
clize Documentation

Release 3.0.1

Yann Kaiser

September 04, 2016

1	About Clize	3
2	Getting started	11
3	Guides	25
4	Reference	35
5	Project documentation	61
	Python Module Index	65

Clize is an argument parser for [Python](#). It focuses on minimizing the effort required to create them:

- Command-line interfaces are created by passing functions to `clize.run`.
- Parameter types are deduced from the functions' parameters.
- A `--help` message is generated from your docstrings. Seriously, why does this still need to be a bullet point?
- Decorators can be used to reuse functionality across functions.
- Clize can be extended with new parameter behavior.

Here's an example:

```
from sigtools.modifiers import kwoargs
from clize import run

@kwoargs('no_capitalize') # turns no_capitalize into a keyword-only parameter
                          # on Python 2
def hello_world(name=None, no_capitalize=False):
    """Greets the world or the given name.

    name: If specified, only greet this person.

    no_capitalize: Don't capitalize the given name.
    """
    if name:
        if not no_capitalize:
            name = name.title()
        return 'Hello {0}!'.format(name)
    return 'Hello world!'

if __name__ == '__main__':
    run(hello_world)
```

`run` takes the function and automatically produces a command-line interface for it:

```
$ pip install --user clize
$ python examples/hello.py --help
Usage: examples/hello.py [OPTIONS] [name]

Greets the world or the given name.

Positional arguments:
  name                If specified, only greet this person.

Options:
  --no-capitalize     Don't capitalize the give name.

Other actions:
  -h, --help          Show the help
$ python hello.py
Hello world!
$ python hello.py john
Hello John!
$ python hello.py dave --no-capitalize
Hello dave!
```

Interested?

- Follow the *[tutorial](#)*
- Browse the *[examples](#)*
- Ask for help on *[Gitter](#)*
- Check out *[why Clize was made](#)*
- Star, watch or fork *[Clize](#)* on *[GitHub](#)*

About Clize

1.1 Why Clize was made

Clize started from the idea that other argument parsers were too complicated to use. Even for a small script, one would have to use a fairly odd interface to first generate a « parser » object with the correct behavior then use it.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("name", nargs='?',
                    help="The person to greet")
parser.add_argument("repeat", nargs='?', type=int, default=1,
                    help="How many times should the person be greeted")
parser.add_argument("--no-capitalize", action='store_const', const=True)

args = parser.parse_args()

if args.name is None:
    greeting = 'Hello world!'
else:
    name = args.name if args.no_capitalize else args.name.title()
    greeting = 'Hello {0}!'.format(name)

for _ in range(args.repeat):
    print(greeting)
```

This doesn't really feel like what Python should be about. It's too much for such a simple CLI.

A much simpler alternative could be to use argument unpacking:

```
import sys.argv

def main(script, name=None, repeat='1'):
    if name is None:
        greeting = 'Hello world!'
    else:
        greeting = 'Hello {0}!'.format(name.title())

    for _ in range(int(repeat)):
        print(greeting)

