Opster

Release 4.2

Oct 21, 2018

Contents

Opster is a command line parser, intended to make writing command line applications easy and painless. It uses built-in Python types (lists, dictionaries, etc) to define options, which makes configuration clear and concise. Additionally, Opster supports parsing arguments for an application that uses subcommands (i.e. `hg commit` or `svn update`).

- Page on PyPI: http://pypi.python.org/pypi/opster/
- Repository: http://hg.piranha.org.ua/opster/
- Requires at least Python 2.6

CHAPTER 1

More documentation

1.1 Changelog

1.1.1 4.2 (2018.10.21)

- When function arguments (which become command line options) had underscores in them, they appeared in help as unnamed argument (GH-60). No more.

1.1.2 4.1 (2013.08.19)

- Improve guessing abilities under Python 3 (options in keyword arguments and keyword only arguments are combined now).

1.1.3 4.0 (2013.06.03)

- Infinitely nested subcommands.
- Tuple options (one of given enumerated values).
- A lot of fixes.

Most of changes were done by Oscar Benjamin.

1.1.4 3.7 (2012.05.03)

- Fixed name in usage on Windows.
- Improved and documented preparsing of global options when using subcommands (GH-25).

1.1.5 3.6 (2012.04.23)

- Now commands can have `-h` option (GH-2).
- Better Windows compatibility (GH-18, GH-20).
- Refactored internal options representation with easier introspectability (GH-19).
- Tests support Python 3 (GH-21).

Thanks for this release are going to Oscar Benjamin, every point in this release is his work.

1.1.6 3.5 (2012.03.25)

- Added command.Error to facilitate easy exits from scripts (GH-12).
- Fixed opster.t output.

1.1.7 3.4 (2012.01.24)

- Fix for installation issue (MANIFEST.in wasn't included).
- Fix for pep8.py complaints (most of them).
- Fix for script name when calling as a command (and not a dispatcher) (GH-4).
- Fix for some 2to3 issues (GH-5).
- Fixed bug with empty arguments for `opster.command` (GH-6).
- opster.t is now able to run under `dash`.
- More output encodings supported (GH-7).
- Coverage support for cram tests (GH-8).
- Fixed combination of varargs and option name with underscore (GH-10).

1.1.8 3.3 (2011.09.04)

- Multicommands: ability to specify global options before specifying name of command

1.1.9 3.2 (2011.08.27)

- Fix for `TypeError:  func() got multiple values for 'argument'`

1.1.10 3.1 (2011.08.27)

- Better aliases support.
- Fixes for options and usage discovery.
- Fix for error handling of dictionary options in multicommands.
- Fix for help not working when global options are defined.

1.1.11 3.0 (2011.08.14)

- **Backward incompatible** Single commands now don't attempt to parse. arguments if you call them. Use `function.command()` attribute (much like earlier `function.help()`) to parse arguments now.
- Switch to Python 2.6.
- Ability to have few subcommand dispatchers in a single runtime (no single global `CMDTABLE` dictionary anymore).

1.1.12 2.2 (2011.03.23)

- adjust indentation level in multiline docstrings (compare 1 and 2)
- small fix for internal getopt exception handling

1.1.13 2.1 (2011.01.23)

- fix help display in case middleware returns original function

1.1.14 2.0 (2011.01.23)

- fix help display when there is no __doc__ declared for function
- `dict` type handling
- `.help()` attribute for every function, printing help on call

1.1.15 1.2 (2010.12.29)

- fix option display for a list of subcommands if docstring starts with a blank line

1.1.16 1.1 (2010.12.07)

- _completion was failing to work when global options were supplied to command dispatcher

1.1.17 1.0 (2010.12.06)

- when middleware was used and command called without arguments, instead of help, traceback was displayed

1.1.18 0.9.13 (2010.11.18)

- fixed exception handling (cleanup previous fix, actually)
- display only name of application, without full path

1.1.19 0.9.12 (2010.11.02)

- fixed trouble with non-ascii characters in docstrings

1.1.20 0.9.11 (2010.09.19)

- fixed exceptions handling
- autocompletion improvements (skips middleware, ability of options completion)

1.1.21 0.9.10 (2010.04.10)

- if default value of an option is a fuction, always call it (None is passed in case when option is not supplied)
- always call a function if it's default argument for an option
- some cleanup with better support for python 3
- initial support for autocompletion (borrowed from PIP)

1.1.22 0.9 - 0.9.9 (since 2009.07.13)

Ancient history ;-)

1.2 Overview

1.2.1 Options

Options are defined as keyword arguments to a script's main function:

```python
from opster import command

@command(usage='[-l HOST] DIR')
def main(dirname,
         listen=('l', 'localhost', 'ip to listen on'),
         port=('p', 8000, 'port to listen on'),
         daemonize=('d', False, 'daemonize process'),
         pid_file=('', '', 'name of file to write process ID to')):
    '''help message contained here'''
    pass

if __name__ == '__main__':
    main.command()
```

Options contents

Each option is defined by a keyword argument, whose name is the long option name (read *note*) and whose default value is a 3-tuple containing the elements:

1. short name
2. default value
3. help string

If a short name renders to False (for example, an empty string), then it's not used at all.

