pycmdparse Documentation

Release 1.0.0

Eric Ace

Mar 17, 2019

Contents

pycmdparse is a small library to help developers of Python console utilities parse the command line and display usage instructions. It's goal is to enable this with a minimum of programming. The Use Case for a console utility developer is:

1. Import this package and subclass the `CmdLine` class in your utility code
2. Initialize the `yaml_def` field (defined in the base class) with a yaml definition of options/params/usage
3. Call the `parse` function of the base class to parse the command line.

If successful, the `parse` function injects fields into your subclass - one for each option defined in the yaml spec. Your utility then accesses the injected fields to get the values provided by the user

If there is an error parsing the command line, or the user specifies -h or –help, your utility calls the base class `display_info` method to display the errors or display usage instructions - as specified in the yaml.

CHAPTER 1

A simple example

This is an illustrative console utility called "os-info":

```
class MyCmdLine(CmdLine):
    yaml_def = '''
    utility:
      name: os-info

    summary: >
      Gets operating system info, and saves it to
      the specified file.

    positional_params:
      params: FILE
      text: >
        Writes the information to FILE

    supported_options:
      - category:
        options:
        - name : verbose
          short: v
          long : verbose
          opt  : bool
          help: >
            Provides additional (more verbose) information

    examples:
      - example: os-info -v my-outfile
        explanation: >
          Gets verbose operating system info and writes
          it to 'my-outfile' in the current working directory
    '''

    # Fields will be injected if not defined. If defined, their
```

(continues on next page)

(continued from previous page)

```
    # values will be set by the parser. The 'name' key in the
    # yaml above specifies the Python field name to inject into
    # the subclass for each option your utility supports

    verbose = None

if __name__ == "__main__":
    parse_result = MyCmdLine.parse(sys.argv)
    if parse_result.value != ParseResultEnum.SUCCESS.value:
        MyCmdLine.display_info(parse_result)
        exit(1)
    import platform
    with open(MyCmdLine.positional_params[0], "w") as f:
        f.write("sys info: %s\n" % str(platform.uname()))
        if MyCmdLine.verbose:
            f.write("python version: %s\n" %
                platform.python_version())
```

If the user entered the following on the command line:

```
os-info --help
```

They would see the following displayed on the console:

```
os-info
=======
Gets the operating system version, and saves it to the
specified file.

Usage:

os-info [-v,--verbose] FILE

Writes the information to FILE.

Options and parameters:

-v,--verbose Optional. Provides additional (more verbose)
             information

Examples:

os-info -v my-outfile

Gets verbose operating system info and writes it
to 'my-outfile' in the current working directory
```

If the user entered the following on the command line:

```
os-info --purple
```

They would see the following displayed on the console:

```
Error:

Unsupported option: '--purple'
```

(continues on next page)

(continued from previous page)

```

For usage instructions, try: os-info -h (or os-info --help)
```

Obviously with such a simple example, you wouldn't need `pycmdparse`. The library is intended to help with complex command lines.

CHAPTER 2

Terms

1. **arg**: An *arg* is a token on the command line. The first arg is the command name
2. **option**: An *option* is an argument used by the command. E.g.: –verbose. Options begin with a dash or a double dash
3. **parameter**: A *parameter* is a value that is used by an option or by the command. In this expression: `--max-threads=100`, *–max-threads* is the option and *100* is the parameter. Positional parameters are parameters used by the command that are not paired with an option. In this expression: `my-command FOO`, *FOO* is a positional param.

CHAPTER 3

Features

- Uses yaml to define command-line requirements and usage instructions
- Supports two types of options:
  - A **bool** option is true or false. Sometimes referred to as a switch. E.g.: -v, or –verbose. The value is false if omitted from the command line, and true if present on the command line
  - A **param** option takes one or more parameters. The default is a single param option. E.g.: –threads=100. A param option can be defined to accept an *exact* number of parameters, *up to* a specified number of parameters, or can accept *no limit* to the number of parameters
- Supports short-form options (-v) and long-form options (–verbose). The yaml can specify both or either.
- Supports required and non-required options. Non-required options can have a default specified in the yaml. If a required option is omitted from the command line, then it is a parse error. If a non-required option with a default is not specified on the command line, then the option value is the default in the yaml
- parameters can be expressed as follows on the command line: `--max-threads=100`. `--max-threads 100`. `-t=100`. `-t 100`. All are equivalent.
- Supports concatenation of short-form options. E.g.: `-v -t -c` and `-vtc` are handled identically. In addition, if a short-form option takes a value, it can also be concatenated. These are the same: `-v -t -c=100` and `-vtc=100` `-vtc 100` `-v -t -c 100`
- Provides basic data typing of parameters: int, bool, float, date. If you specify a data type then the parser validates the parameter so you don't have to
- For options taking multiple params, these can be provided on the command line this way: `--takes-three X Y Z` or this way: `--takes-three X --takes-three Y --takes-three Z`
- Supports the double dash ("–") option to indicate the beginning of positional parameters
- Parses positional parameters and provides them in a list
- Enables a custom validation call-back for you to perform any parameter validations not provided out of the box
- Displays usage instructions in a generally consistent form - fitted to the width of the console window so you don't have to spend time on formatting help text in your utility