main(*sys.argv)
```

This makes better use of Python concepts: a function bundles a series of statements with a list of parameters, and that bundle is now accessible from the command-line.

However, we lose several features in this process: Our simpler version can't process named arguments like `--no-capitalize`, there is no `--help` function of any sort, and all errors just come up as tracebacks, which would be confusing for the uninitiated.

Those shortcomings are not inherent to bundling behavior and parameters into functions. Functions can have keyword-only parameters (and this can be backported to Python 2.x), and those parameter lists can be examined at run time. Specific errors can be reformatted, and so forth.

Clize was made to address these shortcomings while expanding on this idea that command-line parameters are analogous to those of a function call in Python.

The following table summarizes a few direct translations Clize makes:

Python construct	Command-line equivalent
Function	Command
List of functions	Multiple commands
Docstring	Source for the <code>--help</code> output
Decorator	Transformation of a command
Positional parameter	Positional parameter
Keyword-only parameter	Named parameter (like <code>--one</code>)
Parameter with a default value	Optional parameter
Parameter with <code>False</code> as default	Flag (<code>True</code> if present, <code>False</code> otherwise)

Some concepts fall outside these straightforward relationships, but in all cases your part of the command-line interface remains a normal function. You can call that function normally, have another command from your CLI use it, or test it like any other function.

For when Python constructs aren't enough, Clize uses parameter annotations, a yet mostly unexplored feature of Python 3. For instance, you can specify value converters for the received arguments, or replace how a parameter is implemented completely.

Even though its basic purpose could be called *magicky*, Clize attempts to limit magic in the sense that anything Clize's own parameters do can also be done in custom parameters. For instance, `--help` in a command-line will trigger the displaying of the help screen, even if there were errors beforehand. You might never need to do this, but the option is there if and when you ever need this. `argparse` for instance does not let you do this.

With Clize, you start simple but you remain flexible throughout, with options for refactoring and extending your command-line interface.

1.1.1 “Why not use an existing parser instead of making your own?”

Argument parsing is a rather common need for Python programs. As such, there are many argument parsers in existence, including no less than three in the standard library alone!

The general answer is that they are different. The fact that there are so many different parsers available shows that argument parsing APIs are far from being a “solved problem”. Clize offers a very different approach from those of `argparse`, `Click` or `docopt`. Each of these always have you write a specification for your CLI.

Clize comes with less batteries included than the above. It focuses on providing just the behavior that corresponds with Python parameters, along with just a few extras. Clize can afford to do this because unlike these, Clize can be extended to accomodate custom needs. Every parameter kind can be implemented by external code and made usable the same way as `clize.parameters.multi` or `clize.parameters.argument_decorator`.

1.1.2 “Why not create a thin wrapper around argparse?”

Back during Clize’s first release, `argparse`’s parser would have been sufficient for what Clize proposed, though I wasn’t really interested in dealing with it at the time. With Clize 3’s extensible parser, replacing it with `argparse` would be a loss in flexibility, in parameter capabilities and help message formatting.

1.2 An illustrated history

This document recounts the story of each clize version. Here’s the short version first:

Version	Release date	Major changes
1.0b	April 4th 2011	First release
2.0	October 7th 2012	Subcommands, Python 3 syntax support
2.2	August 31st 2013	Minor additions
2.4	October 2nd 2013	Bugfixes
3.0	May 13th 2015	Extensibility, decorators, focus on py3

You can also browse the [release notes](#).

And here’s the story:

1.2.1 Before Clize

After having used `optparse` and `twisted`’s `usage.Options` in various medium-sized projects, and viewing `argparse` as being more of the same, I wondered if I needed all this when writing a trivial program. I realized that if I only wanted to read some positional arguments, I could just use tuple unpacking on `sys.argv`:

```
from __future__ import print_function

import sys

script, first, last = sys.argv
print('Hello', first, last)
```

If you used argument packing in a function call instead, you gain the ability to make use of default values:

```
from __future__ import print_function

import sys

def greet(script, first, last, greeting="Hello"):
    print(greeting, first, last)

greet(*sys.argv)
```

It works nicely save for the fact that you can’t request a help page from it or have named options. So I set out to add those capabilities while doing my best to keep it as simple as possible, like the above example.

1.2.2 1.0: A first release

```
from __future__ import print_function

from clize import clize, run


@clize
def greet(first, last, greeting="Hello"):
    print(greeting, first, last)

run(greet)
```

Thanks to the ability in Python to look at a function's signature, you gained a `--help` page, and `greeting` was available as `--greeting` on the command line, while adding just one line of code. This was very different from what `argparse` had to offer. It allowed you to almost completely ignore argument parsing and just write your program's logic as a function, with your parameters' documented in the docstring.

In a way, Clize had opinions about what a CLI should and shouldn't be like. For instance, it was impossible for named parameters to be required. It was generally very rigid, which was fine given its purpose of serving smaller programs.

It hadn't visibly garnered much interest. I was still a user myself, and no other argument parser had really interested me, so I kept using it and watched out for possible improvements. Aside from the subcommand dispatcher, there was little user feedback so the inspiration ended up coming from somewhere else.

1.2.3 2.0: Function annotations

Clize 2.0 came out with two major features. *Subcommands* and a new way of specifying additional information on the parameters. I'll skip over subcommands because they are already a well established concept in argument parsing. See *Multiple commands* for their documentation.

Through now forgotten circumstances, I came across [PEP 3107](#) implemented since Python 3.0, which proposed a syntax for adding information about parameters.

Up until then, if you wanted to add an alias to a named parameter, it looked a bit like this:

