object):

    @tree.leaf()
    def command_one(self):
        return 42

print(tree.execute())
```

4.5 Groups

4.5.1 Argument groups

It has been implement the simple argument group as described as `argparse.ArgumentParser.add_argument_group()` . The parameters of the `command_tree.groups.ArgumentGroup` are the exact same like the argparse one.

Usage:

Listing 4.9: groups/arg_group.py

```
from command_tree import CommandTree, ArgumentGroup

tree = CommandTree()

@tree.root()
class Root(object):

    grp1 = ArgumentGroup(tree, "platypus")

    @tree.leaf()
    @grp1.argument("--foo")
    @grp1.argument("--bar")
    def add(self, foo = 42, bar = 21):
        return foo + bar

print(tree.execute())
```

Result:

```
$ python groups/arg_group.py add -h

usage: arg_group.py add [-h] [--foo FOO] [--bar BAR]

optional arguments:
-h, --help  show this help message and exit

platypus:
--foo FOO
--bar BAR
```

4.5.2 Mutual exclusion

It has been implement the mutually exclusive argument group as described as `argparse.ArgumentParser.add_mutually_exclusive_group()` . The parameters of the `command_tree.groups.MutuallyExclusiveGroup` are the exact same like the argparse one.

Usage:

Listing 4.10: groups/mutex.py

```
from command_tree import CommandTree, MutuallyExclusiveGroup

tree = CommandTree()

@tree.root()
class Root(object):

    grp1 = MutuallyExclusiveGroup(tree, required = True)

    @tree.leaf()
    @grp1.argument("--foo")
    @grp1.argument("--bar")
    def add(self, foo = 42, bar = 21):
        return foo + bar
```

```

print(tree.execute())
```

Result:

```
$ python groups/mutex.py add --foo 1 --bar 2

usage: mutex.py add [-h] (--foo FOO | --bar BAR)
mutex.py add: error: argument --bar: not allowed with argument --foo
```

4.5.3 Mutex group in argument group

If you want to add a mutex group into an argument group, it's possible:

Listing 4.11: groups/mutex_in_arg.py

```
from command_tree import CommandTree, ArgumentGroup, MutuallyExclusiveGroup

tree = CommandTree()

@tree.root()
class Root(object):

    arg_grp = ArgumentGroup(tree, "platypus")

    mutex = MutuallyExclusiveGroup(tree, required = True, argument_group = arg_grp)

    @tree.leaf()
    @mutex.argument("--foo")
    @mutex.argument("--bar")
    def add(self, foo = 42, bar = 21):
        return foo + bar

print(tree.execute())
```

4.6 Parsing the docstring for help

4.6.1 Google (default) format

The command-tree by default can parse the classes and function docstring for search help for commands and arguments. The default comment format defined by the Google. For more info, see https://google.github.io/styleguide/pyguide.html#Comments.

Listing 4.12: help.py

```
from command_tree import CommandTree

tree = CommandTree()

@tree.root()
class Root(object):

    @tree.leaf()
```

```
    @tree.argument()
    def command1(self, arg1):
        """Help for command1

        Args:
            arg1: help for arg1
        """
        return int(arg1) / 2

    @tree.leaf()
    @tree.argument()
    def command2(self, arg1):
        """Help for command2

        Args:
            arg1: help for arg1
        """
        return int(arg1) * 2

print(tree.execute())
```

python examples/help.py -h

```
usage: help.py [-h] subcommand ...

positional arguments:
subcommand
    command1  Help for command1
    command2  Help for command2

optional arguments:
-h, --help  show this help message and exit
```

python examples/help.py command1 -h

```
usage: help.py command1 [-h] arg1

positional arguments:
arg1        help for arg1

optional arguments:
-h, --help  show this help message and exit
```

4.6.2 Custom format

But if you want to use an other comment format, you can specify a custom comment parser in the config:

Listing 4.13: help-custom.py

```
from command_tree import CommandTree, Config
from command_tree.doc_string_parser import DocStringInfo, ParserBase

class MyDocStringParser(ParserBase):

    def parse(self, content):
        info = DocStringInfo()
```

```

        # parse the content and put into a DocStringInfo instance ...

        return info

config = Config(docstring_parser = MyDocStringParser())

tree = CommandTree(config)

@tree.root()
class Root(object):

    @tree.leaf()
    @tree.argument()
    def command1(self, arg1):
        """Help for command1

        Parameters
        ----------
        arg1 : int
            Description of arg1
        """
        return int(arg1) / 2

print(tree.execute())
```

4.7 The node handler

TODO

```
from command_tree import CommandTree

tree = CommandTree()

@tree.optional
@tree.root()
@tree.argument('-V', action = 'store_true')
class Root(object):

    def __init__(self, version):
        pass

    @tree.leaf()
    def command1(self):
        return "1"

    @tree.node_handler
    def handler(self, version):
        if version:
            return "42.0"

print(tree.execute())
```

CHAPTER 5

Other

5.1 Alternatives

Before we started to develop the command-tree we searched for other solutions but did not found a good alternative.

5.1.1 argparse

- **pro:**
  - builtin
- **contra:**
  - very low-level
  - needs lots of code to build a tree

5.1.2 click

- https://github.com/pallets/click
- **pro:**
  - ~3800 star
  - nested arguments (like argparser's subparsers)
  - very richfull
- **contra:**
  - build a nested struct with flat struct
  - positional arguments cannot have help ("Arguments cannot be documented this way. This is to follow the general convention of Unix tools of using arguments for only the most necessary things and to document them in the introduction text by referring to them by name.")

  - not argparse compatible

5.1.3 docopt

- https://github.com/docopt/docopt
- **pro:**
  - ~4800 star
  - multilang
- **contra:**
  - no support for multi level commands

5.1.4 arghandler

- https://github.com/druths/arghandler
- **contra:**
  - wut

CHAPTER 6

Indices and tables

- genindex
- modindex
- search