- Enables you to categorize your supported options. These categories are displayed in the usage instructions. So if you have groups or related sets of options, you can categorize them for readability,
- Enables you to explicitly define a brief usage scenario - like “my-utility [options] FILE”. If you don’t explicitly define a brief usage scenario, `pycmdparse` builds one for you from the defined supported options and positional params.
- Injects fields into your subclass based on the defined options so you have an intuitive way of accessing the command line values. Boolean options are python `bool` fields. Single-value param options are scalars. Multi-valued param options and positional params are lists.

CHAPTER 4

The code

https://github.com/aceeric/pycmdparse

CHAPTER 5

Docs

5.1 Developer Guide

5.1.1 Getting Started

To use pycmdparse, you subclass the `CmdLine` class. The minimum requirement is to initialize the `yaml_def` base class field with a YAML string that defines the options and usage instructions for your utility. The intro section has an example of that. Here it is repeated.

This is an illustrative console utility called "os-info". This utility displays some information about the operating environment. This code would be in a python file in your utility:

```
import sys

from pycmdparse.abstract_opt import AbstractOpt
from pycmdparse.cmdline import CmdLine
from pycmdparse.opt_acceptresult_enum import OptAcceptResultEnum
from pycmdparse.parseresult_enum import ParseResultEnum
from pycmdparse.positional_params import PositionalParams

class MyCmdLine(CmdLine):
    yaml_def = '''
    utility:
      name: os-info

    summary: >
      Gets operating system info, and saves it to
      the specified file.

    positional_params:
      params: FILE
      text: >
        Writes the information to FILE
```

(continues on next page)

(continued from previous page)

```
    supported_options:
      - category:
        options:
        - name : verbose
          short: v
          long : verbose
          opt  : bool
          help: >
            Provides additional (more verbose) information

    examples:
      - example: os-info -v my-outfile
        explanation: >
          Gets verbose operating system info and writes
          it to 'my-outfile' in the current working directory
    '''
    verbose = None

if __name__ == "__main__":
    parse_result = MyCmdLine.parse(sys.argv)
    if parse_result.value != ParseResultEnum.SUCCESS.value:
        MyCmdLine.display_info(parse_result)
        exit(1)
    import platform
    with open(MyCmdLine.positional_params[0], "w") as f:
        f.write("sys info: %s\n" % str(platform.uname()))
        if MyCmdLine.verbose:
            f.write("python version: %s\n" %
                platform.python_version())
```

Key points:

1. The `yaml_def` base class field is initialized with yaml that defines the usage and options for the utilty
2. The main code calls the `MyCmdLine.parse()` method, passing `sys.argv` froim the Python interpreter. This initializes the base class from the yaml and then parses the command line in accordance with the yaml.
3. If the parse returns `ParseResultEnum.SUCCESS` then the code can access command line values using injected fields. In the example above, `verbose` is an injected field. (It's explicitly declared to avoid reference errors from the IDE.)
4. If the parse returns anything else, then the utility passes the return result to the base class `display_info` method to either display parse errors, or usage instructions.

5.1.2 YAML

Here is an empty schema for `pycmdparse`. The elipsis (...) indicate that a value is required. This shows the structure of the yaml. Below, each section is documented. Note - every top-level section in the yaml is optional.

```
utility:
  name: ...
  require_args: ...
summary: >
  ...
usage: >
  ...
```

(continues on next page)

(continued from previous page)

```
positional_params:
  params: ...
  text: >
    ...
supported_options:
  - category: ...
    options:
    - name      : ...
      short     : ...
      long      : ...
      hint      : ...
      opt       : ...
      required  : ...
      datatype  : ...
      multi_type: ...
      count     : ...
      help: >
        ...
details: >
  ...
examples:
  - example: ...
    explanation: >
      ...
addendum: >
  ...
```

Here are the details on the schema. In this section, example content will be provided, replacing the elipsis above. The content will be for a hypothetical *foo-utility*.

