
funcargparse Documentation

Release 0.2.2

Philipp Sommer

May 24, 2019

CONTENTS

1	Content	3
1.1	Getting started	3
1.1.1	Motivation	3
1.1.2	Usage	5
1.1.3	Subparsers	6
1.1.4	Chaining subparsers	7
1.2	Interpretation guidelines for docstrings	9
1.2.1	Prerequisites for the docstrings	9
1.2.2	Default interpretations	10
1.2.3	Epilog and descriptions	11
1.3	API Reference	13
1.4	Changelog	23
1.4.1	v0.2.2	23
1.4.2	v0.2.1	23
1.4.3	v0.2.0	23
	Changed	23
	Added	23
1.5	Installation	24
1.6	Requirements	24
1.7	Indices and tables	24
	Bibliography	25
	Python Module Index	27

docs	
tests	
package	

Welcome! Additionally to the default behaviour of the `argparse.ArgumentParser`, the `funcargparse.FuncArgParser` allows you to

1. automatically create a parser entirely from the docstring of a function, including the *help*, *metavar*, *action*, *type* and other parameters
2. Let's you *chain subparsers*

There are a lot of `argparse` extensions out there, but through the use of the `docrep` package, this package can extract much more information to automate the creation of the command line utility.

See *Getting started* for more information.

1.1 Getting started

1.1.1 Motivation

Suppose we want a simple script that adds or multiplies two numbers. This code should then be

1. callable inside python (i.e. we create a function)
2. executable from the command line

So let's setup the function in a file called 'add_or_multiply.py' like this

```
In [1]: def do_something(a, b, multiply=False):
...:     """
...:     Multiply or add one number to the others
...:
...:     Parameters
...:     -----
...:     a: int
...:         Number 1
...:     b: list of int
...:         A list of numbers to add `a` to
...:     multiply: bool
...:         If True, the numbers are multiplied, not added
...:     """
...:     if multiply:
...:         result = [n * a for n in b]
...:     else:
...:         result = [n + a for n in b]
...:     print(result)
...:
```

Now, if you want to make a command line script out of it, the usual methodology is to create an `argparse.ArgumentParser` instance and parse the arguments like this

```
In [2]: if __name__ == '__main__':
...:     from argparse import ArgumentParser
...:     parser = ArgumentParser(
...:         description='Multiply or add two numbers')
...:     parser.add_argument('a', type=int, help='Number 1')
...:     parser.add_argument('b', type=int, nargs='+',
...:                         help='A list of numbers to add `a` to')
...:     parser.add_argument('-m', '--multiply', action='store_true',
```

(continues on next page)

(continued from previous page)

```

....:             help='Multiply the numbers instead of adding them')
....:     args = parser.parse_args('3 2 -m'.split())
....:     do_something(**vars(args))
....:

```

Now, if you parse the arguments, you get

```

In [3]: parser.print_help()
usage: sphinx-build [-h] [-m] a b [b ...]

Multiply or add two numbers

positional arguments:
  a                Number 1
  b                A list of numbers to add `a` to

optional arguments:
  -h, --help      show this help message and exit
  -m, --multiply  Multiply the numbers instead of adding them

```

However, you could skip the entire lines above, if you just use the `funcargparse.FuncArgParser`

```

In [4]: from funcargparse import FuncArgParser

In [5]: parser = FuncArgParser()

In [6]: parser.setup_args(do_something)
Out[6]: <function __main__.do_something(a, b, multiply=False)>

In [7]: parser.update_short(multiply='m')

In [8]: actions = parser.create_arguments()

In [9]: parser.print_help()
usage: sphinx-build [-h] [-m] int int [int ...]

Multiply or add one number to the others

positional arguments:
  int                Number 1
  int                A list of numbers to add `a` to

optional arguments:
  -h, --help      show this help message and exit
  -m, --multiply  If True, the numbers are multiplied, not added

```

or you use the parser right in the beginning as a decorator

```

In [10]: @parser.update_shortf(multiply='m')
....: @parser.setup_args
....: def do_something(a, b, multiply=False):
....:     """
....:     Multiply or add one number to the others
....:
....:     Parameters
....:     -----
....:     a: int

```

(continues on next page)