If the keyword argument name contains underscores, they are converted to dashes when generating the long option name since this is the typical convention for command line applications. If the keyword argument name ends with an underscore and is a python keyword, the trailing underscore will be removed.

Options processing

The default value for each option also determines how any values supplied for the option should be parsed:

- string: the string is passed as is
- integer: the value is convert to an integer
- boolean/None: `not default` is passed and option takes no value
- function: function is called with value and the return value is used
- list: the value is appended to this list
- tuple: value is checked to be present in default value (i.e. tuple behaves as a list of choices)
- dictionary: the value is then assumed to be in the format `key=value` and is then assigned to this dictionary, *example*

Note that only the boolean/None case results in an option that does not require an argument.

Usage

Running the python script above will trigger Opster's command line parsing facility, using arguments from `sys.argv`. `sys.argv[0]` will be prepended to the usage string (you can put it in another place using macro `%name` in the usage string).

In the case above simply running the script with no arguments will display help, since the script has a required positional argument:

```
> python example.py
main: invalid arguments

example.py [-l HOST] DIR

help message contained here

options:

 -l --listen     ip to listen on (default: localhost)
 -p --port       port to listen on (default: 8000)
 -d --daemonize  daemonize process
    --pid-file   name of file to write process ID to
 -h --help       display help
```

In general, to obtain help you can run the script with `python example.py --help` or `python example.py -h` (unless `-h` is used as the short name for another option).

You can parse command line strings programmatically, supplying a list of arguments to `.command()` attribute of the decorated main function:

```
main.command('-l 0.0.0.0 /my/dir'.split())
```

Or you still can use your function in python:

```
main('/tmp', listen='0.0.0.0')
```

In this case no type conversion (which is done during argument parsing) will be performed.

1.2.2 Subcommands

It's pretty common for a complex application to have a system of subcommands, and Opster provides a facility for handling them. It's easy to define them:

```
from opster import command, dispatch

@command(usage='[-t]', shortlist=True)
def simple(test=('t', False, 'just test execution')):
    '''
    Just a simple command to print keys of received arguments.

    I assure you! Nothing to see here. ;-)
    '''
    pass

@command(usage='[-p] [--exit value] ...', name='complex', hide=True)
def complex_(pass_=('p', False, "don't run the command"),
             exit=('', 100, 'exit with supplied code'),
             name=('n', '', 'optional name'),
             *args):
    '''This is a more complex command intended to do something'''
    pass

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

Your application will always also have the `help` command when it uses the subcommand system.

Usage

Usage is the same as with a single command, except that running without arguments will display the shortlist of commands:

```
> python multicommands.py
usage: multicommands.py <command> [options]

commands:

 simple  Just a simple command to print keys of received arguments.
```

Provided no commands have been marked with `shortlist=True`, all commands will be displayed (excluding those that have `hide=True`). Also, you can run `python multicommands.py help`, which will show the list of all commands (still excluding hidden commands).

Using `help command` or `command --help` will display a help on this command:

```
> python multicommands.py help simple
multicommands.py simple [-t]

Just a simple command to print keys of received arguments.
```

(continues on next page)

(continued from previous page)

```
    I assure you! Nothing to see here. ;-)

options:

 -t --test      just test execution
 -h --help      display help
```

Nested subcommands

It is possible to do sets of commands nested within each other, for example:

```
from opster import Dispatcher

d = Dispatcher()
nestedDispatcher = Dispatcher()

@d.command()
def info(host=('h', 'localhost', 'hostname'),
         port=('p', 8080, 'port')):
    '''Return some info'''
    print("INFO")


@nestedDispatcher.command(name='action')
def action(host=('h', 'localhost', 'hostname'),
           port=('p', 8080, 'port')):
    '''Make another action'''
    print("Action")

d.nest('nested', nestedDispatcher, 'some nested application commands')

if __name__ == "__main__":
    d.dispatch()
```

Usage with nested subcommands

Usage is the same as with a single dispatcher:

```
> python t.py nested --help
usage: t.py nested <command> [options]

commands:

action  Make another action
help    Show help for a given help topic or a help overview.
```

Global options

In case your application has options that every command should receive they can be declared in the following format:

```
options = [('v', 'verbose', False, 'enable additional output'),
           ('q', 'quiet', False, 'suppress output')]
```

Which is, obviously `(shortname, longname, default, help)`.

They can then be passed to `dispatch`:

```
if __name__ == '__main__':
    dispatch(globaloptions=options)
```

Global options must have a different `longname` from any options used in a subcommand. If a subcommand has an option with the same `shortname` as a global option, then the `shortname` will be used for the subcommand option (overriding the option in `globaloptions`).

Global options can be used before the argument that names the subcommand:

```
> python multicommands.py --quiet complex
write
warn
[100]
```

This is useful since it enables a user to alias a script with something like:

```
alias multi='python multicommands.py --quiet'
```

so that a global option is always enabled. However, non-global options may not appear before the subcommand argument:

```
> python multicommands.py --name=dave complex
error: option --name not recognized

usage: multicommands.py <command> [options]

commands:

 help     Show help for a given help topic or a help overview.
 nodoc    (no help text available)
 simple   Just simple command to print keys of received arguments.
```

Inner structure

@command and *@dispatch* are actually aliases for internal *Dispatcher* class. They assign and dispatch on a global object `opster._dispatcher`.

1.2.3 Partial names

Nice property of opster is that there is no need to type any option or subcommand name completely. You are always free to use only first few letter of name so opster can identify what are you trying to run.

For example, if we will use application created earlier, it's possible to call it like this:

```
app comp --ex 5
```

This means we're calling `complex_`, passing 5 as an argument for option `exit`.

1.2.4 Help generation

Help is generated automatically and is available by the `-h/--help` command line option or by `help` subcommand (if you're using subcommand system).

It is generated from the usage string, the function docstring and the help strings provided for each option and wrapped to length of 70 characters so it looks like:

```
> python multicommands.py help complex
multicommands.py complex: [-p] [--exit value] ...

This is a more complex command intended to do something

options:

 -p --pass  don't run the command
    --exit  exit with supplied code (default: 100)
 -n --name  optional name
 -h --help  show help
```

The default value is displayed here only if it does not evaluate as `False`.

If you need to display help from inside your application, you can always use the fact that the help-displaying function is attached to your decorated function object, i.e.:

```
@command()
def something():
    if some_consequences:
        something.help()
```

See an example from the tests.

1.2.5 Error messages

Opster provides a mechanism to quit out of script execution returning a message to the user: simply raise `command.Error` at any point. Opster will catch the error and display its message to the script user. For example:

```
from opster import command

@command()
def main(algorithm=('a', 'fast', 'algorithm: slow or fast')):
    '''
    script that uses two possible algorithms.
    '''
    if algorithm not in ('short', 'fast'):
        raise command.Error('unrecognised algorithm "{0}"'.format(algorithm))
    pass

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

Now we can do:

```
> python quit.py --algorithm=quick
unrecognised algorithm "quick"
```

1.3 API

1.3.1 Default API

`opster.`**`command`**(*options=None*, *usage=None*, *name=None*, *shortlist=False*, *hide=False*, *aliases=()*)

Decorator to mark function to be used as command for CLI.

Usage:

```
from opster import command, dispatch

@command()
def run(argument,
        optionalargument=None,
        option=('o', 'default', 'help for option'),
        no_short_name=('', False, 'help for this option')):
    print argument, optionalargument, option, no_short_name

if __name__ == '__main__':
    run.command()

# or, if you want to have multiple subcommands:
if __name__ == '__main__':
    dispatch()
```

Optional arguments:

- `options`: options in format described later. If not supplied, will be determined from function.
- `usage`: usage string for function, replaces `%name` with name of program or subcommand. In case if it's subcommand and `%name` is not present, usage is prepended by `name`
- `name`: used for multiple subcommands. Defaults to wrapped function name
- `shortlist`: if command should be included in shortlist. Used only with multiple subcommands
- `hide`: if command should be hidden from help listing. Used only with multiple subcommands, overrides `shortlist`
- `aliases`: list of aliases for command

If defined, options should be a list of 4-tuples in format:

```
(shortname, longname, default, help)
```

Where:

- `shortname` is a single letter which can be used then as an option specifier on command line (like `-a`). Will be not used if contains falsy value (empty string, for example)
- `longname` - main identificator of an option, can be used as on a command line with double dashes (like `--longname`)
- `default` value for an option, type of it determines how option will be processed
- `help` string displayed as a help for an option when asked to

`opster.`**`dispatch`**(*args=None*, *cmdtable=None*, *globaloptions=None*, *middleware=None*, *scriptname=None*)

Dispatch command line arguments using subcommands.

- `args`: list of arguments, default: `sys.argv[1:]`

1.3.2 If you need more

class `opster.`**`Dispatcher`**(*cmdtable=None*, *globaloptions=None*, *middleware=None*)

Central object for command dispatching system.