Utility

```
utility:
    name: foo-utility
    require_args: true
```

The *name* key identifies the utility name - what users will invoke on the command line. In this case, it is the *foo-utility*. In the usage instructions, this utility name displays at the top of the usage instructions, with a double underline.

If you want to require options and/or positional params, specify *require_args*: true. Then, if the user just offers the utility name on the command line with no args, the parser will return a parse result of SHOW_USAGE. If *require_args* is false in the yaml or omitted, then if the user simply types the utility name on the command line, this will not cause a parse error. This could be useful in a situation where your utility has defaults for every single command line option/param - or - doesn't support any command line options/params.

Summary

```
summary: >
    The foo-utility searches the internet for all available
    information about the etymology of 'foo'. (See
    https://en.wikipedia.org/wiki/Foobar). Various options and
    parameters can be provided as command line arguments to tailor
    the behavior of the utility.
```

Provide a top-line summary to help the user quickly understand the purpose of the utility. This displays to the console under the program name in the help.

Usage

```
usage: >
 foo-utility [options] PREVIOUSFOO
```

The *usage* section is a really brief synopsis of what the command line looks like to invoke the utility. If there is no usage section, then usage is generated to the console by pycmdparse from the defined options/parameters as well as the *positional_params*. (An example of pycmdparse-generated usage is shown in the positional params section below.)

This example provides an explicit usage section. So, whatever is provided here is displayed verbatim.

Positional Params

```
positional_params:
  params: PREVIOUSFOO
  text: >
    PREVIOUSFOO is an optional file spec. If the results of a prior
    foo analysis are available in the PREVIOUSFOO file, then the
    utility only displays the deltas between the current foo
    etymology, and the etymology saved in the specified file.
    This parameter can be an absolute - or relative - file
    specifier.
```

The existence of the *positional_params* entry causes positional param parsing. Positional params are everything after "–" on the command line, or, everything on the command line after all known options are parsed, or, everything on the command line if there are no defined options.

The *positional_params* entry contains two sub-entries: *params*, and *text*. Both are used only to format usage to the console - and only if the *usage* entry above is not provided. The value of the *params* key is appended to the supported options, and the *text* is appended to that, on a separate line. So the pycmdparse-generated usage - including supported options and positional params - for the foo-utility - would print to the console as follows, using the *positional_params* spec in this yaml:

```
Usage:

foo-utility [-v,--verbose] [-h,--help]
            [-d,--depth <n>]
            [-e,--exclude <term1 ...>] PREVIOUSFOO

PREVIOUSFOO is an optional file spec. If the results of a prior foo
analysis are available in the PREVIOUSFOO file, then the utility
only displays the deltas between the current foo etymology, and the
etymology saved in the specified file. This parameter can be an
absolute - or relative - file specifier.
```

Note that the *params* entry has no meaning to pycmdparse. It's only a mnemonic for the user.

Supported Options

```
supported_options:
  - category: Common options
    options:
    - name    : verbose
      short   : v
      long    : verbose
      opt     : bool
      help: >
        Causes verbose output. Can result in significant volumes of
        information to be emanated to the console. Use with caution.
    - name    : help
```

(continues on next page)

(continued from previous page)

```
      short    : h
      long     : help
      opt      : bool
      help: >
        Displays this help text.
  - category: Less common options
    options:
    - name      : depth
      short     : d
      long      : depth
      hint      : n
      required  : false
      datatype  : int
      opt       : param
      default   : 1
      help: >
        Specifies the recursion level of the search. If not
        specified on the command line, then a default value
        of one (1) is used. Increasing the recursion level
        can provide a better analysis result, but can
        significantly increase the processing time.
        The max value is 92.
    - name      : exclude
      short     : e
      long      : exclude
      hint      : term1 ...
      required  : false
      opt       : param
      multi_type: no-limit
      count     :
      help: >
        Specifies a list of terms that cause the utility
        to stop recursing at any given level. Multiple terms
        can be provided. There is no limit to the number
        of terms.
```

The *supported_options* entry defines the options and associated params for the utility. If this entry exists, then option parsing occurs. Otherwise, no option parsing occurs. All options support a single-character (short) form, and/or a long form. Example: `-t`, and `--timeout`. Options are case-sensitive. There are two types of options:

An example of a *bool* is: `--verbose`. It is False by default, and only True if provided on the command line. It is always optional, since it always has a value.

