mando Documentation

Release 0.2

Michele Lacchia

Sep 27, 2017

Contents

mando is a wrapper around `argparse`, allowing you to write complete CLI applications in seconds while maintaining all the flexibility.

CHAPTER 1

The problem

`argparse` is great for single-command applications, which only have some options and one, default command. Unfortunately, when more commands are added, the code grows too much along with its complexity.

CHAPTER 2

The solution

mando makes an attempt to simplify this. Since commands are nothing but functions, mando simply provides a couple of decorators and the job is done. mando tries to infer as much as possible, in order to allow you to write just the code that is strictly necessary.

This example should showcase most of mando's features:

```
# gnu.py
from mando import main, command, arg


@arg('maxdepth', metavar='<levels>')
def find(path, pattern, maxdepth=None, P=False, D=None):
    '''Mock some features of the GNU find command.

    This is not at all a complete program, but a simple representation to
    showcase mando's coolest features.

    :param path: The starting path.
    :param pattern: The pattern to look for.
    :param -d, --maxdepth <int>: Descend at most <levels>.
    :param -P: Do not follow symlinks.
    :param -D <debug-opt>: Debug option, print diagnostic information.'''

    if maxdepth is not None and maxdepth < 2:
        print('If you choose maxdepth, at least set it > 1')
    if P:
        print('Following symlinks...')
    print('Debug options: {0}'.format(D))
    print('Starting search with pattern: {0}'.format(pattern))
    print('No file found!')


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

mando extracts information from your command's docstring. So you can document your code and create the CLI application at once! In the above example the Sphinx format is used, but mando does not force you to write ReST docstrings. Currently, it supports the following styles:

- Sphinx (the default one)
- Google
- Numpy

To see how to specify the docstring format, see *Other Docstring Formats*.

The first paragraph is taken to generate the command's *help*. The remaining part (after removing all `:param:`'s) is the *description*. For everything that does not fit in the docstring, mando provides the `@arg` decorator, to override arbitrary arguments before they get passed to `argparse`.

```
$ python gnu.py -h
usage: gnu.py [-h] {find} ...

positional arguments:
  {find}
    find      Mock some features of the GNU find command.

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

$ python gnu.py find -h
usage: gnu.py find [-h] [-d <levels>] [-P] [-D <debug-opt>] path pattern

This is not at all a complete program, but a simple representation to showcase
mando's coolest features.

positional arguments:
  path                  The starting path.
  pattern               The pattern to look for.

optional arguments:
  -h, --help            show this help message and exit
  -d <levels>, --maxdepth <levels>
                        Descend at most <levels>.
  -P                    Do not follow symlinks.
  -D <debug-opt>        Debug option, print diagnostic information.
```

As you can see the short options and metavars have been passed to argparse. Now let's check the program itself:

```
$ python gnu.py find . "*.py"
Debug options: None
Starting search with pattern: *.py
No file found!
$ python gnu.py find . "*.py" -P
Following symlinks...
Debug options: None
Starting search with pattern: *.py
No file found!
$ python gnu.py find . "*" -P -D dbg
Following symlinks...
Debug options: dbg
Starting search with pattern: *
No file found!
$ python gnu.py find . "*" -P -D "dbg,follow,trace"
```

```
Following symlinks...
Debug options: dbg,follow,trace
Starting search with pattern: *
No file found!

$ python gnu.py find -d 1 . "*.pyc"
If you choose maxdepth, at least set it > 1
Debug options: None
Starting search with pattern: *.pyc
No file found!
$ python gnu.py find --maxdepth 0 . "*.pyc"
If you choose maxdepth, at least set it > 1
Debug options: None
Starting search with pattern: *.pyc
No file found!
$ python gnu.py find --maxdepth 4 . "*.pyc"
Debug options: None
Starting search with pattern: *.pyc
No file found!