- `cmdtable`: dict of commands. Will be populated with functions, decorated with `Dispatcher.command`.
- `globaloptions`: list of options which are applied to all commands, will contain `--help` option at least.
- `middleware`: global decorator for all commands.

`command`(*options=None*, *usage=None*, *name=None*, *shortlist=False*, *hide=False*, *aliases=()*)

Decorator to mark function to be used as command for CLI.

Usage:

```
from opster import command, dispatch

@command()
def run(argument,
        optionalargument=None,
        option=('o', 'default', 'help for option'),
        no_short_name=('', False, 'help for this option')):
    print argument, optionalargument, option, no_short_name

if __name__ == '__main__':
    run.command()

# or, if you want to have multiple subcommands:
if __name__ == '__main__':
    dispatch()
```

Optional arguments:

- `options`: options in format described later. If not supplied, will be determined from function.
- `usage`: usage string for function, replaces `%name` with name of program or subcommand. In case if it's subcommand and `%name` is not present, usage is prepended by `name`
- `name`: used for multiple subcommands. Defaults to wrapped function name
- `shortlist`: if command should be included in shortlist. Used only with multiple subcommands
- `hide`: if command should be hidden from help listing. Used only with multiple subcommands, overrides `shortlist`
- `aliases`: list of aliases for command

If defined, options should be a list of 4-tuples in format:

```
(shortname, longname, default, help)
```

Where:

- `shortname` is a single letter which can be used then as an option specifier on command line (like `-a`). Will be not used if contains falsy value (empty string, for example)

- `longname` - main identificator of an option, can be used as on a command line with double dashes (like `--longname`)
- `default` value for an option, type of it determines how option will be processed
- `help` string displayed as a help for an option when asked to

dispatch(*args=None*, *scriptname=None*)
Dispatch command line arguments using subcommands.

- `args`: list of arguments, default: `sys.argv[1:]`

1.4 Opster tests

This is a test suite for opster library. Just read it to get some idea of how it works.

1.4.1 Actors cast

Choose python version:

```
$   if [ -z "$PYTHON" ]; then
>      PYTHON="python"
>   fi
```

Add opster to the PYTHONPATH:

```
$   if [ -z "$OPSTER_DIR" ]; then
>      OPSTER_DIR="$TESTDIR/.."
>   fi
>   export PYTHONPATH="$OPSTER_DIR"
```

Define function to make it simpler:

```
$ run() {
>   name=$1
>   shift
>   export COVERAGE_FILE=$TESTDIR/coverage.db
>   if [ -z "$COVERAGE" ]; then
>      "$PYTHON" "$TESTDIR/$name" "$@"
>   else
>      coverage run -a --rcfile="$TESTDIR/../.coveragerc" "$TESTDIR/$name" "$@"
>   fi
> }
```

Main characters:

- multicommands.py
- test_opts.py

1.4.2 Action

Check if usage is working:

```
$ run multicommands.py
usage: multicommands.py <command> [options]

commands:

 nodoc    (no help text available)
 simple   Just simple command to print keys of received arguments.
```

Ok, then let's run it:

```
$ run multicommands.py simple
['test', 'ui']
```

Yeah, nice one, but we know that command `complex` is just hidden there. Let's check it out:

```
$ run multicommands.py help complex
multicommands.py complex [-p] [--exit value] ...

That's more complex command intended to do something

    \xd0\x98 \xd1\x81\xd0\xb0\xd0\xbc\xd0\xbe\xd0\xb5␣
↪\xd0\xb3\xd0\xbb\xd0\xb0\xd0\xb2\xd0\xbd\xd0\xbe\xd0\xb5 - \xd0\xbc\xd1\x8b␣
↪\xd1\x82\xd1\x83\xd1\x82␣
↪\xd0\xbd\xd0\xb5\xd0\xbc\xd0\xbd\xd0\xbe\xd0\xb6\xd0\xb5\xd1\x87\xd0\xba\xd0\xbe␣
↪\xd1\x82\xd0\xb5\xd0\xba\xd1\x81\xd1\x82\xd0\xb0 \xd0\xbd\xd0\xb5 \xd0\xb2 ascii␣
↪\xd0\xbd\xd0\xb0\xd0\xbf\xd0\xb8\xd1\x88\xd0\xb5\xd0\xbc (esc)
    \xd0\xb8 \xd0\xbf\xd0\xbe\xd1\x81\xd0\xbc\xd0\xbe\xd1\x82\xd1\x80\xd0\xb8\xd0\xbc,
↪ \xd1\x87\xd1\x82\xd0\xbe \xd0\xb1\xd1\x83\xd0\xb4\xd0\xb5\xd1\x82. :) (esc)

options:

 -p --pass      don't run the command
    --exit      exit with supplied code (default: 100)
 -n --name      optional name
 -v --verbose   enable additional output
 -q --quiet     suppress output
 -h --help      display help
```