A *param* option is an option taking one or more params, like `--filelist FILE1 FILE2 FILE3`, or `--file FILE`. A param option's parameters are terminated differently depending on the param type. More details are provided below.

Param options are either required, or not required. Required options that are not provided on the command line cause a parse error. Non-required options can have a default in the yaml. Non-required options that are not provided on the command line and that don't have default specified have a value of `None` upon conclusion of arg parsing.

All options must belong to a category. If the category entry has a value, then it is displayed to the console when usage instructions are displayed. Otherwise the presence of the category has no effect. The purpose is to support categorization of options, which some complex utilities will want. The fact that it is required in the yaml just simplifies the pycmdparse yaml handling. Multiple categories are supported but not required.

The example foo-utility supports the following options: `--verbose`, `--exclude`, and `--depth`. `--verbose` is boolean, `--exclude` is param accepting multiple values, and `--depth` is param accepting only a single value.

Each option is an array of key/value entries. The supported keys are listed for each option type. If a key is omitted, its value is None. Each option requires either a short-form _or_ long-form option key. Both are allowed.

The table below describes the behavior of each of the keys used to define an option:

key	description
name	Optional. The Python field name that you want injected into your subclass to hold the option value. Must be a valid Python identifier. If not supplied, then `pycmdparse` will use either the long key, or the short key for the field to inject. If the long key is used, dashes in the long key are replaced by underscores to try to make a valid identifier. If an invalid identifier is defined explicitly or through derivation from the long or short key, an exception is thrown.
short	The short (single-character) option. E.g. “v” will match `-v` on the command line. Don’t include the dash in the yaml.
long	The long option. E.g. “verbose” will match `--verbose` on the command line. Either a short - or a long - option is required. Both can be provided. Don’t include the double-dash in the yaml.
opt	The option type. Either *bool*, or *param*. If omitted, then the option is defined as a *param* option taking exactly one value. E.g.: `--max-threads=1`
hint	An optional mnemonic to the user for param-type options. E.g., if you have an option `--timeout-interval`, you might define a hint of “n” to let the user know via the usage instructions that a number is expected. If you do this in the yaml, then in the usage instructions, the option displays like this: `-t, --timeout-interval <n>`
re-quired	true or false indicating that the option is required - or not - on the command line. If omitted from the yaml, the option is not required to be provided by the user. If the option is required, but not provided, then a parse error is returned by the `parse` function.
de-fault	Non-required options can have a default. If the option is not provided on the command line, it is initialized with this default value. A non-required option that is not provided and doesn’t have a default gets a value of `None` injected into your class. If the option is a mult-type (see below) then you can initialize with an array using valid yaml array syntax.
datatype	An optional data type. If you provide a data type then the params are validated against the specified type. It’s pretty limited at present: int float, bool, and date are supported. A date param matches YYYY-MM-DD, or MM-DD-YYYY with dots, dashes, or slashes as the separator. If omitted, the value is a string.
multi_type	An optional multi type for *param* options. Valid values: `exactly`, `at-most`, and `no-limit`. Works in tandem with the *count* key below. If *exactly*, then exactly <count> params are expected. Some examples are provided in a later section. If *at-most* then at most <count> params are parsed. If *no-limit*, then params are parsed until the next option is encountered on the command line - or all command line tokens are read.
count	See `multi-type` above.
help	Free-form text describing what the option does.

Details

```
details: >
  The recursion algorithm uses a weighting scheme to determine the
  amount of detailed parsing to perform at any given level of the
  search hierarchy. The following search terms illustrate the
  weighting:

     weight  term
     ------  ------
     1       foo
     2       bar
     3       baz
     4       foobar
```

The details section is just a place to put more detail than seems appropriate in the *usage* section. Some utilities have really complex options and parameters. For example, if a parameter value is itself a lookup into a table, or if there are many many usage scenarios, and so forth.Embedded newlines in the yaml are preserved (e.g. for tabular formatting if needed.) Otherwise, content is fitted by pycmdparse to the console window width.