(continued from previous page)

```

.....:     Number 1
.....:     b: list of int
.....:     A list of numbers to add `a` to
.....:     multiply: bool
.....:     If True, the numbers are multiplied, not added
.....:     """
.....:     if multiply:
.....:         result = [n * a for n in b]
.....:     else:
.....:         result = [n + a for n in b]
.....:     print(result)
.....:

```

```
In [11]: actions = parser.create_arguments()
```

```
In [12]: parser.print_help()
```

```
usage: sphinx-build [-h] [-m] int int [int ...]
```

Multiply or add one number to the others

positional arguments:

```
int           Number 1
int           A list of numbers to add `a` to
```

optional arguments:

```
-h, --help      show this help message and exit
-m, --multiply  If True, the numbers are multiplied, not added
```

The `FuncArgParser` interpretes the docstring (see *Interpretation guidelines for docstrings*) and sets up the arguments.

Your `'__main__'` part could then simply look like

```
In [13]: if __name__ == '__main__':
.....:     parser.parse_to_func()
.....:
```

1.1.2 Usage

Generally the usage is

1. create an instance of the `FuncArgParser` class
2. setup the arguments using the `setup_args()` function
3. modify the arguments (optional) either
 - a. in the `FuncArgParser.unfinished_arguments` dictionary
 - b. using the `update_arg()`, `update_short()`, `update_long()` or `append2help()` methods
 - c. using the equivalent decorator methods `update_argf()`, `update_shortf()`, `update_longf()` or `append2helpf()`
4. create the arguments using the `create_arguments()` method

1.1.3 Subparsers

You can also use subparsers for controlling you program (see the `argparse.ArgumentParser.add_subparsers()` method). They can either be implemented the classical way via

```
In [14]: subparsers = parser.add_subparsers()
In [15]: subparser = subparsers.add_parser('test')
```

And then as with the parent parser you can use function docstrings.

```
In [16]: @subparser.setup_args
...: def my_other_func(b=1):
...:     """
...:     Subparser summary
...:
...:     Parameters
...:     -----
...:     b: int
...:         Anything"""
...:     print(b * 500)
...:

In [17]: subparser.create_arguments()
Out[17]: [_StoreAction(option_strings=['-b'], dest='b', nargs=None, const=None, ↵
↵default=1, type=<class 'int'>, choices=None, help='Anything', metavar='int')]

In [18]: parser.print_help()
////////////////////////////////////
↵sphinx-build [-h] {test} ...

positional arguments:
  {test}

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

On the other hand, you can use the `setup_subparser()` method to directly create the subparser

```
In [19]: parser.add_subparsers()
Out[19]: _SubParsersAction(option_strings=[], dest=='SUPPRESS==', nargs='A...', ↵
↵const=None, default=None, type=None, choices={}, help=None, metavar=None)

In [20]: @parser.setup_subparser
...: def my_other_func(b=1):
...:     """
...:     Subparser summary
...:
...:     Parameters
...:     -----
...:     b: int
...:         Anything"""
...:     print(b * 500)
...:

In [21]: parser.create_arguments(subparsers=True)
Out[21]: []
```

(continues on next page)

(continued from previous page)

```
In [22]: parser.print_help()
\\repeated backslashes usage: sphinx-build [-h] {my-other-func} ...

positional arguments:
  {my-other-func}
  my-other-func  Subparser summary

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

which now created the `my-other-func` sub command.

1.1.4 Chaining subparsers

Separate from the usage of the function docstring, we implemented the possibility to chain subparsers. This changes the handling of subparsers compared to the default behaviour (which is inherited from the `argparse.ArgumentParser`). The difference can be shown in the following example

```
In [23]: from argparse import ArgumentParser

In [24]: argparser = ArgumentParser()

In [25]: funcargparser = FuncArgParser()

In [26]: sps_argparse = argparser.add_subparsers()

In [27]: sps_funcargparse = funcargparser.add_subparsers(chain=True)

In [28]: sps_argparse.add_parser('dummy').add_argument('-a')
Out [28]: _StoreAction(option_strings=['-a'], dest='a', nargs=None, const=None, ↵
↵default=None, type=None, choices=None, help=None, metavar=None)

In [29]: sps_funcargparse.add_parser('dummy').add_argument('-a')
\\repeated backslashes
↵_StoreAction(option_strings=['-a'], dest='a', nargs=None, const=None, default=None, ↵
↵type=None, choices=None, help=None, metavar=None)

In [30]: ns_default = argparser.parse_args('dummy -a 3'.split())

In [31]: ns_chained = funcargparser.parse_args('dummy -a 3'.split())

In [32]: print(ns_default, ns_chained)
Namespace(a='3') Namespace(dummy=Namespace(a='3'))
```

