argdeco Documentation

Release 3.0.0

Kay-Uwe (Kiwi) Lorenz

Jul 20, 2021

Contents

I like `argparse` module very much, because you can create great and sophisticating command line argument configurations. But if you create many small tools or if you even work with subcommands, it gets a bit cumbersome and you produce a lot of code, which destracts from essentials.

This module aims to ease creating of command line interfaces using the power of decorators wrapping `argparse`.

CHAPTER 1

Install

Install it using pip:

```
pip install argdeco
```

CHAPTER 2

Overview

Example for a simple main:

```
from argdeco import main, arg, opt

@main(
    arg('--input', '-i', help="input file", default="-"),
    arg('--output', '-o', help="output file", default="-"),
    opt('--flag'),
)
def main(input, output, flag):
    input  = (input  == '-') and sys.stdin  or open(input,  'r')
    output = (output == '-') and sys.stdout or open(output, 'w')

    if flag:
       print("flagged")

    for line in input:
        output.write(line)

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

You can run the command with:

```
$ echo "x" | copy.py --flag
flagged
x
```

Example for commands:

```
from argdeco import main, command, arg

@command( "hello", arg("greet", help="the one to greet") )
def greet(greet):
```

(continues on next page)

(continued from previous page)

```
    """
    greet a person.

    This command will print out a greeting to the person
    named in the argument.
    """
    print("hello %s" % greet)

@command( "bye", arg("greet", help="the one to say goodbye") )
def bye(greet):
    """
    say goodbye to a person.

    This command will print out a goodbye to the person
    named in the argument.
    """
    print("goodbye %s" % greet)

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

Here some run examples:

```
$ greet.py hello "Mr. Bean"
hello Mr. Bean

$ greet.py bye "Mr. Bean"
goodby Mr. Bean
```

You might have noticed, that arguments passed to `arg` are the same like the ones passed to `argparse.ArgumentParser.add_argument()`.

CHAPTER 3

Bash completion

argdeco uses argcomplete by default.

3.1 Enable completion for your script

If you install your script as `myscript` executable, you have to make sure, that following line is in the user's `~/.bashrc`:

```
eval "$(register-python-argcomplete myscript)"
```

For convenience you can run:

```
python -m argdeco install-bash-completions myscript
```

Or:

```
python -m argdeco install-bash-completions myscript --dest ~/.profile
```

For uninstalling run:

```
python -m argdeco uninstall-bash-completions myscript
```

3.2 Global completion activation

For activating it globally, a user has to activate global completion.

Note: On a Ubuntu system, you can it install as a user with:

```
activate-global-python-argcomplete --user
```

This installs the completion script to `~/.bash_completion.d/`. This is not automatically invoked.

So create a `~/.bash_completion` file to enable `~/.bash_completion.d/`:

```
echo "for f in ~/.bash_completion/* ; do source $f ; done" > ~/.bash_completion
```

In your script you have to make sure, that the string PYTHON_ARGCOMPLETE_OK can be found within the first 1024 characters in your executable.

If you use a custom python script (installed via setup scripts) as entry point, you can achieve this by importing a symbol for activating completion:

```
from argdeco import main, command, arg, PYTHON_ARGCOMPLETE_OK
```

If you specify an entry point in your setup.py, you should call the entrypoint PYTHON_ARGCOMPLETE_OK:

```
setup(
   # ...
   entry_points={
      'console_scripts': [
          'myscript = myscript.cli:PYTHON_ARGCOMPLETE_OK',
      ]
   }
   # ...
)
```

And in your module `myscript.cli`:

```
from argdeco import main as PYTHON_ARGCOMPLETE_OK, main

@main
def my_main():
   pass
```

CHAPTER 4

Contents

4.1 argdeco

argdeco – use argparse with decorators

This module is main user interface.