Check if integer errors correctly:

```
$ run multicommands.py complex --exit q
error: invalid option value 'q' for option 'exit'

multicommands.py complex [-p] [--exit value] ...

That's more complex command intended to do something

    \xd0\x98 \xd1\x81\xd0\xb0\xd0\xbc\xd0\xbe\xd0\xb5␣
↪\xd0\xb3\xd0\xbb\xd0\xb0\xd0\xb2\xd0\xbd\xd0\xbe\xd0\xb5 - \xd0\xbc\xd1\x8b␣
↪\xd1\x82\xd1\x83\xd1\x82␣
↪\xd0\xbd\xd0\xb5\xd0\xbc\xd0\xbd\xd0\xbe\xd0\xb6\xd0\xb5\xd1\x87\xd0\xba\xd0\xbe␣
↪\xd1\x82\xd0\xb5\xd0\xba\xd1\x81\xd1\x82\xd0\xb0 \xd0\xbd\xd0\xb5 \xd0\xb2 ascii␣
↪\xd0\xbd\xd0\xb0\xd0\xbf\xd0\xb8\xd1\x88\xd0\xb5\xd0\xbc (esc)
    \xd0\xb8 \xd0\xbf\xd0\xbe\xd1\x81\xd0\xbc\xd0\xbe\xd1\x82\xd1\x80\xd0\xb8\xd0\xbc,
↪ \xd1\x87\xd1\x82\xd0\xbe \xd0\xb1\xd1\x83\xd0\xb4\xd0\xb5\xd1\x82. :) (esc)

options:
```

(continues on next page)

(continued from previous page)

```
 -p --pass     don't run the command
    --exit     exit with supplied code (default: 100)
 -n --name     optional name
 -v --verbose  enable additional output
 -q --quiet    suppress output
 -h --help     display help
```

Opster can parse non-global options after the command argument:

```
$ run multicommands.py complex --name dave
write
info
warn
[100]
```

We also get the right command when using –opt=value syntax:

```
$ run multicommands.py complex --name=dave
write
info
warn
[100]
```

Global options can appear before the command argument.

> $ run multicommands.py –quiet complex write warn [100]

However, non-global options before the command argument are not allowed:

```
$ run multicommands.py --name=dave complex
error: option --name not recognized

usage: multicommands.py <command> [options]

commands:

 help    Show help for a given help topic or a help overview.
 nodoc   (no help text available)
 simple  Just simple command to print keys of received arguments.
```

regardless of which syntax you use:

```
$ run multicommands.py --name dave complex
error: option --name not recognized

usage: multicommands.py <command> [options]

commands:

 help    Show help for a given help topic or a help overview.
 nodoc   (no help text available)
 simple  Just simple command to print keys of received arguments.
```

Opster won't accidentally run the simple command because you tried to pass simple as an argument to the –name option:

```
$ run multicommands.py --name simple complex
error: option --name not recognized

usage: multicommands.py <command> [options]

commands:

 help    Show help for a given help topic or a help overview.
 nodoc   (no help text available)
 simple  Just simple command to print keys of received arguments.
```

We also have completion:

```
$ run multicommands.py _completion
# opster bash completion start
_opster_completion()
{
    COMPREPLY=( $( COMP_WORDS="${COMP_WORDS[*]}" \
                   COMP_CWORD=$COMP_CWORD \
                   OPSTER_AUTO_COMPLETE=1 $1 ) )
}
complete -o default -F _opster_completion multicommands.py
# opster bash completion end
```

Now we're going to test if a script with a single command will work (not everyone needs subcommands, you know):

```
$ run test_opts.py
test_opts.py: invalid arguments

test_opts.py [-l HOST] DIR

Command with option declaration as keyword arguments

options:

 -l --listen       ip to listen on (default: localhost)
 -p --port         port to listen on (default: 8000)
 -d --daemonize    daemonize process
    --pid-file     name of file to write process ID to
 -D --definitions  just some definitions
 -t --test         testing help for a function (default: test)
 -h --help         display help
```

Yeah, I've got it, I should supply some arguments:

```
$ run test_opts.py -d -p 5656 --listen anywhere right-here
{'daemonize': True,
 'definitions': {},
 'dirname': 'right-here',
 'listen': 'anywhere',
 'pid_file': '',
 'port': 5656,
 'test': 'test'}
```

Now let's test our definitions:

```
$ run test_opts.py -D a=b so-what?
{'daemonize': False,
 'definitions': {'a': 'b'},
 'dirname': 'so-what?',
 'listen': 'localhost',
 'pid_file': '',
 'port': 8000,
 'test': 'test'}
```