So while the default behaviour is, to put the arguments in the main namespace like

```
In [33]: ns_default.a
Out [33]: '3'
```

the chained subparser procedure puts the commands for the `'dummy'` command into an extra namespace like

```
In [34]: ns_chained.dummy.a
Out [34]: '3'
```

This has the advantages that we don't mix up subparsers if we chain them. So here is an example demonstrating the power of it

```
In [35]: sps_argparse.add_parser('dummy2').add_argument('-a')
Out[35]: _StoreAction(option_strings=['-a'], dest='a', nargs=None, const=None, ↵
↵default=None, type=None, choices=None, help=None, metavar=None)

In [36]: sps_funcargparse.add_parser('dummy2').add_argument('-a')
//////////////////////////////////////
↵_StoreAction(option_strings=['-a'], dest='a', nargs=None, const=None, default=None, ↵
↵type=None, choices=None, help=None, metavar=None)

# with allowing chained subcommands, we get
In [37]: ns_chained = funcargparser.parse_args('dummy -a 3 dummy2 -a 4'.split())

In [38]: print(ns_chained.dummy.a, ns_chained.dummy2.a)
3 4

# on the other side, the default ArgumentParser raises an error because
# chaining is not allowed
In [39]: ns_default = argparse.parse_args('dummy -a 3 dummy2 -a 4'.split())
\\\\\\An exception has occurred, use %tb to see the full traceback.

SystemExit: 2
```

Furthermore, you can use the `parse_chained()` and the `parse_known_chained()` methods to parse directly to the subparsers.

```
In [40]: parser = FuncArgParser()

In [41]: sps = parser.add_subparsers(chain=True)

In [42]: @parser.setup_subparser
.....: def subcommand_1():
.....:     print('Calling subcommand 1')
.....:     return 1
.....:

In [43]: @parser.setup_subparser
.....: def subcommand_2():
.....:     print('Calling subcommand 2')
.....:     return 2
.....:

In [44]: parser.create_arguments(True)
Out[44]: []

In [45]: parser.parse_chained('subcommand-1 subcommand-2'.split())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Calling subcommand 2
Calling subcommand 1
Out[45]: Namespace(subcommand_1=1, subcommand_2=2)
```

Warning: If you reuse an already existing command in the subcommand of another subcommand, the latter one get's preferred. See this example

(continued from previous page)

```

...
...     Parameters
...     -----
...     a: int
...         The documentation of a"""
...     pass

```

1.2.2 Default interpretations

To make common arguments more accessible, the `setup_args()` method already has some preconfigured settings:

1. non-optional (e.g. name in `def my_func(name, a=1):` ... automatically get the 'positional' flag. You can remove that via:

```
>>> parser.pop_key('<argname>', 'positional')
```

2. if the default argument in the function is a boolean and the specified type is 'bool' (e.g.:

```

>>> def my_func(switch=False):
...     """
...     Some function
...
...     Parameters
...     -----
...     switch: bool
...         This will be inserted as a switch
...     """
The default settings for the `switch` parameter are ``action='store_true``
(or ``action='store_false`` if the default value would be ``True``)

```

3. If the type specification in the docstring corresponds to a builtin type (e.g. 'float', 'int', 'str'), this type will be used to interpret the commandline arguments. For example:

```

>>> def my_func(num=0):
...     """
...     Some function
...
...     Parameters
...     -----
...     num: float
...         A floating point number
...     """

```

will for the 'num' parameter lead to the settings `type=float`

4. If the type description starts with 'list of', it will allow multiple arguments and interpretes everything after it as the type of it. For example:

```

>>> def my_func(num=[1, 2, 3]):
...     """
...     Some function
...
...     Parameters
...     -----
...     num: list of floats

```

(continues on next page)