4.1.1 Quickstart

If you want to create a simple program:

```
from argdeco import main, arg, opt

@main(
    arg('--first', help="first argument"),
    opt('--flag-1', help="toggle 1st thing"),
    opt('--flag-2', help="toggle 2nd thing"),
)
def my_main(first, flag_1, flag_2):
    # do something here
    pass

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

If you want to create a program with subcommands:

```
from argdeco import main, command, arg, opt

@command('cmd1')
def cmd1(global_arg):
    print("this is the first command")
```

(continues on next page)

(continued from previous page)

```
@command("cmd2", arg("--foo-bar"))
def cmd2(global_arg, foo_bar):
    print("this is the second command with arg: %s" % foo_bar)

if __name__ == '__main__':
    main(arg('--global-arg', help="a global arg applied to all commands"))
```

4.1.2 Compiling arguments

If you have many arguments it may get cumbersome to list all the arguments in the decorator and the function:

```
@command('cmd1',
    arg('--first', '-f'),
    arg('--second', '-s'),
    arg('--third', '-t'),
    arg('--fourth', '-F'),
)
def cmd1(first, second, third, fourth):
    pass
```

Think of having many of such functions maybe even repeating some arguments. Then it becomes handy to use compiled arguments:

```
from argdeco import main, command

@command('cmd',
    arg('--first', '-f'),
    arg('--second', '-s'),
    arg('--third', '-t'),
    arg('--fourth', '-F'),
)
def cmd1(opts):
    if opts['first'] == '1':
        ...

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

`compile` can have following values:

Value	Alias	Description
None	'kwargs'	Passed to handler as keyword arguments
True	'dict'	Args passed to handler as single dictionary
'args'		Pass args namespace as returned from `argparse.ArgumentParser.parse_args()`
function		You can also pass a function, which is explained in *Compile functions*

Compile to args:

```
from argdeco import main, command

@command('cmd',
```

(continues on next page)

(continued from previous page)

```
    arg('--first', '-f'),
    arg('--second', '-s'),
    arg('--third', '-t'),
    arg('--fourth', '-F'),
)
def cmd1(args):
    if args.first == '1':
        ...

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

Compile functions

If you need even more control of your arguments, you can pass custom compile functions, which gets args namespace and opts keyword arguments as parameter and is expected to return:

type	description
dict	This will be passed as keyword arguments to handler function
tuple/list	A tuple with two values, a list (or tuple) and a dictionary, which are passed as args and kwargs to handler.
list/tuple	If the tuple or list does not match requirements in above, it is assumed, that no kwargs shall be passe and this is the args list for positional parameters.

You can use such a function work preprocessing some args and manipulate the parameters passed to the handlers.

Compiler factory

Compile functions are usually not directly connected with command decorator and usually do not know about it (unless you share it globally). If you need to access data from command decorator instance or need for other reasons more control of argument setup, you can use a compiler factory.

A compiler factory is initialized with `CommandDecorator` instance.

It must return a function, which will get args as returned from `argparse.ArgumentParser.parse_args()` and keyword arguments.

Here you see the most simplest one:

```
def my_factory(command):
    def my_compiler(args, **opts):
        return opts

    return my_compiler
```

4.1.3 Validating and transforming arguments

With `argparse.Action` `argparse` module provides a method to provide custom argument handlers. *`argdeco`* provides some eases for this as well:

```
from argdeco import arg, command, main
import dateutil.parser

@arg('--date', '-d')
def arg_date(value):
    return dateutil.parser.parse(value)

@main(arg_date)
def handle_date(date)
    print(date.isoformat())

main()
```

There is also a complex (more powerful) way, which is

```
import dateutil.parser

from argdeco import arg, command, main

@arg("-d", "--date", help="pass some date")
def arg_date(self, parser, namespace, values, option_string=None):
     # here we can do some validations
     print "self: %s" % self
     setattr(namespace, self.dest, dateutil.parser.parse(values))

@command("check_date", arg_date)
def check_date(date):
    print(date)

main()
```

4.1.4 Working with subcommands

You may want to implement a CLI like git has. This is quite easy with *argdeco*:

```
from argdeco import main, command, arg, opt
from textwrap import dedent

# we will implement a sample `remote` command here

# global arguments (for all commands)
main(
    arg('--config-file', '-C', help="pass a config file"),
)

# create a new decorator for sub-command 'remote' actions
remote_command = command.add_subcommands('remote',
    help="manage remote sites",  # description in global command list
    subcommands = dict(
        title = "remote commands",   # caption of subcommand list
        description = dedent('''                Here is some documentation about␣
↪remote commands.

            There is a lot to say ...
        ''')
    )
```

(continues on next page)

(continued from previous page)

```
)

@remote_command('add',
    arg('remote_name', help="name of remote site"),
    arg('url', help="url of remote site"),
    opt('--tags', help="get all tags when requesting remote site"),
)
def cmd_remote_add(config_file, remote_name, url, tags):
    ...