```
from __future__ import print_function

from clize import clize, run


@clize(require_excess=True, aliases={'reverse': ['r']})
def echo(reverse=False, *args):
    text = ' '.join(args)
    if reverse:
        text = text[::-1]
    print(text)

run(echo)
```

Many things involved passing parameters in the decorator. It was generally quite ugly, especially when more than one parameter needed adjusting, at which point the decorator call grew to the point of needing to be split over multiple lines.

The parameter annotation syntax from [PEP 3107](#) was fit to replace this. You could tag the parameter directly with the alias or conversion function or whatever. It involved looking at the type of each annotation, but it was a lot more

practical than spelling *alias*, *converter* and the parameter's name all over the place.

It also allowed for keyword-only parameters from [PEP 3102](#) to map directly to named parameters while others would always be positional parameters.

```
from __future__ import print_function

from clize import clize, run

@clize(require_excess=True)
def echo(*args, reverse='r'=False):
    text = ' '.join(args)
    if reverse:
        text = text[::-1]
    print(text)

run(echo)
```

Python 3 wasn't quite there yet, so these were just features on the side at the time. I liked it a lot however and used it whenever I could, but had to use the older interface whenever I had to use Python 2.

1.2.4 3.0: The rewrite

Python 3.3 introduced `inspect.signature`, an alternative to the rough `inspect.getfullargspec`. This provided an opportunity to start again from scratch to build something on a solid yet flexible base.

For versions of Python below 3.3, a backport of `inspect.signature` existed on PyPI. This inspired a Python 3-first approach: The old interface was deprecated in favor of the one described just above.

```
from clize import run, parameter

def echo(*args: parameter.required, reverse='r'=False):
    text = ' '.join(args)
    if reverse:
        text = text[::-1]
    print(text)

run(echo)
```

Since the `@clize` decorator is gone, `echo` is now just a regular function that could theoretically be used in non-cli code or tests.

Users looking to keep Python 2 compatibility would have to use a compability layer for keyword-only parameters and annotations: `sigtools.modifiers`.

```
from __future__ import print_function

from sigtools import modifiers
from clize import run, parameter

@modifiers.autokwargs
@modifiers.annotate(args=parameter.REQUIRED, reverse='r')
def echo(reverse=False, *args):
    text = ' '.join(args)
    if reverse:
        text = text[::-1]
    print(text)
```