Examples

```
examples:
  - example: foo-utility --verbose --exclude fizzbin frobozz
    explanation: >
      Performs a full traversal, with detailed diagnostic
      information displaying to the console, but terminating
      recursion into any hierarchy containing the terms
      'fizzbin', or 'frobozz'.

  - example: >
      foo-utility --verbose --exclude fizzbin frobozz --
      my-saved-search-file
    explanation: >
      Same as the example above, but in this case compares the
      results determined by the utility to the results previously
      generated in the file 'my-saved-search-file' in the current
      working directory. Only the deltas display to the console.
      (Note - the specified file must adhere to the foo-utility's
      stringent formatting requirements.)

  - example: foo-utility -d 42
    explanation: >
      Performs a search with no search term exclusions, and minimal
      (non-verbose) console output. But only recurses to
      a depth of 42.
```

The *examples* entry contains a list of *example* entries. Examples are just that. They consist of an *example* key, and an *explanation* key. They are displayed below the details section, pretty much as they appear in the yaml.

Addendum

```
addendum: >
  Version 1.2.3, Copyright (C) The Author 2019\n

  In the Public Domain\n

  Github: https://github.com/theauthor/foo-utility
```

The *addendum* section is for copyright, version, author, license, URL, anything else. Content is displayed as is, fitted to the console window width.

5.1.3 Option Examples

This section presents some examples of defining options in the yaml, and the resulting behavior of the library.

The bare minimum

```
supported_options:
  - category:
    options:
    - long: max-threads
```

The only key provided is the long option. So this will match `--max-threads` on the command line, and will be defined as a param option taking exactly one parameter. So the command line could look like: `--max-threads=1`, or `--max-threads 1`. If the command line looked like `--max-threads`, that would be a parse error. The field name injected into your subclass would be: `max_threads` and it would contain a scalar value. You would access the value thus:

```
# if cmd line is --max-threads=1, then prints "Max Threads=1":
print("Max Threads={}".format(MyCmdLine.max_threads))
```

A bool, with an explicit name, and both short and long forms

```
supported_options:
  - category:
    options:
    - name:  wax_on
      short: w
      long : wax-on
      opt  : bool
```

Matches `--wax-on` and `-w` on the command line. Always optional on the command line, because bool options are never required. Has a value of false if omitted from the command line, and a value of true if provided on the command line. The field name injected into your subclass would be: `wax_on` as explicitly defined, and it would contain a bool value, and would never have a value of `None`. You would access the value thus:

```
# if cmd line is --wax-on then prints "Wax On":
if MyCmdLine.wax_on:
    print("Wax On")
else:
    print("Wax Off")
```

A parm, taking exactly one value

```
supported_options:
  - category:
    options:
    - name    : depth
      short   : d
      long    : depth
      hint    : n
      required: false
      datatype: int
      opt     : param
      default : 1
```

In the usage instructions, the option displays like: `-d,--depth <n>` indicating that a single parameter is required that's probably a number ("n"). Since neither the *multi-type* key, nor the *count* key are specified, this defaults to an EXACTLY ONE param option. Meaning: when the command line is parsed, exactly one param is expected. So: `-d 1` would be valid. But this would be a parse error: `-d`.

Let's say you didn't define positional params. In this case, `-d 4 5 6` would also be a parse error. The reason is, the parser would initialize your option with the value 4, then "5" and "6" would not belong to anything so that would trigger a parse error. If, on the other hand, you did define positional params, then "5" and "6" would get assigned to the positional params because the rule is - after all options are parsed, everything left goes into positional params.

If the command line looked like this: `-d=123` then you would access the value thus:

```
print("Your depth plus ten is: " + str(MyCmdLine.depth + 10))
```

A parm, taking exactly three values

```
supported_options:
  - category:
    options:
    - name      : takes_3
      short     : t
      long      : takes-three
      opt       : param
      multi_type: exactly
      count     : 3
      default   :
        - ONE
        - TWO
        - THREE
```

This example is a param option taking three params. It's initialized with defaults. Since `required` is not specified, the option is not required on the command line. Let's say, in this example, that positional params are also defined. Then this is a valid command line: `--takes-three A B C 'this is a positional param'`. The parse stops as soon as it receives three params. You would access the field in your subclass like this:

```python
if len(MyCmdLine.takes_3) >= 1:
    print("First Param: " + MyCmdLine.takes_3[0])
if len(MyCmdLine.takes_3) >= 2:
    print("Second Param: " + MyCmdLine.takes_3[1])
if len(MyCmdLine.takes_3) >= 3:
    print("Third Param: " + MyCmdLine.takes_3[2])
```