As long as only the last option has a parameter We can combine short options into one argument:

```
$ run test_opts.py -dDa=b so-what?
{'daemonize': True,
 'definitions': {'a': 'b'},
 'dirname': 'so-what?',
 'listen': 'localhost',
 'pid_file': '',
 'port': 8000,
 'test': 'test'}
```

The parameter can be in a separate argument:

```
$ run test_opts.py -dD a=b so-what?
{'daemonize': True,
 'definitions': {'a': 'b'},
 'dirname': 'so-what?',
 'listen': 'localhost',
 'pid_file': '',
 'port': 8000,
 'test': 'test'}

$ run test_opts.py -D can-i-haz fail?
error: wrong definition: 'can-i-haz' (should be in format KEY=VALUE)

test_opts.py [-l HOST] DIR

Command with option declaration as keyword arguments

options:

 -l --listen       ip to listen on (default: localhost)
 -p --port         port to listen on (default: 8000)
 -d --daemonize    daemonize process
    --pid-file     name of file to write process ID to
 -D --definitions  just some definitions
 -t --test         testing help for a function (default: test)
 -h --help         display help
```

Should we check passing some invalid arguments? I think so:

```
$ run test_opts.py --wrong-option
error: option --wrong-option not recognized

test_opts.py [-l HOST] DIR

Command with option declaration as keyword arguments
```

(continues on next page)

(continued from previous page)

```
options:

 -l --listen       ip to listen on (default: localhost)
 -p --port         port to listen on (default: 8000)
 -d --daemonize    daemonize process
    --pid-file     name of file to write process ID to
 -D --definitions  just some definitions
 -t --test         testing help for a function (default: test)
 -h --help         display help
```

Lets try some extended option types:

```
$ run test_extopts.py -h
test_extopts.py [OPTIONS]

Command using extended option types

options:

 -m --money  amount of money (default: 100.00)
 -r --ratio  input/output ratio (default: 1/4)
 -h --help   display help

$ run test_extopts.py
money: <class 'decimal.Decimal'> 100.00
ratio: <class 'fractions.Fraction'> 1/4

$ run test_extopts.py --money=-.12 --ratio='5/6'
money: <class 'decimal.Decimal'> -0.12
ratio: <class 'fractions.Fraction'> 5/6
```

Another things should be checked: calling help display from the function itself:

```
$ run selfhelp.py --assist
selfhelp [OPTIONS]

Displays ability to show help

options:

    --assist  show help
 -h --help    display help
```

Are we getting nicely stripped body when not following subject/body convention of writing commands?

```
$ run hello.py --help
hello.py [OPTIONS] NAME [TIMES]

Hello world continues the long established tradition
of delivering simple, but working programs in all
kinds of programming languages.

This tests different docstring formatting (just text instead of having
subject and body).

options:
```

(continues on next page)

(continued from previous page)

```
 -g --greeting  Greeting to use (default: Hello)
 -h --help      display help
```

There is no problems with having required and optional arguments at the same time:

```
$ run hello.py stranger
Hello stranger
$ run hello.py stranger 2 -g 'Good bye'
Good bye stranger
Good bye stranger
$ run hello.py stranger -g 'Good bye' 2
Good bye stranger
Good bye stranger
```

And there is no problems handling unicode data:

```
$ OPSTER_ARG_ENCODING=utf-8 run hello.py  -g
\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82␣
↪\xd0\xba\xd1\x80\xd0\xbe\xd1\x81\xd0\xb0\xd0\xb2\xd1\x87\xd0\xb5\xd0\xb3 (esc)
```

There is no problems with `TypeError: function() got multiple values for keyword argument 'option'`:

```
$ run multivalueserr.py some arguments hehe
I work! False bopt some arguments ('hehe',)
```

Opster can give helpful error messages if arguments are invalid:

```
$ run quit.py --help
quit.py [OPTIONS]

script that uses different algorithms and numbers of cpus

options:

 -a --algo1  algorithm: slow or fast (default: fast)
 -A --algo2  algorithm: slow or fast (default: slow)
 -n --ncpus  number of cpus to use (default: 1)
 -h --help   display help
```

Scripts can exit with a user-defined error message at any time by raising `command.Error`:

```
$ run quit.py --algo1=quick
unrecognised algorithm "quick"
```

Or arguments can be rejected because they are not in a tuple:

```
$ run quit.py --algo2=quick
error: unrecognised value: 'quick' (should be one of slow, fast)

quit.py [OPTIONS]

script that uses different algorithms and numbers of cpus

options:

```

(continues on next page)