@remote_command('rename',
    arg('old_name', help="old name of remote"),
    arg('new_name', help="new name of remote"),
)
def cmd_remote_rename(config_file, old_name, new_name):
    ...
```

If you run `add_subcommands(..., subcommands={...})`, all the keyword arguments of add_subcommands, except the `subcommands` one, will be passed to `argparse.ArgumentParser.add_parser()` and the subcommands dictionary will be passed as keyword arguments to `argparse.ArgumentParser.add_subparsers()`.

4.2 main

argdeco.main – the main function

This module provides *Main*, which can be used to create main functions.

For ease it provides common arguments like *debug*, *verbosity* and *quiet* which control whether you want to print stacktraces, and how verbose the logging is. These arguments will not be passed to command handlers or main function handler.

Usually you will import the global main instance provided in argdeco:

```
from argdeco import main
```

In this case, *main.command* is also provided as global symbol:

```
from argdeco import main, command

@main.command(...)
def cmd(...):
    ...

# is equivalent to
@command(...)
def cmd(...):
    ...
```

Bug you can also create an own instance:

```
from argdeco.main import Main
main = Main()

@main.command('foo', ...)
```

(continues on next page)

(continued from previous page)

```
def my_cmd(...):
    ...
```

If you want to make use of the predefined (global) args:

```
if __name__ == '__main__':
    main(verbosity=True, debug=True, quiet=True)
```

class argdeco.main.**Main**(*debug=False*, *verbosity=False*, *quiet=False*, *compile=None*, *compiler_factory=None*, *command=None*, *log_format='%(name)-20.20s %(levelname)-10.10s %(message)s'*, *error_handler=<built-in function exit>*, *error_code=1*, *catch_exceptions=(<type 'exceptions.SystemError'>, <type 'exceptions.AssertionError'>, <class 'argdeco.main.ArgParseExit'>)*, ***kwargs*)

Main function provider

An instance of this class can be used as main function for your program. It provides a :py:attribute:

Parameters

- **debug** – Set True if you want main to manage the debug arg. (default: False).

  Set global logging levels to DEBUG and print out full exception stack traces.

- **verbosity** – Control global logging log levels (default: False)

  If this is turned on, default log level will be set to `ERROR` and following argument are provided:

  -v, --verbose
  set log level to level `WARNING`

  -vv, -v `-v`**, --verbose** `--verbose`
  Set log level to level `INFO`

  -vvv, -v `-v -v`
  Set log level to level `DEBUG`

- **quiet** – If you set this to `True`, argument *`--quiet`* will be added:

  --quiet
  If this option is passed, global log level will be set to `CRITICAL`.

- **command** – CommandDecorator instance to use. This defaults to None, and for each main instance there will be created a *`CommandDecorator`* instance.

- **compile** – This parameter is passed *`CommandDecorator`* instance and controls, if arguments passed to handlers are compiled in some way.

- **compiler_factory** – This parameter is passed *`CommandDecorator`* instance and defines a factory function, which returns a compile function.

  You may either use `compile` or `compiler_factory`.

- **log_format** – This parameter is passed to `logging.basicConfig()` to define log output. (default: `"%(name)s %(levelname)s %(message)s"`)

- **error_code** – This is the error code to be returned on an exception (default: 1).

- **error_handler** – Pass a function, which handles errors. This function will get the error code returned from a command (or main) function and do something with it. Default is `sys.exit()`.

If you do not want to exit the program after running the main funtion you have to set `error_handler` to `None`.

If you want to access the managed arguments (quiet, verbosity, debug), you can access them as attributes of the main instance:

```
if not main.quiet:
    print("be loud")

if main.debug:
    print("debug is on")
```

__call__(**args*, ***kwargs*)

You can call *Main* instance in various ways. As function or as decorator. As long you did not have decorated a function with this *Main* instance, you can invoke it as function for confiugration.

As soon there is defined some action, invoking the instance, will execute the actions.

Configure some global arguments:

```
main(
    arg('--global', '-g', help="a global argument"),
)
```

Decorate a function to be called as main function:

```
@main
def my_main():
    return 0

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

Decorate a function to be main function and define arguments of it:

```
@main(
    arg('--first', '-f', help="first argument"),
    arg('--second', '-s', help="second argument"),
)
def main(first, second):
    return 0    # successful

if __name__ == "__main__":
    main(debug=True)
```

Parameters

- ***args** – All arguments of type `arg` are filtered out and added as global argument to underlying *CommandDecorator* instance.