1.3 API Reference

Classes

<code>FuncArgParser(*args, **kwargs)</code>	Subclass of an argument parser that get's parts of the information
---	--

class FuncArgParser (*args, **kwargs)

Bases: `argparse.ArgumentParser`

Subclass of an argument parser that get's parts of the information from a given function

Parameters ***args, **kwargs** – These arguments are determined by the `argparse.ArgumentParser` base class. Note that by default, we use a `argparse.RawTextHelpFormatter` class for the `formatter_class` keyword, whereas the `argparse.ArgumentParser` uses a `argparse.HelpFormatter`

Other Parameters

- **epilog_sections** (*list of str*) – The default sections to use for the epilog (see the `epilog_sections` attribute). They can also be specified each time the `setup_args()` method is called
- **epilog_formatter** (*{'header', 'bold', 'rubric'} or function*) – Specify how the epilog sections should be formatted and defaults to the `epilog_formatter` attribute. This can either be a string out of 'header', 'bold', or 'rubric' or a callable (i.e. function) that takes two arguments, the section title and the section text, and returns a string.

'heading' Use section headers such as:

```
Notes
-----
```

'bold' Just make a bold header for the section, e.g. `**Notes**`

'rubric' Use a rubric rst directive, e.g. `.. rubric:: Notes`

Methods

<code>add_subparsers(*args, **kwargs)</code>	Add subparsers to this parser
<code>create_arguments([subparsers])</code>	Create and add the arguments
<code>extract_as_epilog(text[, sections, ...])</code>	Extract epilog sections from the a docstring
<code>format_bold(section, text)</code>	Make a bold formatting for the section header
<code>format_epilog_section(section, text)</code>	Format a section for the epilog by inserting a format
<code>format_heading(section, text)</code>	Make a bold formatting for the section header
<code>format_rubric(section, text)</code>	Make a bold formatting for the section header
<code>get_param_doc(doc, param)</code>	Get the documentation and datatype for a parameter
<code>get_subparser(name)</code>	Convenience method to get a certain subparser
<code>grouparg(arg[, my_arg, parent_cmds])</code>	Grouping function for chaining subcommands
<code>parse2func([args, func])</code>	Parse the command line arguments to the setup function
<code>parse_chained([args])</code>	Parse the argument directly to the function used for setup
<code>parse_known2func([args, func])</code>	Parse the command line arguments to the setup function

Continued on next page

Table 2 – continued from previous page

<code>parse_known_args([args, namespace])</code>	
<code>parse_known_chained([args])</code>	Parse the argument directly to the function used for setup
<code>setup_subparser([func, setup_as, insert_at, ...])</code>	Create a subparser with the name of the given function

Modification methods

<code>append2help(arg, s)</code>	Append the given string to the help of argument <i>arg</i>
<code>pop_arg(*args, **kwargs)</code>	Delete a previously defined argument from the parser
<code>pop_key(arg, key, *args, **kwargs)</code>	Delete a previously defined key for the <i>add_argument</i>
<code>update_arg(arg[, if_existent])</code>	Update the <i>add_argument</i> data for the given parameter
<code>update_long(**kwargs)</code>	Update the long optional arguments (those with two leading '-')
<code>update_short(**kwargs)</code>	Update the short optional arguments (those with one leading '-')

Decorator methods

<code>append2helpf(arg, s)</code>	Append the given string to the help of argument <i>arg</i>
<code>pop_argf(*args, **kwargs)</code>	Delete a previously defined argument from the parser via decorators
<code>pop_keyf(*args, **kwargs)</code>	Delete a previously defined key for the <i>add_argument</i>
<code>setup_args([func, setup_as, insert_at, ...])</code>	Add the parameters from the given <i>func</i> to the parameter settings
<code>update_argf(arg, **kwargs)</code>	Update the arguments as a decorator
<code>update_longf(**kwargs)</code>	Update the long optional arguments belonging to a function
<code>update_shortf(**kwargs)</code>	Update the short optional arguments belonging to a function

Attributes

<code>epilog_formatter</code>	The formatter specification for the epilog.
<code>epilog_sections</code>	The sections to extract from a function docstring that should be used in the epilog of this parser.
<code>unfinished_arguments</code>	The unfinished arguments after the setup

add_subparsers (*args, **kwargs)