$ python gnu.py find --maxdepth 4 .
usage: gnu.py find [-h] [-d <levels>] [-P] [-D <debug-opt>] path pattern
gnu.py find: error: too few arguments
```

CHAPTER 3

Contents

Usage

Defining commands

A command is a function decorated with `@command`. mando tries to extract as much as information as possible from the function's docstring and its signature.

The paragraph of the docstring is the command's **help**. For optimal results it shouldn't be longer than one line. The second paragraph contains the command's **description**, which can be as long as needed. If only one paragraph is present, it is used for both the help and the description. You can document the parameters with the common Sphinx's `:param:` syntax.

For example, this program generates the following helps:

```
from mando import command, main


@command
def cmd(foo, bar):
    '''Here stands the help.

    And here the description of this useless command.

    :param foo: Well, the first arg.
    :param bar: Obviously the second arg. Nonsense.'''

    print(arg, bar)


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

```
$ python command.py -h
usage: command.py [-h] {cmd} ...

positional arguments:
  {cmd}
    cmd       Here stands the help.

optional arguments:
  -h, --help  show this help message and exit
$ python command.py cmd -h
usage: command.py cmd [-h] foo bar

And here the description of this useless command.

positional arguments:
  foo         Well, the first arg.
  bar         Obviously the second arg. Nonsense.

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

Long and short options (flags)

You can specify short options in the docstring as well, with the `:param:` syntax. The recognized formats are these:

- `:param -O: Option help`
- `:param --option:  Option help`
- `:param -o, --output:  Option help`

Example:

```
from mando import command, main


@command
def ex(foo, b=None, spam=None):
    '''Nothing interesting.

    :param foo: Bla bla.
    :param -b: A little flag.
    :param -s, --spam: Spam spam spam spam.'''

    print(foo, b, spam)

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

Usage:

```
$ python short_options.py ex -h
usage: short_options.py ex [-h] [-b B] [-s SPAM] foo

Nothing interesting.

positional arguments:
  foo                   Bla bla.
```

```
optional arguments:
  -h, --help            show this help message and exit
  -b B                  A little flag.
  -s SPAM, --spam SPAM  Spam spam spam spam.
$ python short_options.py ex 2
('2', None, None)
$ python short_options.py ex 2 -b 8
('2', '8', None)
$ python short_options.py ex 2 -b 8 -s 9
('2', '8', '9')
$ python short_options.py ex 2 -b 8 --spam 9
('2', '8', '9')
```

How default arguments are handled

If an argument has a default, then mando takes it as an optional argument, while those which do not have a default are interpreted as positional arguments. Here are the actions taken by mando when a default argument is encountered:

Default argument type	What mando specifies in `add_argument()`
bool	*action* `store_true` or `store_false` is added
list	*action* `append` is added.
int	*type* `int()` is added.
float	*type* `float()` is added.
str	*type* `str()` is added.

So, for example, if a default argument is an integer, mando will automatically convert command line arguments to `int()`:

```
from mando import command, main


@command
def po(a=2, b=3):
    print(a ** b)


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

```
$ python default_args.py po -h
usage: default_args.py po [-h] [-a A] [-b B]

optional arguments:
  -h, --help  show this help message and exit
  -a A
  -b B
$ python default_args.py po -a 4 -b 9
262144
```

Note that passing the arguments positionally does not work, because `argparse` expects optional args and `a` and `b` are already filled with defaults:

```
$ python default_args.py po
8
$ python default_args.py po 9 8
```

```
usage: default_args.py [-h] {po} ...
default_args.py: error: unrecognized arguments: 9 8
```

To overcome this, mando allows you to specify positional arguments' types in the docstring, as explained in the next section.

Adding *type* and *metavar* in the docstring

This is especially useful for positional arguments, but it is usually used for all type of arguments. The notation is this: `:param {opt-name} <type>: Help.` `<type>` must be a built-in type among the following:

- `<i>`, `<int>`, `<integer>` to cast to `int()`;
- also `<n>`, `<num>`, `<number>` to cast to `int()`;
- `<s>`, `<str>`, `<string>` to cast to `str()`;
- `<f>`, `<float>` to cast to `float()`.

mando also adds `<type>` as a metavar. Actual usage:

```
from mando import command, main


@command
def pow(a, b, mod=None):
    '''Mimic Python's pow() function.