  All other arguments are collected – if any to be *argv*. If there are any other parameters, this function switches into regular *main* mode and will execute the main function passing *argv*. If there are no arguments defined, `sys.argv` is used as default.

- ****kwargs** – You can pass various keyword arguments to tweak behaviour of the main function.

  argv You can set explicitly the `argv` vector. This becomes handy, if you want to pass an empty `argv` list and do not want to use the default sys.argv.

debug Turn on debug argument, see *Main* for more info.

verbosity Turn on verbose argument, see *Main* for more info.

quiet Turn on quiet argument, see *Main* for more info.

error_handler Tweak the error handler. This will be only local to this call.

compile Set compile for this call.

compiler_factory Set compiler_factory for this call.

Returns

Decorator mode Returns the instance itself, to be invoked as decorator.

Run mode Returns whatever error_handler returns, when getting the return value of the invoked action function

add_arguments(**args*)

Explicitely add arguments:

```
main.add_arguments( arg('--first'), arg('--second') )
```

This function wraps `argdeco.command_decorator.C()`

Parameters ***args** – arguments to be added.

configure(*debug=None*, *quiet=None*, *verbosity=None*, *traceback=None*, *compile=None*, *compiler_factory=None*, *catch_exceptions=None*, ***kwargs*)

configure behaviour of main, e.g. managed args

install_bash_completion(*script_name=None*, *dest='~/.bashrc'*)

add line to activate bash_completion for given script_name into dest

You can use this for letting the user install bash_completion:

```
from argdeco import command, main

@command("install-bash-completion",
    arg('--dest', help="destination", default="~/.bashrc")
)
def install_bash_completion(dest):
    main.install_bash_completion(dest=dest)
```

uninstall_bash_completion(*script_name=None*, *dest='~/.bashrc'*)

remove line to activate bash_completion for given script_name from given dest

You can use this for letting the user uninstall bash_completion:

```
from argdeco import command, main

@command("uninstall-bash-completion",
    arg('--dest', help="destination", default="~/.bashrc")
)
def uninstall_bash_completion(dest):
    main.uninstall_bash_completion(dest=dest)
```

4.3 command_decorator

class argdeco.command_decorator.**CommandDecorator**(**args*, ***kwargs*)
Create a decorator to decorate functions with their arguments.

add_command(*command*, **args*, ***kwargs*)
add a command.

This is basically a wrapper for add_parser()

add_subcommands(*command*, **args*, ***kwargs*)
add subcommands.

If command already defined, pass args and kwargs to add_subparsers() method, else to add_parser() method. This behaviour is for convenience, because I mostly use the sequence:

```
>>> p = parser.add_parser('foo', help="some help")
>>> subparser = p.add_subparsers()
```

If you want to configure your sub_parsers, you can do it with:

```
>>> command.add_subcommands('cmd',
        help = "cmd help"
        subcommands = dict(
            title = "title"
            description = "subcommands description"
        )
    )
```

execute(*argv=None*, *compile=None*, *preprocessor=None*, *compiler_factory=None*)
Parse arguments and execute decorated function

argv: list of arguments compile:

- None, pass args as keyword args to function
- True, pass args as single dictionary
- function, get args from parse_args() and return a pair of tuple and dict to be passed as args and kwargs to function

get_config_name(*action*, *name=None*)
get the name for configuration

This returns a name respecting commands and subcommands. So if you have a command name "index" with subcommand "ls", which has option "–all", you will pass the action for subcommand "ls" and the options's dest name ("all" in this case), then this function will return "index.ls.all" as configuration name for this option.

update(*command=None*, ***kwargs*)
update data, which is usually passed in ArgumentParser initialization

e.g. command.update(prog="foo")

exception argdeco.command_decorator.**NoAction**

4.4 argdeco.arguments

argdeco.arguments – manage arguments

class argdeco.arguments.**ArgAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Internal class to handle argument actions

There are two ways

class argdeco.arguments.**arg**(**args*, ***opts*)

Represent arguments passed with add_argument() to an argparser

See https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser.add_argument

class argdeco.arguments.**group**(**args*, ***opts*)

Argument group

This class is a wrapper for `argparse.ArgumentParser.add_argument_group()`.