Add subparsers to this parser

Parameters

- ****kwargs** (*args,) – As specified by the original `argparse.ArgumentParser.add_subparsers()` method
- **chain** (*bool*) – Default: False. If True, It is enabled to chain subparsers

append2help (*arg, s*)

Append the given string to the help of argument *arg*

Parameters

- **arg** (*str*) – The function argument
- **s** (*str*) – The string to append to the help

append2helpf (*arg, s*)

Append the given string to the help of argument *arg*

Parameters

- **arg** (*str*) – The function argument
- **s** (*str*) – The string to append to the help

create_arguments (*subparsers=False*)

Create and add the arguments

Parameters **subparsers** (*bool*) – If True, the arguments of the subparsers are also created

epilog_formatter = 'heading'

The formatter specification for the epilog. This can either be a string out of 'header', 'bold', or 'rubric' or a callable (i.e. function) that takes two arguments, the section title and the section text, and returns a string.

'heading' Use section headers such as:

```
Notes
-----
```

'bold' Just make a bold header for the section, e.g. `**Notes**`

'rubric' Use a rubric rst directive, e.g. `.. rubric:: Notes`

Warning: When building a sphinx documentation using the sphinx-argparse module, this value should be set to 'bold' or 'rubric'! Just add this two lines to your conf.py:

```
import funcargparse
funcargparse.FuncArgParser.epilog_formatter = 'rubric'
```

epilog_sections = ['Notes', 'References']

The sections to extract from a function docstring that should be used in the epilog of this parser. See also the `setup_args()` method

extract_as_epilog (*text, sections=None, overwrite=False, append=True*)

Extract epilog sections from the a docstring

Parameters

- **text** – The docstring to use
- **sections** (*list of str*) – The headers of the sections to extract. If None, the `epilog_sections` attribute is used
- **overwrite** (*bool*) – If True, overwrite the existing epilog
- **append** (*bool*) – If True, append to the existing epilog

static format_bold (*section, text*)

Make a bold formatting for the section header

format_epilog_section (*section, text*)

Format a section for the epilog by inserting a format

static format_heading (*section, text*)

static format_rubric (*section, text*)

Make a bold formatting for the section header

static get_param_doc (*doc, param*)

Get the documentation and datatype for a parameter

This function returns the documentation and the argument for a napoleon like structured docstring *doc*

Parameters

- **doc** (*str*) – The base docstring to use
- **param** (*str*) – The argument to use

Returns

- *str* – The documentation of the given *param*
- *str* – The datatype of the given *param*

get_subparser (*name*)

Convenience method to get a certain subparser

Parameters **name** (*str*) – The name of the subparser

Returns The subparsers corresponding to *name*

Return type *FuncArgParser*

grouparg (*arg, my_arg=None, parent_cmds=[]*)

Grouping function for chaining subcommands

Parameters

- **arg** (*str*) – The current command line argument that is parsed
- **my_arg** (*str*) – The name of this subparser. If None, this parser is the main parser and has no parent parser
- **parent_cmds** (*list of str*) – The available commands of the parent parsers

Returns The grouping key for the given *arg* or None if the key does not correspond to this parser or this parser is the main parser and does not have seen a subparser yet

Return type *str* or *None*

Notes

Quite complicated, there is no real need to deal with this function

parse2func (*args=None, func=None*)

Parse the command line arguments to the setup function

This method parses the given command line arguments to the function used in the *setup_args()* method to setup up this parser

Parameters

- **args** (*list*) – The list of command line arguments

- **func** (*function*) – An alternative function to use. If None, the last function or the one specified through the *setup_as* parameter in the *setup_args()* is used.

Returns What ever is returned by the called function

Return type `object`

Note: This method does not cover subparsers!

parse_chained (*args=None*)

Parse the argument directly to the function used for setup

This function parses the command line arguments to the function that has been used for the *setup_args()*.

Parameters **args** (*list*) – The arguments parsed to the *parse_args()* function

Returns The namespace with mapping from command name to the function return

Return type `argparse.Namespace`

See also:

parse_known_chained()

parse_known2func (*args=None, func=None*)

Parse the command line arguments to the setup function

This method parses the given command line arguments to the function used in the *setup_args()* method to setup up this parser

Parameters

- **args** (*list*) – The list of command line arguments
- **func** (*function or str*) – An alternative function to use. If None, the last function or the one specified through the *setup_as* parameter in the *setup_args()* is used.

