docopt Documentation

Release

Vladimir Keleshev

June 04, 2012

CONTENTS

Note: New in version 0.3.0: (sub)commands support, `[options]` shortcut.

Warning: Incompatible change in 0.3.0: `docopt` function returns a dict.

Isn't it awesome how `optparse` and `argparse` generate help and usage-messages based on your code?!

Hell no! You know what's awesome? It's when the option parser *is* generated based on the beautiful usage-message that you write in a docstring! This way you don't need to write this stupid repeatable parser-code, and instead can write only the usage-message—*the way you want it*.

`docopt` helps you create most beautiful command-line interfaces *easily*:

```
"""Naval Fate.

Usage:
  naval_fate.py ship new <name>...
  naval_fate.py ship [<name>] move <x> <y> [--speed=<kn>]
  naval_fate.py ship shoot <x> <y>
  naval_fate.py mine (set|remove) <x> <y> [--moored|--drifting]
  naval_fate.py -h | --help
  naval_fate.py --version

Options:
  -h --help     Show this screen.
  --version     Show version.
  --speed=<kn>  Speed in knots [default: 10].
  --moored      Mored (anchored) mine.
  --drifting    Drifting mine.

"""
from docopt import docopt


if __name__ == '__main__':
    arguments = docopt(__doc__, version='Naval Fate 2.0')
    print(arguments)
```

Beat that! The option parser is generated based on docstring above, that you pass to `docopt` function. `docopt` parses the usage-pattern (`"Usage: ..."`) and option-descriptions (lines starting with dash `-`) and ensures that program invocation matches the ussage-pattern; it parses options, arguments and commands based on that. The basic idea is that *a good usage-message has all necessary information in it to make a parser*.

Even pep257 recommends putting usage-message in the module docstrings.

CHAPTER

ONE

INSTALLATION

Use pip or easy_install:

```
pip install docopt
```

Alternatively you can just drop `docopt.py` file into your project—it is self-contained. Get source on github.

`docopt` is tested with Python 2.5, 2.6, 2.7, 3.1, 3.2.

CHAPTER

TWO

API

```
from docopt import docopt
```

docopt(*doc[, argv=sys.argv[1:]][, help=True][, version=None]*)

`docopt` takes 1 required and 3 optional arguments:

- `doc` should be a module docstring (`__doc__`) or some other string that describes **usage-message** in a human-readable format, that will be parsed to create the option parser. The simple rules of how to write such a docstring are given in next sections. Here is a quick example of such a string:

```
"""Usage: my_program.py [-hso FILE] [--quiet | --verbose] [INPUT ...]

-h --help    show this
-s --sorted  sorted output
-o FILE      specify output file [default: ./test.txt]
--quiet      print less text
--verbose    print more text

"""
```

- `argv` is optional argument vector; by default it is the argument vector passed to your program (`sys.argv[1:]`). You can supply it with list of strings (similar to `sys.argv`) e.g. `['--verbose', '-o', 'hai.txt']`.
- `help`, by default `True`, specifies whether the parser should automatically print the usage-message (supplied as `doc`) and terminate, in case `-h` or `--help` options are encountered. If you want to handle `-h` or `--help` options manually (as all other options), set `help=False`.
- `version`, by default `None`, is an optional argument that specifies the version of your program. If supplied, then, if parser encounters `--version` option, it will print the supplied version and terminate. `version` could be any printable object, but most likely a string, e.g. `"2.1.0rc1"`.

Note: when `docopt` is set to automatically handle `-h`, `--help` and `--version` options, you still need to mention them in `doc`. Also for your users to know about them.

The **return** value is just dictionary with options, arguments and commands, with keys spelled exactly like in usage-message (long versions of options are given priority). For example, if you invoke the top example as:

```
naval_fate.py ship Guardian move 100 150 --speed=15
```

the return dictionary will be:

```
{'--drifting': False,    'mine': False,
 '--help': False,        'move': True,
 '--moored': False,      'new': False,
 '--speed': '15',        'remove': False,
 '--version': False,     'set': False,
 '<name>': ['Guardian'], 'ship': True,
 '<x>': '100',           'shoot': False,
 '<y>': '150'}
```

This turns out to be the most straight-forward, unambiguous and readable format possible. You can instantly see that `args['<name>']` is an argument, `args['--speed']` is an options, and `args['move']` is a command.

CHAPTER

THREE

USAGE-MESSAGE FORMAT

Usage-message consists of 2 parts:

- Usage-pattern, e.g.:

```
Usage: my_program.py [-hso FILE] [--quiet | --verbose] [INPUT ...]
```

- Option-description, e.g.:

```
-h --help    show this
-s --sorted  sorted output
-o FILE      specify output file [default: ./test.txt]
--quiet      print less text
--verbose    print more text
```

Their format is described below; other text is ignored. Also, take a look at our beautiful examples.