(Note - the following command-line form is also supported for options taking multiple params: `--takes-three A --takes-three B --takes-three C`.) One additional thing to note about EXACTLY params is - the tokens pulled from the command line are not examined. So, if the command line looks like: `--takes-three A --foo --bar` then the value of the option will be `["A", "--foo", "--bar"]`

The reiterate, the field value injected into your subclass is a scalar for cases where the param only takes one value, and a list for cases where the param takes more than one value - as defined in the yaml. In list cases, if no params are provided and no default is defined and the option is not required, then the field value will be an empty list, vs. `None`.

A parm, taking at most three values

```
supported_options:
  - category:
    options:
    - name      : at_most_3
      long      : at-most-3
      opt       : param
      multi_type: at-most
      count     : 3
```

For `at-most` and `no-limit` *multi-type* params, the presence of the next option stops the parser from assigning parameter values to the current option. So, the following command line would be valid: `--at-most-3 ONE TWO -- POSITIONAL`. Or, if there was another option `--foo` that was supported, then this would be a valid command line: `--at-most-3 ONE TWO --foo`. In this case: `--at-most-3 ONE TWO THREE POSITIONAL`, the param picks up the values "ONE", "TWO", and "THREE" and stops gathering tokens from the command line, leaving the value "POSITIONAL" for positional params.

A parm, taking unlimited values

```
supported_options:
  - category:
```

(continues on next page)

(continued from previous page)

```
    options:
    - long      : touch-type
      opt       : param
      multi_type: no-limit
```

In this example, the command line can contain any number of params for this option, and as for the `at-most` case, the next option, or the positional params option terminates collection of params:

```
--touch-type The quick brown fox jumps over the lazy dog -- positional params
```

5.1.4 Custom Validation

You will likely have custom validation that you need to perform on your command line options. For example, you might enforce that an option value belongs to a list of valid values. Or you might require a file to exist, etc.

`pycmdparse` provides a validator call back. If you define a function in your subclass that matches this signature:

```
@classmethod
def validator(cls, to_validate):
```

. . . then once all built-in validations have passed, your validator will be called to validate each option, as well as the positional params. Here's a skeleton showing how to get started:

```
@classmethod
def validator(cls, to_validate):
    some_error_condition = False
    if isinstance(to_validate, PositionalParams):
        if some_error_condition:
            return OptAcceptResultEnum.ERROR, "TODO message"
    elif isinstance(to_validate, AbstractOpt):
        if to_validate.opt_name == "your_field":
            if some_error_condition:
                return OptAcceptResultEnum.ERROR, "TODO message"
    return None,
```

You can see that there is one if block to validate the positional params, and one if block to validate options. Your callback will be called once for each option, and once for the list of positional params. So, for example, you could enforce a specific number of positional params, etc.

Your callback is expected to return a tuple. If your validation fails, then element zero is `OptAcceptResultEnum.ERROR` as shown, and element one is a message. If there is no error, then a tuple is returned with `None` in element zero.

If your callback returns an error, then you'll pick that up in the return value from your call to the `parse` function, and it will be handled the same way as if the library determined that the command line didn't parse successfully.

Example

```
class MyCmdLine(CmdLine):
    yaml_def = '''
    utility:
      name: my-util
    supported_options:
      - category:
        options:
        - name       : it_hurts
```

(continues on next page)

(continued from previous page)

```
          long      : it-hurts
          opt       : param
          multi_type: exactly
          count     : 1
    '''

    it_hurts = None

    @classmethod
    def validator(cls, to_validate):
        if isinstance(to_validate, AbstractOpt):
            if to_validate.opt_name == "it_hurts":
                if it_hurts == "When I go like this":
                    return OptAcceptResultEnum.ERROR,
                        "Don't go like that"
        return None,

if __name__ == "__main__":
    parse_result = MyCmdLine.parse(sys.argv)
    if parse_result.value != ParseResultEnum.SUCCESS.value:
        MyCmdLine.display_info(parse_result)
        exit(1)
```

In this example, the following command line:

```
my-util --it-hurts='When I go like this'
```

Would produce the following output:

```
Error:

Don't go like that

For usage instructions, try: my-util -h (or my-util --help)
```

CHAPTER 6

Search Page

- search