Returns

- *object* – What ever is returned by the called function
 - *list* – The remaining command line arguments that could not be interpreted
-

Note: This method does not cover subparsers!

parse_known_args (*args=None, namespace=None*)

parse_known_chained (*args=None*)

Parse the argument directly to the function used for setup

This function parses the command line arguments to the function that has been used for the *setup_args()* method.

Parameters **args** (*list*) – The arguments parsed to the *parse_args()* function

Returns

- *argparse.Namespace* – The namespace with mapping from command name to the function return
 - *list* – The remaining arguments that could not be interpreted
-

See also:

`parse_known()`

pop_arg (**args, **kwargs*)

Delete a previously defined argument from the parser

pop_argf (**args, **kwargs*)

Delete a previously defined argument from the parser via decorators

Same as `pop_arg()` but it can be used as a decorator

pop_key (*arg, key, *args, **kwargs*)

Delete a previously defined key for the `add_argument`

pop_keyf (**args, **kwargs*)

Delete a previously defined key for the `add_argument`

Same as `pop_key()` but it can be used as a decorator

setup_args (*func=None, setup_as=None, insert_at=None, interpret=True, epilog_sections=None, overwrite=False, append_epilog=True*)

Add the parameters from the given `func` to the parameter settings

Parameters

- **func** (*function*) – The function to use. If `None`, a function will be returned that can be used as a decorator
- **setup_as** (*str*) – The attribute that shall be assigned to the function in the resulting namespace. If specified, this function will be used when calling the `parse2func()` method
- **insert_at** (*int*) – The position where the given `func` should be inserted. If `None`, it will be appended at the end and used when calling the `parse2func()` method
- **interpret** (*bool*) – If `True` (default), the docstrings are interpreted and switches and lists are automatically inserted (see the [interpretation-docs])
- **epilog_sections** (*list of str*) – The headers of the sections to extract. If `None`, the `epilog_sections` attribute is used
- **overwrite** (*bool*) – If `True`, overwrite the existing epilog and the existing description of the parser
- **append_epilog** (*bool*) – If `True`, append to the existing epilog

Returns Either the function that can be used as a decorator (if `func` is `None`), or the given `func` itself.

Return type function

Examples

Use this method as a decorator:

```

>>> @parser.setup_args
... def do_something(a=1):
...     '''
...     Just an example
...
...     Parameters
...     -----

```

(continues on next page)

(continued from previous page)

```

a: int
    A number to increment by one
'''
return a + 1
>>> args = parser.parse_args('-a 2'.split())

```

Or by specifying the `setup_as` function:

```

>>> @parser.setup_args(setup_as='func')
... def do_something(a=1):
    '''
    Just an example

    Parameters
    -----
    a: int
        A number to increment by one
    '''
    return a + 1
>>> args = parser.parse_args('-a 2'.split())
>>> args.func is do_something
>>> parser.parse2func('-a 2'.split())
3

```

References

setup_subparser (*func=None, setup_as=None, insert_at=None, interpret=True, epilogs=None, log_sections=None, overwrite=False, append_epilog=True, return_parser=False, name=None, **kwargs*)

Create a subparser with the name of the given function

Parameters are the same as for the `setup_args()` function, other parameters are parsed to the `add_subparsers()` method if (and only if) this method has not already been called.

Parameters

- **func** (*function*) – The function to use. If `None`, a function will be returned that can be used as a decorator
- **setup_as** (*str*) – The attribute that shall be assigned to the function in the resulting namespace. If specified, this function will be used when calling the `parse2func()` method
- **insert_at** (*int*) – The position where the given `func` should be inserted. If `None`, it will be appended at the end and used when calling the `parse2func()` method
- **interpret** (*bool*) – If `True` (default), the docstrings are interpreted and switches and lists are automatically inserted (see the [interpretation-docs])
- **epilog_sections** (*list of str*) – The headers of the sections to extract. If `None`, the `epilog_sections` attribute is used
- **overwrite** (*bool*) – If `True`, overwrite the existing epilog and the existing description of the parser
- **append_epilog** (*bool*) – If `True`, append to the existing epilog
- **return_parser** (*bool*) – If `True`, the create parser is returned instead of the function

