Clams Documentation

Release 0.0.4

Nick Zarczynski

November 28, 2017

Contents

CHAPTER 1

Overview

Create simple, nested, command-line interfaces with Clams (Command Line Applications Made Simple).

CHAPTER 2

Overview

Clams

Create simple, nested, command-line interfaces.

Example

A simple example with `hello` and `goodbye` subcommands. This can be found at /demo/salutation.py.

```
from clams import arg, Command

salutation = Command('salutation')

@salutation.register('hello')
@arg('name', nargs='?')  # <== same interface as argparse's `add_argument`
def handler(name):
    print 'Hello %s' % name or 'Nick'

@salutation.register('goodbye')
@arg('name', nargs='?')
def handler(name):
    print 'Goodbye %s' % name or 'Nick'

if __name__ == '__main__':
    salutation.init()
    salutation.parse_args()
```

Usage:

```
$ cd demo

$ ./salutation.py hello
Hello Nick

$ ./salutation.py hello Jason
Hello Jason

$ ./salutation.py goodbye "my friend."
Goodbye my friend.
```

For more in-depth examples, see the /demo directory.

class clams.**Command**(*name*, *title=''*, *description=''*)

Bases: object

add_argument_tuple(*arg_tuple*)

Add a new argument to this Command.

Parameters **arg_tuple** (*tuple*) – A tuple of `(*args, **kwargs)` that will be passed to `argparse.ArgumentParser.add_argument`.

add_handler(*handler*)

Add a handler to be called with the parsed argument namespace.

Parameters **handler** (*function*) – A function that accepts the arguments defined for this command.

add_subcommand(*command*)

Add a new subcommand to this Command.

Parameters **command** (Command) – The Command instance to add.

init()

Initialize/Build the `argparse.ArgumentParser` and subparsers.

This must be done before calling the `parse_args` method.

parse_args(*args=None*, *namespace=None*)

Parse the command-line arguments and call the associated handler.

The signature is the same as argparse.ArgumentParser.parse_args.

Parameters

- **args** (*list*) – A list of argument strings. If `None` the list is taken from `sys.argv`.
- **namespace** (*argparse.Namespace*) – A Namespace instance. Defaults to a new empty Namespace.

Returns

- *The return value of the handler called with the populated Namespace as*
- *kwargs.*

register(*name=None*)

Decorator to (create and) register a command from a function.

Parameters **name** (*Optional[str]*) – If present, create a command and register it (see `register_command`).

Example

```
mygit = Command(name='status')

@mygit.register(name='status')
def status():
    print 'Nothing to commit.'

@mygit.register()
@command(name='log')
def log():
    print 'Show logs.'
```

register_command(*name*)

Decorator to create and register a command from a function.

Parameters **name** (*str*) – The name given to the registered command.

Example

```
mygit = Command(name='mygit')

@mygit.register_command(name='status')
def status():
    print 'Nothing to commit.'
```

clams.**arg**(**args*, ***kwargs*)

Annotate a function by adding the args/kwargs to the meta-data.

This appends an Argparse "argument" to the function's `ARGPARSE_ARGS_LIST` attribute, creating `ARGPARSE_ARGS_LIST` if it does not already exist. Aside from that, it returns the decorated function unmodified, and unwrapped.

The "arguments" are simply `(args, kwargs)` tuples which will be passed to the Argparse parser created from the function as `parser.add_argument(*args, **kwargs)`.

argparse.ArgumentParser.add_argument should be consulted for up-to-date documentation on the accepted arguments. For convenience, a list has been included here.

Parameters

- **name/flags** (*str or list*) – Either a name or a list of (positional) option strings, e.g. ('foo') or ('-f', '–foo').
- **action** (*str*) – The basic type of action to be taken when this argument is encountered at the command line.
- **nargs** (*str*) – The number of command-line arguments that should be consumed.
- **const** – A constant value required by some action and nargs selections.
- **default** – The value produced if the argument is absent from the command line.
- **type** (*type*) – The type to which the command-line argument should be converted.
- **choices** – A container of the allowable values for the argument.
- **required** (*bool*) – Whether or not the command-line option may be omitted (optionals only).
- **help** (*str*) – A brief description of what the argument does.
- **metavar** (*str*) – A name for the argument in usage messages.
- **dest** (*str*) – The name of the attribute to be added to the object returned by parse_args().

Example

```
@command(name='echo')
@arg('-n', '--num', type=int, default=42)
@arg('-s', '--some-switch', action='store_false')
@arg('foo')
def echo(foo, num, some_switch):
    print foo, num
```

```
>>> echo_subcommand = mycommand.add_subcommand(echo)
>>> mycommand.init()
>>> mycommand.parse_args(['echo', 'hi', '-n', '42'])
hi 42
```

See also:

argparse.ArgumentParser.add_argument

clams.**command**(*name*)

Create a command, using the wrapped function as the handler.

Parameters **name** (*str*) – Name given to the created Command instance.

Returns A new instance of Command, with handler set to the wrapped function.

Return type *Command*

clams.**register**(*command*)

Register a command with a parent command.

The `register` decorator decorates a Command instance (not a function). It is intended to be used with the `command` decorator (which decorates a function and returns a Command instance).

Parameters **comand** (Command) – The parent command.

Example

```
mygit = Command(name='status')

@register(mygit)
@command('status')
def status():
    print 'Nothing to commit.'
```

clams.**register_command**(*parent_command*, *name*)

Create and register a command with a parent command.

Parameters

- **parent_comand** (Command) – The parent command.
- **name** (*str*) – Name given to the created Command instance.

Example

```
mygit = Command(name='status')

@register_command(mygit, 'status')
def status():
    print 'Nothing to commit.'
```

Contributing

Eventually I'll add some instructions and more information about contributing. In the meantime, if you want to get involved just send an email to nick@unb.services, a pull-request to github, or add an issue to our tracker.

Development Tools

Clams uses unb-cli to simplify and standardize common development and project management tasks.

Unfortunately, unb-cli is not (yet) available as an open source package.

Docs for the Docs

Documentation is managed by Sphinx.

Documentation is built using `unb-cli build sphinx`.

CHAPTER 3

Indices and tables

- modindex
- genindex
- search

C

A

C

I

P

R