    :param a <float>: The base.
    :param b <float>: The exponent.
    :param -m, --mod <int>: Modulus.'''

    if mod is not None:
        print((a ** b) % mod)
    else:
        print(a ** b)


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

```
$ python types.py pow -h
usage: types.py pow [-h] [-m <int>] a b

Mimic Python's pow() function.

positional arguments:
a                     The base.
b                     The exponent.

optional arguments:
-h, --help            show this help message and exit
-m <int>, --mod <int>
                    Modulus.
$ python types.py pow 5 8
390625.0
$ python types.py pow 4.5 8.3
264036.437449
```

```
$ python types.py pow 5 8 -m 8
1.0
```

Overriding arguments with `@arg`

You may need to specify some argument to argparse, and it is not possible to include in the docstring. mando provides the `@arg` decorator to accomplish this. Its signature is as follows: `@arg(arg_name, *args, **kwargs)`, where `arg_name` must be among the function's arguments, while the remaining arguments will be directly passed to `argparse.add_argument()`. Note that this decorator will override other arguments that mando inferred either from the defaults or from the docstring.

`@command` Arguments

There are three special arguments to the `@command()` decorator to allow for special processing for the decorated function. The first argument, also available as keyword `name='alias_name'` will allow for an alias of the command. The second argument, also available as keyword `doctype='rest'` allows for Numpy or Google formatted docstrings to be used. The third is only available as keyword `formatter_class='argparse_formatter_class'` to format the display of the docstring.

Aliasing Commands

A common use-case for this is represented by a function with underscores in it. Usually commands have dashes instead. So, you may specify the aliasing name to the `@command()` decorator, this way:

```
@command('very-powerful-cmd')
def very_powerful_cmd(arg, verbose=False):
    pass
```

And call it as follows:

```
$ python prog.py very-powerful-cmd 2 --verbose
```

Note that the original name will be discarded and won't be usable.

Other Docstring Formats

There are three commonly accepted formats for docstrings. The Sphinx docstring, and the mando dialect of Sphinx described in this documentation are treated equally and is the default documentation style named `rest` for REStructured Text. The other two available styles are `numpy` and `google`. This allows projects that use mando, but already have docstrings in these other formats not to have to convert the docstrings.

An example of using a Numpy formatted docstring in mando:

```
@command(doctype='numpy')
def simple_numpy_docstring(arg1, arg2="string"):
    '''One line summary.

    Extended description.

    Parameters
    ----------
    arg1 : int
```

```
        Description of `arg1`
    arg2 : str
        Description of `arg2`

    Returns
    -------
    str
        Description of return value.
    '''
    return int(arg1) * arg2
```

An example of using a Google formatted docstring in mando:

```
@program.command(doctype='google')
def simple_google_docstring(arg1, arg2="string"):
    '''One line summary.

    Extended description.

    Args:
      arg1(int): Description of `arg1`
      arg2(str): Description of `arg2`
    Returns:
      str: Description of return value.
    '''
    return int(arg1) * arg2
```

Formatter Class

For the help display there is the opportunity to use special formatters. Any argparse compatible formatter class can be used. There is an alternative formatter class available with mando that will display on ANSI terminals.

The ANSI formatter class has to be imported from mando and used as follows:

```
from mando.rst_text_formatter import RSTHelpFormatter

@command(formatter_class=RSTHelpFormatter)
def pow(a, b, mod=None):
    '''Mimic Python's pow() function.

    :param a <float>: The base.
    :param b <float>: The exponent.
    :param -m, --mod <int>: Modulus.'''

    if mod is not None:
        print((a ** b) % mod)
    else:
        print(a ** b)
```

Shell autocompletion

Mando supports autocompletion via the optional dependency `argcomplete`. If that package is installed, mando detects it automatically without the need to do anything else.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search