- **name** (*str*) – The name of the created parser. If *None*, the function name is used and underscores ('_') are replaced by minus ('-')
- ****kwargs** – Any other parameter that is passed to the `add_parser` method that creates the parser

Returns Either the function that can be used as a decorator (if *func* is *None*), or the given *func* itself. If `return_parser` is *True*, the created subparser is returned

Return type *FuncArgParser* or function

Examples

Use this method as a decorator:

```
>>> from funcargparse import FuncArgParser
>>> parser = FuncArgParser()
>>> @parser.setup_subparser
... def my_func(my_argument=None):
...     pass
>>> args = parser.parse_args('my-func -my-argument 1'.split())
```

unfinished_arguments = {}

The unfinished arguments after the setup

update_arg (*arg*, *if_existent=None*, ***kwargs*)

Update the *add_argument* data for the given parameter

Parameters

- **arg** (*str*) – The name of the function argument
- **if_existent** (*bool* or *None*) – If *True*, the argument is updated. If *None* (default), the argument is only updated, if it exists. Otherwise, if *False*, the given ***kwargs* are only used if the argument is not yet existing
- ****kwargs** – The keyword arguments any parameter for the `argparse.ArgumentParser.add_argument()` method

update_argf (*arg*, ***kwargs*)

Update the arguments as a decorator

Parameters

- **arg** (*str*) – The name of the function argument
- **if_existent** (*bool* or *None*) – If *True*, the argument is updated. If *None* (default), the argument is only updated, if it exists. Otherwise, if *False*, the given ***kwargs* are only used if the argument is not yet existing
- ****kwargs** – The keyword arguments any parameter for the `argparse.ArgumentParser.add_argument()` method

Examples

Use this method as a decorator:


```

>>> from funcargparse import FuncArgParser

>>> parser = FuncArgParser()

>>> @parser.update_argf('my_argument', type=int)
... def my_func(my_argument=None):
...     pass

>>> args = parser.parse_args('my-func -my-argument 1'.split())

>>> isinstance(args.my_argument, int)
True

```

See also:

update_arg()

update_long (**kwargs)

Update the long optional arguments (those with two leading '-')

This method updates the short argument name for the specified function arguments as stored in *unfinished_arguments*

Parameters ****kwargs** – Keywords must be keys in the *unfinished_arguments* dictionary (i.e. keywords of the root functions), values the long argument names

Examples

Setting:

```

>>> parser.update_long(something='s', something_else='se')

```

is basically the same as:

```

>>> parser.update_arg('something', long='s')
>>> parser.update_arg('something_else', long='se')

```

which in turn is basically comparable to:

```

>>> parser.add_argument('--s', dest='something', ...)
>>> parser.add_argument('--se', dest='something_else', ...)

```

See also:

update_short(), *update_longf()*

update_longf (**kwargs)

Update the long optional arguments belonging to a function

This method acts exactly like *update_long()* but works as a decorator (see *update_arg()* and *update_argf()*)

Parameters ****kwargs** – Keywords must be keys in the *unfinished_arguments* dictionary (i.e. keywords of the root functions), values the long argument names

Returns The function that can be used as a decorator

Return type function

Examples

Use this method as a decorator:

```
>>> @parser.update_shortf(something='s', something_else='se')
... def do_something(something=None, something_else=None):
...     ...
```

See also the examples in `update_long()`.

See also:

`update_short()`, `update_longf()`

`update_short(**kwargs)`

Update the short optional arguments (those with one leading '-')

This method updates the short argument name for the specified function arguments as stored in `unfinished_arguments`

Parameters ****kwargs** – Keywords must be keys in the `unfinished_arguments` dictionary (i.e. keywords of the root functions), values the short argument names

Examples

Setting:

```
>>> parser.update_short(something='s', something_else='se')
```

is basically the same as:

```
>>> parser.update_arg('something', short='s')
>>> parser.update_arg('something_else', short='se')
```

which in turn is basically comparable to:

```
>>> parser.add_argument('-s', '--something', ...)
>>> parser.add_argument('-se', '--something_else', ...)
```