(continued from previous page)

```
 -a --algo1  algorithm: slow or fast (default: fast)
 -A --algo2  algorithm: slow or fast (default: slow)
 -n --ncpus  number of cpus to use (default: 1)
 -h --help   display help

$ run quit.py --ncpus=-1
error: unrecognised value: '-1' (should be one of 1, 2, 3, 4)

quit.py [OPTIONS]

script that uses different algorithms and numbers of cpus

options:

 -a --algo1  algorithm: slow or fast (default: fast)
 -A --algo2  algorithm: slow or fast (default: slow)
 -n --ncpus  number of cpus to use (default: 1)
 -h --help   display help
```

Just check that the tuple options work when there's no error:

```
$ run quit.py --algo1=fast --algo2=slow --ncpus=3
algo1: fast
algo2: slow
ncpus: 3
```

That's all for today; see you next time!

There is no problems with handling variable argumentrs and underscores:

```
$ run varargs.py --test-option test1 var1 var2
{'args': ('var1', 'var2'), 'test_option': 'test1'}
$ run varargs.py var1 var2
{'args': ('var1', 'var2'), 'test_option': 'test'}
```

We should check that we can still run opster scripts written using the old API:

```
$ run oldapi.py help
usage: oldapi.py <command> [options]

commands:

 cmd1   (no help text available)
 cmd2   (no help text available)
 help   Show help for a given help topic or a help overview.
$ run oldapi.py cmd1
Not being quiet!
$ run oldapi.py cmd2 --verbose 1 2 3
1
2
3
```

We can have an option that uses the '-h' short name (although we use it as a short name for '–help':

```
$ run ls.py --help
ls [-h]

```

(continues on next page)

(continued from previous page)

```
(no help text available)

options:

 -h --human  Pretty print filesizes
    --nohelp1
 -n --nohelp2
    --help   display help
```

Let's just check that the scriptname argument `scriptname='ls'` works as expected for the error messages:

```
$ run ls.py invalid
ls: invalid arguments

ls [-h]

(no help text available)

options:

 -h --human  Pretty print filesizes
    --nohelp1
 -n --nohelp2
    --help   display help

$ run ls.py --invalid
error: option --invalid not recognized

ls [-h]

(no help text available)

options:

 -h --human  Pretty print filesizes
    --nohelp1
 -n --nohelp2
    --help   display help
```

We can also supply the `scriptname` argument to `dispatch`:

```
$ run scriptname.py
usage: newname <command> [options]

commands:

 cmd   (no help text available)
 help  Show help for a given help topic or a help overview.

$ run scriptname.py help
usage: newname <command> [options]

commands:

 cmd   (no help text available)
 help  Show help for a given help topic or a help overview.
```

(continues on next page)

(continued from previous page)

```
$ run scriptname.py help cmd
newname cmd [-h]

(no help text available)

options:

 -h --help  display help

$ run scriptname.py cmd --help
newname cmd [-h]

(no help text available)

options:

 -h --help  display help

$ run scriptname.py cmd invalid
cmd: invalid arguments

newname cmd [-h]

(no help text available)

options:

 -h --help  display help

$ run scriptname.py cmd --invalid
error: option --invalid not recognized

newname cmd [-h]

(no help text available)

options:

 -h --help  display help
```

It is possible to nest a dispatcher as a command within another dispatcher so that we can have subsubcommands.

```
$ run subcmds.py
usage: subcmds.py <command> [options]

commands:

 cmd   Help for cmd
 cmd2  (no help text available)
 help  Show help for a given help topic or a help overview.

$ run subcmds.py help cmd
usage: subcmds.py cmd <command> [options]

commands:

 subcmd1  Help for subcmd1
```

(continues on next page)

(continued from previous page)

```
 subcmd2  Help for subcmd2
 subcmd3  Help for subcmd3

$ run subcmds.py cmd --help
usage: subcmds.py cmd <command> [options]

commands:

 help     Show help for a given help topic or a help overview.
 subcmd1  Help for subcmd1
 subcmd2  Help for subcmd2
 subcmd3  Help for subcmd3

$ run subcmds.py cmd2 --help
subcmds.py cmd2 [OPTIONS]

(no help text available)

options:

 -h --showhelp  Print the help message
    --help      display help

$ run subcmds.py cmd2 --showhelp
Showing the help:
subcmds.py cmd2 [OPTIONS]

(no help text available)

options:

 -h --showhelp  Print the help message
    --help      display help

$ run subcmds.py cmd subcmd1 --help
subcmds.py cmd subcmd1 [OPTIONS]

Help for subcmd1

options:

 -q --quiet     quietly
 -h --showhelp  Print the help message
    --help      display help

$ run subcmds.py help cmd subcmd1
subcmds.py cmd subcmd1 [OPTIONS]

Help for subcmd1

options:

 -q --quiet     quietly
 -h --showhelp  Print the help message
    --help      display help

$ run subcmds.py cmd subcmd1
```