3.1 Usage-pattern format

Usage-pattern is a substring of `doc` that starts with `usage:` (case-*in*sensitive) and ends with *visibly* empty line. Minimum example:

```
"""Usage: my_program.py

"""
```

The first word after `usage:` is interpreted as your program's name. You can specify your program's name several times to signify several exclusive patterns:

```
"""Usage: my_program.py FILE
          my_program.py COUNT FILE

"""
```

Each pattern can consist of following elements:

- **<arguments>**, **ARGUMENTS**. Arguments are specified as either upper-case words, e.g. `my_program.py CONTENT-PATH` or words surrounded by angular brackets: `my_program.py <content-path>`.
- **–options**. Options are words started with dash (-), e.g. `--output`, `-o`. You can "stack" several of one-letter options, e.g. `-oiv` which will be same as `-o -i -v`. Options can have arguments, e.g. `--input=FILE` or `-i FILE` or even `-iFILE`. However it is important that you specify all option-descriptions (see next section).
- **commands** are words that do *not* follow the described above conventions of `--options` or `<arguments>` or `ARGUMENTS`.

Use the following operators to specify patterns:

- **[ ]** (brackets) **optional** elements. e.g.: `my_program.py [-hvqo FILE]`
- **( )** (parens) **required** elements. All elements that are *not* put in **[ ]** are also required, e.g.: `my_program.py --path=<path> <file>...` is same as `my_program.py (--path=<path> <file>...)`. (Note, "required options" might be not a good idea for your users).
- **|** (pipe) **mutualy exclussive** elements. Group them using **( )** if one of the mutually exclussive elements is required: `my_program.py (--clockwise | --counter-clockwise) TIME`. Group them using **[ ]** if none of the mutually-exclusive elements are required: `my_program.py [--left | --right]`.
- **...** (ellipsis) **one or more** elements. To specify that arbitrary number of repeating elements could be accepted use ellipsis (`...`), e.g. `my_program.py FILE ...` means one or more `FILE`-s are accepted. If you want to accept zero or more elements, use brackets, e.g.: `my_program.py [FILE ...]`. Ellipsis works as unary operator on expression to the left.
- **[options]** (case sensitive) shortcut for any options. You can use it if you want to specify that usage pattern could be provided with any options defined below in option-descriptions and do not want to enumerate them all in pattern.

If your usage-patterns allow to match same-named argument several times, parser will put matched values into a list, e.g. in case pattern is `my-program.py FILE FILE` then `args['FILE']` will be a list; in case pattern is `my-program.py FILE...` it will also be a list.

3.2 Options-description format

Options-description is a list of options that you put below your ussage-patterns. It is required to list all options that are in ussage-patterns, their short/long versions (if any), and default values (if any).

- Every line in `doc` that starts with `-` or `--` (not counting spaces) is treated as an option description, e.g.:

```
Options:
  --verbose   # GOOD
  -o FILE     # GOOD
Other: --bad  # BAD, line does not start with dash "-"
```

- To specify that option has an argument, put a word describing that argument after space (or equals = sign) as shown below. You can use comma if you want to separate options. In the example below both lines are valid, however you are recommended to stick to a single style.

```
-o FILE --output=FILE       # without comma, with "=" sign
-i <file>, --input <file>   # with comma, wihtout "=" sing
```

- Use two spaces to separate options with their informal description.

```
--verbose More text.   # BAD, will be treated as if verbose option had
                       # an argument "More", so use 2 spaces instead
-q        Quit.        # GOOD
-o FILE   Output file. # GOOD
--stdout  Use stdout.  # GOOD, 2 spaces
```

- If you want to set a default value for an option with an argument, put it into the option description, in form `[default: <my-default-value>]`.

```
--coefficient=K  The K coefficient [default: 2.95]
--output=FILE    Output file [default: test.txt]
--directory=DIR  Some directory [default: ./]
```

CHAPTER

FOUR

DEVELOPMENT

`docopt` lives on github.

We would *love* to hear what you think about `docopt` on our issues page.

Contribute, make pull requrests, report bugs, suggest ideas and discuss `docopt`. You can also drop a line directly to vladimir@keleshev.com.

CHAPTER

FIVE

PORTING `DOCOPT` TO OTHER LANGUAGES

We think `docopt` is so good, we want to share it beyound the Python community!

Help developing Ruby port, or create port for your favorite language! You are encouraged to use Python version as reference implementation. Language-agnostic test-suite is on it's way to be developed.

Porting discussion is on issues page.

CHAPTER

SIX

CHANGELOG

`docopt` follows semantic versioning. First release with stable API will be 1.0 (soon). Until then you are encouraged to specify explicitly the version in your dependency tools, e.g.:

```
pip install docopt==0.3.0
```

- 0.3.0 Support for (sub)commands like `git remote add`. Introduce `[options]` shortcut for any options. **Incompatible changes**: `docopt` returns dictionary.
- 0.2.0 Usage-pattern matching. Positional arguments parsing based on usage patterns. **Incompatible changes**: `docopt` returns namespace (for arguments), not list. Usage-pattern is formalized.
- 0.1.0 Initial release. Options-parsing only (based on options-description).