docopt Documentation

Release 0.2.0

Vladimir Keleshev

May 26, 2012

CONTENTS

Note: since version 0.2 docopt parses both options and arguments, and is better than ever, however that lead to some API incompatibility with 0.1 line.

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 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 a beautiful usage-message (*the way you want it*), which adds readability to your code.

Imagine you are writing a program and thinking to allow it's usage as follows:

```
Usage: prog [-vqrh] [FILE ...]
       prog (--left | --right) CORRECTION FILE
```

Using argparse you would end up writing something like this:

```
import argparse
import sys


def process_arguments():
    parser = argparse.ArgumentParser(
            description='Process FILE and optionally apply correction to '
                        'either left-hand side or right-hand side.')
    parser.add_argument('correction', metavar='CORRECTION', nargs='*',
                        help='correction angle, needs FILE, --left or '
                             '--right to be present')
    parser.add_argument('file', metavar='FILE', nargs='?',
                        help='optional input file')
    parser.add_argument('-v', dest='v', action='store_true',
                        help='verbose mode')
    parser.add_argument('-q', dest='q', action='store_true',
                        help='quiet mode')
    parser.add_argument('-r', dest='r', action='store_true',
                        help='make report')
    left_or_right = parser.add_mutually_exclusive_group(required=False)
    left_or_right.add_argument('--left', dest='left', action='store_true',
                        help='use left-hand side')
    left_or_right.add_argument('--right', dest='right', action='store_true',
                        help='use right-hand side')
    arguments = parser.parse_args()
    if (arguments.correction and not (arguments.left or arguments.right)
            and not arguments.file):
        sys.stderr.write('correction angle, needs FILE, --left or --right '
                         'to be present')
        parser.print_help()
    return arguments


if __name__ == '__main__':
    arguments = process_arguments()
```

While docopt allows you to write an awesome, readable, pythonic code like *that*:

```
"""Usage: prog [-vqrh] [FILE ...]
          prog (--left | --right) CORRECTION FILE

Process FILE and optionally apply correction to either left-hand side or
```

```
right-hand side.

Arguments:
  FILE        optional input file
  CORRECTION  correction angle, needs FILE, --left or --right to be present

Options:
  -h --help
  -v       verbose mode
  -q       quiet mode
  -r       make report
  --left   use left-hand side
  --right  use right-hand side

"""
from docopt import docopt


if __name__ == '__main__':
    options, arguments = docopt(__doc__)  # parse arguments based on docstring above
```

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

Using `docopt` you stay DRY and follow pep257 at the same time:

> The docstring of a script (a stand-alone program) should be usable as its "usage" message, printed when the script is invoked with incorrect or missing arguments (or perhaps with a "-h" option, for "help").

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 2.6, 2.7, 3.2, and is known to work with other versions as well.

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` for your users to know about them.

The **return** value is a tuple `options, arguments`, where:

- **`options` is a namespace with option values:**
  - leading dashes (-) are stripped: `--path => options.path`
  - longer variant is given precedence: `-v --verbose => options.verbose`
  - characters not allowed in names are substituted by underscore (_): `--print-out => options.print_out`,

- **`arguments` is a namespace with argument values:**
  - leading/trailing lower/greater-than signes (used by one convention) are stripped: `<output> => arguments.output`
  - upper-case words (used by another convention) are lowered: `PATH => arguments.path`
  - characters not allowed in names are substituted by underscore (_): `<correction-angle> => arguments.correction_angle`, `HOST:PORT => arguments.host_port`

CHAPTER

THREE

USAGE-MESSAGE FORMAT

The main idea behind `docopt` is that a good usage-message (that describes options and arguments unambiguously) is enough to generate a parser.

Here are the simple rules (that you probably already follow) for your usage-message to be parsable.

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. You can also take a look at more examples.

3.1 Usage-pattern format

Usage-pattern is a substring of `doc` that starts with `usage:` (not case-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** are specified as either upper-case words, e.g. `my_program.py CONTENT-PATH` or words surrounded by greater/less-than signs: `my_program.py <content-path>`.

- **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 options-descriptions (see next section) to avoid ambiguity.
- **Optional** things. If option or argument is optional (not required), put it in brackets, e.g. `my_program.py [-hvqo FILE]`
- **Required** things. If option or argument is required (not optional), don't put it in squared brackets: `my_program.py --path=PATH FILE`. (Although "required options" might be not a good idea for your users).
- **Mutualy exclusive** things. Use horisontal bar (`|`) to specify mutually exclusive things, and group them with parenthesis (`()`): `my_program.py (--clockwise | --counter-clockwise) TIME`. You can group with brackets (`[]`) to specify that neither of mutually exclusive things are required: `my_program.py [--left | --right]`.
- **One or more** things. To specify that arbitrary number of repeating things 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 things, use brackets, e.g.: `my_program.py [FILE ...]`. Ellipsis works as unary operator on expression to the left.

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 `arguments.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 an 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. Feel free to contribute, make pull requrests, report bugs, suggest ideas and discuss `docopt` in “issues”. You can also drop me a line at vladimir@keleshev.com.