```
run(echo)
```

`sigtools` was created specifically because of Clize, but it aims to be a generic library for manipulating function signatures. Because of Clize’s reliance on accurate introspection data on functions and callables in general, `sigtools` also provided tools to fill the gap when `inspect.signature` stumbles.

For instance, when a decorator replaces a function and complements its parameters, `inspect.signature` would only produce something like `(spam, *args, ham, **kwargs)` when Clize would need more information about what `*args` and `**kwargs` mean.

`sigtools` thus provided decorators such as `forwards` and the higher-level `wrapper_decorator` for specifying what these parameters meant. This allowed for *creating decorators for CLI functions* in a way analogous to regular decorators, which was up until then something other introspection-based tools had never done. It greatly improved Clize’s usefulness with multiple commands.

With the parser being completely rewritten, a large part of the argument parsing was moved away from the monolithic “iterate over `sys.argv`” loop to one that deferred much of the behaviour to parameter objects determined from the function signature. This allows for library and application authors to almost completely *customize how their parameters work*, including things like replicating `--help`’s behavior of working even if there are errors beforehand, or other completely bizarre stuff.

This is a departure from Clize’s opiniated beginnings, but the defaults remain sane and it usually takes someone to create new `Parameter` subclasses for bizarre stuff to be made. In return Clize gained a flexibility few other argument parsers offer.

1.3 Frequently asked questions

Drats! These docs are all new and the FAQ is almost empty! Help me improve this dire situation by *contacting me* with your questions!

Table of contents

- *What versions of Python are supported?*
- *What libraries are required for Clize to run?*
- *I just installed Clize using `pip` and I still get `ImportErrors`*
- *What is `sigtools` and why is it a separate library?*
- *What other libraries can be used for argument parsing?*
- *How can I write mutually exclusive flags?*
- *Some of my commands share features, can I reuse code somehow?*
- *Where can I find more help?*
- *Contacting the author*

1.3.1 What versions of Python are supported?

Clize is tested to run succesfully on Python 2.6, 2.7, 3.2, 3.3 and 3.4.

1.3.2 What libraries are required for Clize to run?

Using `pip` to install Clize from PyPI as in *Installation* will automatically install the right dependencies for you.

If you still need the list, Clize always requires:

- `six`: For helping run Clize on both Python 2 and 3.
- `sigtools`: Utilities to help manipulate function signatures.

If you wish to use `clize.converters.datetime`, you need:

- `python-dateutil`: For parsing dates.

`pip` will install `dateutil` if you specify to install Clize with the `datetime` option, i.e. `pip install "clize[datetime]"`.

On Python 2, `sigtools` requires:

- `funcsigs`: A backport of `inspect.signature` to Python 2.

Finally, on Python 2.6, this is also needed:

- `ordereddict`: A backport of `collections.OrderedDict`.

1.3.3 I just installed Clize using `pip` and I still get `ImportErrors`

Old versions of `pip` do not read Python-version dependent requirements and therefore do not install `funcsigs` or `ordereddict`. To remedy this, you can:

- Upgrade `pip` and *install Clize* again. (Use the `-U` flag of `pip install` to force a reinstall.)
- Install the *dependencies* manually.

1.3.4 What is `sigtools` and why is it a separate library?

`sigtools` is used in many of the examples throughout this documentation, and it is maintained by the same person as Clize, thus the above question.

Clize's purpose is twofold:

- Convert the idioms of a function signature into those of a CLI,
- Parse the input that CLI arguments are.

It turns out that just asking for the function signature from `inspect.signature` is not good enough:

- Python 2 users cannot write keyword-only parameters.
- `inspect.signature` cannot process decorators that return a function with slightly altered parameters.

For the first point, Clize could have accepted an argument that said “do as if that parameter was keyword-only and make it a named parameter on the CLI” (and in fact it used to), but that would have Clize behave according to a signature *and a bunch of things around it*, which is a concept it tries to steer away from.

For the second, some tooling would be necessary to specify how exactly a decorator affected a wrapped function's parameters.

Modifying and making signatures more useful was both complex and independent from command-line argument parsing, so it was made a separate library as `sigtools`.

So there you have it, `sigtools` helps you add keyword-only parameters on Python 2, and helps decorators specify how they alter parameters on decorated functions. All Clize sees is the finished accurate signature from which it infers a CLI.

1.3.5 What other libraries can be used for argument parsing?

Similar to Clize in that parameter behavior is inferred from a function signature:

- **Baker**: Similar to Clize 2, with a decorator-based approach for adding commands.
- **argh**: Based on `argparse`, optionally lets you add parameters using a decorator API similar to `argparse`.

Others:

- `argparse` from the standard library.
- **doctopt**: Generates a CLI parser based on the `--help` page you give it.
- **click**: `argparse`-like specifications as decorators, powerful command nesting tools
- `twisted.python.usage`
- `optparse` and `getopt` from the standard library, both deprecated.
- Many, many more.

1.3.6 How can I write mutually exclusive flags?

Mutually exclusive flags refer to when a user can use one flag A (`--flag-a`) or the other (`--flag-b`), but not both at the same time.

It is a feature that is difficult to express in a function signature as well as on the `--help` screen for the user (other than in the full usage form). It is therefore recommended to use a positional parameter or option that accepts one of specific values. `one_of` can help you do that.

If you still think mutually exclusive parameters are your best option, you can check for the condition in your function and raise `clize.ArgumentError`, as in the *Arbitrary requirements* part of the tutorial.

1.3.7 Some of my commands share features, can I reuse code somehow?

Yes! You can use decorators much like in regular Python code, see *Function compositing*.

1.3.8 Where can I find more help?

You can get help by *contacting me directly*, writing in the dedicated Gitter chatroom, using the `#clize` `#python` hashtags on Twitter, or by posting in the Clize Google+ community.

1.3.9 Contacting the author

You can contact me via `@YannKsr` on Twitter or via `email`. Feel free to ask about Clize!

Getting started

This section contains tutorials for the most commonly used features of Clize.

2.1 Basics

2.1.1 Installation

You can install clize using `pip`. If in an activated virtualenv, type:

```
pip install clize
```

If you wish to do a user-wide install:

```
pip install --user clize
```

2.1.2 A minimal application

A minimal command-line application written with clize consists of writing a function and passing it to `run()`:

```
from clize import run

def hello_world():
    return "Hello world!"

if __name__ == '__main__':
    run(hello_world)
```

If you save this as `helloworld.py` and run it, the function will be run:

```
$ python helloworld.py
Hello world!
```

In this example, `run()` simply takes our function, runs it and prints the result.

2.1.3 Requiring arguments

You can require arguments the same way as you would in any Python function definition. To illustrate, let's write an `echo` command.

```
#!/usr/bin/env python
from clize import run

def echo(word):
    return word

if __name__ == '__main__':
    run(echo)
```

Save it as *echo.py* and run it. You will notice the script requires exactly one argument now:

```
$ ./echo.py
./echo.py: Missing required arguments: word
Usage: ./echo.py [OPTIONS] word
```

```
$ ./echo.py ham
ham
```

```
$ ./echo.py ham spam
./echo.py: Received extra arguments: spam
Usage: ./echo.py [OPTIONS] word
```

2.1.4 Enhancing the `--help` message

If you try to specify `--help` when running either of the previous examples, you will notice that Clize has in fact also generated a `--help` feature for you already:

```
$ ./echo.py --help
Usage: ./echo.py [OPTIONS] word

Positional arguments:
  word

Other actions:
  -h, --help  Show the help
```

It is fairly unhelpful right now, so we should improve that by giving our function a docstring:

```
def echo(word):
    """Echoes word back

    word: One word or quoted string to echo back
    """
    return word
```

As you would expect, it translates to this:

```
$ ./echo.py --help
Usage: ./echo.py [OPTIONS] word

Echoes word back

Positional arguments:
  word  One word or quoted string to echo back

Other actions:
  -h, --help  Show the help
```

See also:*Customizing the help using the docstring*

2.1.5 Accepting options

Clize will treat any regular parameter of your function as a positional parameter of the resulting command. To specify an option to be passed by name, you will need to use keyword-only parameters.

Let's add a pair of options to specify a prefix and suffix around each line of word:

```
from sigtools.modifiers import autokwargs

@autokwargs
def echo(word, prefix='', suffix=''):
    """Echoes text back

    word: One word or quoted string to echo back

    prefix: Prepend this to each line in word

    suffix: Append this to each line in word
    """
    if prefix or suffix:
        return '\n'.join(prefix + line + suffix
                           for line in word.split('\n'))
    return word
```

Here, `autokwargs` turns all parameters with defaults into keyword-only parameters. Clize then treats it as an option instead of a positional parameter.

See also:*Python 2 support for named parameters*

Note: Since Python does not have syntax for specifying keyword-only parameters until Python 3, we will use decorators from the `sigtools.modifiers` module to emulate them in all Python versions supported by Clize.

If you don't need to worry about Python 2 compatibility, you can use the Python 3 features directly:

```
def echo(word, *, prefix='', suffix=''):
    pass # ...
```

The change reflects to the command and its help when run:

```
$ ./echo.py --prefix x: --suffix :y 'spam'
$ ham'
x:spam:y
x:ham:y
```

```
$ ./echo.py --help
Usage: ./echo.py [OPTIONS] word

Echoes text back

Positional arguments:
  word    One word or quoted string to echo back
```

```
Options:
  --prefix=STR    Prepend this to each line in word(default: )
  --suffix=STR    Append this to each line in word(default: )