Usage:

```
@main(
    group(
        arg('--first'),
        arg('--second'),
        title="group title",
        description='''
           Here some group description
        '''
    )
)
def _main(first, second):
    pass
```

class argdeco.arguments.**mutually_exclusive**(**args*, ***opts*)

Mutually exclusive argument group

Usage:

```
@main(
    mutually_exclusive(
        arg('--first'),
        arg('--second'),
        title="group title",
        description='''
           Here some group description
        '''
    )
)
def _main(first, second):
    pass
```

class argdeco.arguments.**opt**(**args*, ***opts*)

Option action="store_true"

4.5 Working with configurations

Working with configurations

A common pattern making use of configuration files:

```
from argdeco import main, command, arg, opt, config_factory
form os.path import expanduser

main.configure(compiler_factory=config_factory(
    config_file=arg('--config-file', '-C', help="configuration file",␣
↪default=expanduser('~/.config/myconfig.yaml'))
))

@command('ls', opt('--all'))
def mycmd(cfg):
    if cfg['ls.all']:
        pass

main()
```

If you want to have `foo.bar` expanded to `{'foo': {'bar': ...}}`, use following:

```
from argdeco import main, command, arg, opt, config_factory, ConfigDict
form os.path import expanduser

main.configure(compiler_factory=config_factory(ConfigDict,
    config_file=arg('--config-file', '-C', help="configuration file",␣
↪default=expanduser('~/.config/myconfig.yaml'))
))

@command('ls', opt('--all'))
def mycmd(cfg):
    if cfg['ls']['all']:
        pass

main()
```

class argdeco.config.**ConfigDict**(*E=None*, ***F*)

dictionary-like class

This class implements a dictionary, which creates deep objects from keys like "foo.bar". Example:

```
>>> c = Config()
>>> c['foo.bar'] = 'x'
>>> c
{'foo': {'bar': 'x'}}
>>> c['foo.bar']
'x'
```

assimilate(*value*)

If value is a dictionary, then make it beeing a dictionary of same class like this. Copy all attributes, which are not controlled by dict class

flatten(*D*)

flatten a nested dictionary D to a flat dictionary

nested keys are separated by '.'

update(*E=None*, ***F*)

flatten nested dictionaries to update pathwise

```
>>> Config({'foo': {'bar': 'glork'}}).update({'foo': {'blub': 'bla'}})
{'foo': {'bar': 'glork', 'blub': 'bla'}
```

In contrast to:

```
>>> {'foo': {'bar': 'glork'}}.update({'foo': {'blub': 'bla'}})
{'foo: {'blub': 'bla'}'}
```

argdeco.config.**config_factory**(*ConfigClass=<type 'dict'>*, *prefix=None*, *config_file=None*)
return a class, which implements the compiler_factory API

Parameters

- **ConfigClass** – defaults to dict. A simple factory (without parameter) for a dictionary-like object, which implements __setitem__() method.

  Additionally you can implement following methods:

  init_args A method to be called to initialize the config object by passing `Namespace` object resulting from `parseargs` method.

  You could load data from a configuration file here.

  compile_args A method, which can return the same like a `compile` function does. If there is no such method, a tuple with a ConfigClass instance as single element is returned.

- **prefix** – Add this prefix to config_name. (e.g. if prefix="foo" and you have config_name="x.y" final config_path results in "foo.x.y")

- **config_file** (`argdeco.arguments.arg`) – An *arg* to provide a config file.

  If you provide this argument, you can implement one of the following methods in your `ConfigClass` to load data from the configfile:

  load If you pass `config_file` argument, this method can be implemented to load configuration data from resulting stream.

  If config_file is '-', stdin stream is passed.

  load_from_file If you prefer to open the file yourself, you can do this, by implementing `load_from_file` instead which has the filename as its single argument.

  update method like `dict.update()`. If neither of `load` or `load_from_file` is present, but `update` is, it is assumed, that config_file is of type YAML (or JSON) and configuration is updated by calling `update` with the parsed data as parameter.

  If you implement neither of these, it is assumed, that configuration file is of type YAML (or plain JSON, as YAML is a superset of it).

  Data is loaded from file and will update configuration object using dict-like `dict.update()` method.

Returns ConfigFactory class, which implements compiler_factory API.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

C

E

F

G

I

M

N

O

U