(continues on next page)

(continued from previous page)

```
running subcmd1

$ run subcmds.py cmd subcmd1 --quiet

$ run subcmds.py cmd subcmd1 --showhelp
running subcmd1
Showing the help:
subcmds.py cmd subcmd1 [OPTIONS]

Help for subcmd1

options:

 -q --quiet     quietly
 -h --showhelp  Print the help message
    --help      display help

$ run subcmds.py cmd subcmd2
subcmd2: invalid arguments

subcmds.py cmd subcmd2 NUMBER

Help for subcmd2

options:

 -h --help  display help

$ run subcmds.py cmd subcmd2 5
running subcmd2 5

$ run subcmds.py help cmd subcmd3
usage: subcmds.py cmd subcmd3 <command> [options]

commands:

 subsubcmd  Help for subsubcmd

$ run subcmds.py help cmd subcmd3 --help
subcmds.py help [TOPIC]

Show help for a given help topic or a help overview.

        With no arguments, print a list of commands with short help messages.

        Given a command name, print help for that command.

options:

 -h --help  display help

$ run subcmds.py help cmd subcmd3 subsubcmd
subcmds.py cmd subcmd3 subsubcmd [OPTIONS]

Help for subsubcmd

options:
```

(continues on next page)

(continued from previous page)

```
 -l --loud      loudly
 -h --showhelp  Print the help message
    --help      display help

$ run subcmds.py cmd subcmd3 subsubcmd --help
subcmds.py cmd subcmd3 subsubcmd [OPTIONS]

Help for subsubcmd

options:

 -l --loud      loudly
 -h --showhelp  Print the help message
    --help      display help

$ run subcmds.py cmd subcmd3 subsubcmd --showhelp
Showing the help:
subcmds.py cmd subcmd3 subsubcmd [OPTIONS]

Help for subsubcmd

options:

 -l --loud      loudly
 -h --showhelp  Print the help message
    --help      display help

$ run subcmds.py cmd subcmd3 subsubcmd

$ run subcmds.py cmd subcmd3 subsubcmd -l
running subsubcmd
```

Check the *varargs* works when calling `main` directly:

```
$ run varargs_py2.py

main():
TypeError raised

main("a"):
shop: a
cheeses: ()
music: False

main("a", "b"):
shop: a
cheeses: ('b',)
music: False

main("a", "b", "c"):
shop: a
cheeses: ('b', 'c')
music: False

main(music=True):
TypeError raised
```

(continues on next page)

(continued from previous page)

```

main("a", music=True):
shop: a
cheeses: ()
music: True

main("a", "b", music=True):
shop: a
cheeses: ('b',)
music: True

main("a", "b", "c", music=True):
shop: a
cheeses: ('b', 'c')
music: True
```

Check that calling main directly still works even if `` `arginfo.defaults` `` is None:

```
$ run nodefaults.py
hello
```

CHAPTER 2

Features

- parsing of arguments from `sys.argv` or custom strings
- *conversion* from strings to the appropriate Python objects
- *help message* generation
- positional and named arguments (i.e. arguments and options)
- *subcommands* (and subcommands of subcommands) support
- short, clean and concise definitions
- *ability to shorten* names of both subcommands and long options

CHAPTER 3

Quick example

Here's an example of a program that defines a single option:

```python
import sys
from opster import command

@command()
def main(message,
         no_newline=('n', False, "don't print a newline")):
    '''Simple echo program'''
    sys.stdout.write(message)
    if not no_newline:
        sys.stdout.write('\n')

if __name__ == '__main__':
    main.command()
```

Running the program above will print the help message::

```
> ./echo.py
echo.py: invalid arguments
echo.py [OPTIONS] MESSAGE

Simple echo program

options:

 -n --no-newline  don't print a newline
 -h --help        display help
```

As you can see, here we have defined an option to not print newlines: the keyword argument is used as the long name for the option and its default value is a 3-tuple, containing short name for an option (can be empty), default value (whose type determines what conversion is applied - *see description*) and a help string for the option.

Any underscores in the keyword argument name are converted into dashes in the long option name.

When a command is called using the long name for an option, the option need not be fully entered. In the case above this could look like `./echo.py --no-new`. This is also true for subcommands: read about them and everything else you'd like to know further on in the documentation.

CHAPTER 4

Nice points

- Opster is a single file, which means that you can easily include it in your application
- When you've decorated a function with `command`, you can continue to use it as an ordinary Python function.
- It's easy to switch between a simple command line application and one that uses subcommands.
- There's no need to type the complete name of an option or subcommand: *just type* as many letters as are needed to distinguish it from the others.

Read more in *Overview*.

Python Module Index

Index