Other actions:
  -h, --help      Show the help
```

2.1.6 Collecting all positional arguments

Just like when defining a regular Python function, you can prefix a parameter with one asterisk and it will collect all remaining positional arguments:

```
@autokwargs
def echo(prefix='', suffix='', *text):
    pass # ...
```

However, just like in Python, this makes the parameter optional. To require that it should receive at least one argument, you will have to tell Clize that `text` is required using an annotation:

```
#!/usr/bin/env python
from sigtools.modifiers import annotate, autokwargs
from clize import Parameter, run

@annotate(text=Parameter.REQUIRED)
@autokwargs
def echo(prefix='', suffix='', *text):
    """Echoes text back

    text: The text to echo back

    prefix: Prepend this to each line in word

    suffix: Append this to each line in word
    """
    text = ' '.join(text)
    if prefix or suffix:
        return '\n'.join(prefix + line + suffix
                          for line in text.split('\n'))
    return text

if __name__ == '__main__':
    run(echo)
```

2.1.7 Accepting flags

Parameters which default to `False` are treated as flags. Let's add a flag to reverse the input:

```
@annotate(text=Parameter.REQUIRED)
@autokwargs
def echo(prefix='', suffix='', reverse=False, *text):
    """Echoes text back

    text: The text to echo back

    reverse: Reverse text before processing
```

```

prefix: Prepend this to each line in word

suffix: Append this to each line in word

"""
text = ' '.join(text)
if reverse:
    text = text[::-1]
if prefix or suffix:
    return '\n'.join(prefix + line + suffix
                      for line in text.split('\n'))
return text

```

```

$ ./echo.py --reverse hello world
dlrow olleh

```

2.1.8 Converting arguments

Clize automatically tries to convert arguments to the type of the receiving parameter's default value. So if you specify an integer as default value, Clize will always give you an integer:

```

@annotate(text=Parameter.REQUIRED)
@autokwargs
def echo(prefix='', suffix='', reverse=False, repeat=1, *text):
    """Echoes text back

    text: The text to echo back

    reverse: Reverse text before processing

    repeat: Amount of times to repeat text

    prefix: Prepend this to each line in word

    suffix: Append this to each line in word

    """
    text = ' '.join(text)
    if reverse:
        text = text[::-1]
    text = text * repeat
    if prefix or suffix:
        return '\n'.join(prefix + line + suffix
                          for line in text.split('\n'))
    return text

```

```

$ ./echo.py --repeat 3 spam
spamspamspam

```

2.1.9 Aliasing options

Now what we have a bunch of options, it would be helpful to have shorter names for them. You can specify aliases for them by annotating the corresponding parameter:

```
@annotate(text=Parameter.REQUIRED,
          prefix='p', suffix='s', reverse='r', repeat='n')
@autokwargs
def echo(prefix='', suffix='', reverse=False, repeat=1, *text):
    pass # ...
```

```
$ ./echo.py --help
Usage: ./echo.py [OPTIONS] text...

Echoes text back

Positional arguments:
  text    The text to echo back

Options:
  -r, --reverse      Reverse text before processing
  -n, --repeat=INT   Amount of times to repeat text(default: 1)
  -p, --prefix=STR   Prepend this to each line in word(default: )
  -s, --suffix=STR   Append this to each line in word(default: )

Other actions:
  -h, --help        Show the help
```

2.1.10 Arbitrary requirements

Let's say we want to give an error if the word *spam* is in the text. To do so, one option is to raise an `ArgumentError` from within your function:

```
from sigtools.modifiers import annotate, autokwargs
from clize import ArgumentError, Parameter, run

@annotate(text=Parameter.REQUIRED,
          prefix='p', suffix='s', reverse='r', repeat='n')
@autokwargs
def echo(prefix='', suffix='', reverse=False, repeat=1, *text):
    """Echoes text back

    text: The text to echo back

    reverse: Reverse text before processing

    repeat: Amount of times to repeat text

    prefix: Prepend this to each line in word

    suffix: Append this to each line in word

    """
    text = ' '.join(text)
    if 'spam' in text:
        raise ArgumentError("I don't want any spam!")
    if reverse:
        text = text[::-1]
    text = text * repeat
    if prefix or suffix:
        return '\n'.join(prefix + line + suffix
                          for line in text.split('\n'))
```