See also:

`update_shortf()`, `update_long()`

`update_shortf(**kwargs)`

Update the short optional arguments belonging to a function

This method acts exactly like `update_short()` but works as a decorator (see `update_arg()` and `update_argf()`)

Parameters ****kwargs** – Keywords must be keys in the `unfinished_arguments` dictionary (i.e. keywords of the root functions), values the short argument names

Returns The function that can be used as a decorator

Return type function

Examples

Use this method as a decorator:

```
>>> @parser.update_shortf(something='s', something_else='se')
... def do_something(something=None, something_else=None):
...     ...
```

See also the examples in `update_short()`.

See also:

`update_short()`, `update_longf()`

1.4 Changelog

1.4.1 v0.2.2

Minor patch to use `inspect.getfullargspec` instead of `inspect.getargspec` because the latter has been deprecated in python 3.0

1.4.2 v0.2.1

Small patch to use `inspect.cleandoc` instead of `docrep.dedents`

1.4.3 v0.2.0

This release just adds some new interpretation features to extract the parser description and the epilog from the parsed function. They might changed how your parser looks drastically!

Changed

- The default `formatter_class` for the `FuncArgParser` is now the `argparse.RawHelpFormatter`, which makes sense since we expect that the documentations are already nicely formatted
- When calling the `FuncArgParser.setup_args()` method, we also look for the *Notes* and *References* sections which will be included in the epilog of the parser. If you want to disable this feature, just initialize the parser with:

```
parser = FuncArgParser(epilog_sections=[])
```

This feature might cause troubles when being used in sphinx documentations in conjunction with the `sphinx-argparse` package. For this, you can change the formatting of the heading with the `FuncArgParser.epilog_formatter` attribute

Added

- Changelog

1.5 Installation

Simply install it via `pip`:

```
$ pip install funcargparse
```

Or you install it via:

```
$ python setup.py install
```

from the source on [GitHub](#).

1.6 Requirements

The package only requires the `docrep` package which we use under the hood to extract the necessary parts from the docstrings.

The package has been tested for python 2.7 and 3.5.

1.7 Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)

BIBLIOGRAPHY

[interpretation-docs] (http://funcargparse.readthedocs.io/en/latest/docstring_interpretation.html)

PYTHON MODULE INDEX

f

[funcargparse](#), 13

A

add_subparsers() (*FuncArgParser* method), 14
 append2help() (*FuncArgParser* method), 14
 append2helpf() (*FuncArgParser* method), 15

C

create_arguments() (*FuncArgParser* method), 15

E

epilog_formatter (*FuncArgParser* attribute), 15
 epilog_sections (*FuncArgParser* attribute), 15
 extract_as_epilog() (*FuncArgParser* method), 15

F

format_bold() (*FuncArgParser* static method), 15
 format_epilog_section() (*FuncArgParser* method), 15
 format_heading() (*FuncArgParser* static method), 16
 format_rubric() (*FuncArgParser* static method), 16
 funcargparse (module), 13
 FuncArgParser (class in *funcargparse*), 13

G

get_param_doc() (*FuncArgParser* static method), 16
 get_subparser() (*FuncArgParser* method), 16
 grouparg() (*FuncArgParser* method), 16

P

parse2func() (*FuncArgParser* method), 16
 parse_chained() (*FuncArgParser* method), 17
 parse_known2func() (*FuncArgParser* method), 17
 parse_known_args() (*FuncArgParser* method), 17
 parse_known_chained() (*FuncArgParser* method), 17
 pop_arg() (*FuncArgParser* method), 18
 pop_argf() (*FuncArgParser* method), 18
 pop_key() (*FuncArgParser* method), 18
 pop_keyf() (*FuncArgParser* method), 18

S

setup_args() (*FuncArgParser* method), 18
 setup_subparser() (*FuncArgParser* method), 19

U

unfinished_arguments (*FuncArgParser* attribute), 20
 update_arg() (*FuncArgParser* method), 20
 update_argf() (*FuncArgParser* method), 20
 update_long() (*FuncArgParser* method), 21
 update_longf() (*FuncArgParser* method), 21
 update_short() (*FuncArgParser* method), 22
 update_shortf() (*FuncArgParser* method), 22