```

    return text

def version():
    """Show the version"""
    return 'echo version 0.2'

if __name__ == '__main__':
    run(echo, alt=version)

```

```

$ ./echo.py spam bacon and eggs
./echo.py: I don't want any spam!
Usage: ./echo.py [OPTIONS] text...

```

If you would like the usage line not to be printed, raise `UserError` instead.

Next up, we will look at how you can have Clize *dispatch to multiple functions* for you.

2.2 Dispatching to multiple functions

So far the previous part of the tutorial showed you how to use clize to *run a single function*. Sometimes your program will need to perform different related actions that involve different parameters. For instance, `git` offers all kinds of commands related to managing a versioned code repository under the `git` command. Alternatively, your program could have one main function and a few auxiliary ones, for instance for verifying the format of a config file, or simply for displaying the program's version.

2.2.1 Alternate actions

These allow you to provide auxiliary functions to your program while one remains the main function. Let's write a program with an alternate command triggered by `--version` that prints the version.

Here are the two functions we could have: `do_nothing` will be the main function while `version` will be provided as an alternate command.

```

VERSION = "0.2"

def do_nothing():
    """Does nothing"""
    return "I did nothing, I swear!"

def version():
    """Show the version"""
    return 'Do Nothing version {0}'.format(VERSION)

```

You use `run` as usual for the main function, but specify the alternate command in the `alt=` parameter:

```

from clize import run

run(do_nothing, alt=version)

```

The `version` function will be available as `--version`:

```
$ python examples/altcommands.py --help
Usage: examples/altcommands.py

Does nothing

Other actions:
  -h, --help    Show the help
  --version     Show the version
```

You can specify more alternate commands in a list. For instance,

```
from sigtools import modifiers

@modifiers.kwoargs('show_time')
def build_date(show_time=False):
    """Show the build date for this version"""
    print("Build date: 17 August 1979", end='')
    if show_time:
        print(" afternoon, about tea time")
    print()

run(do_nothing, alt=[version, build_date])
```

You can instead use a `dict` to specify their names if those automatically drawn from the function names don't suit you:

```
run(do_nothing, alt={
    'totally-not-the-version': version,
    'birthdate': build_date
})
```

```
$ python examples/altcommands.py --help
Usage: examples/altcommands.py

Does nothing

Other actions:
  --birthdate    Show the build date for this version
  -h, --help     Show the help
  --totally-not-the-version
                  Show the version
```

Using a `collections.OrderedDict` instance rather than `dict` will guarantee the order they appear in the help is the same as in the source.

2.2.2 Multiple commands

This allows you to keep multiple commands under a single program without singling one out as the main one. They become available by naming the subcommand directly after the program's name on the command line.

Let's see how we can use it in a mock todo list application:

```
def add(*text):
    """Adds an entry to the to-do list.

    text: The text associated with the entry.
```

```

"""
    return "OK I will remember that."

def list_():
    """Lists the existing entries."""
    return "Sorry I forgot it all :("

```

You can specify multiple commands to run by passing each function as an argument to `run`:

```

from clize import run

run(add, list_)

```

```

$ python examples/multicommands.py add A very important note.
OK I will remember that.
$ python examples/multicommands.py list
Sorry I forgot it all :(

```

Alternatively, as with *alternate commands*, you can pass in an `iterable`, a `dict` or an `OrderedDict`.

Because it isn't passed a regular function with a docstring, Clize can't determine an appropriate description from a docstring. You can explicitly give it a description with the `description=` parameter. Likewise, you can add footnotes with the `footnotes=` parameter. The format is the same as with other docstrings, just without documentation for parameters.

```

run(add, list_, description="""
    A reliable to-do list utility.

    Store entries at your own risk.
""")

```

```

$ python examples/multicommands.py --help
Usage: examples/multicommands.py command [args...]

A reliable to-do list utility.

Store entries at your own risk.

Commands:
  add  Adds an entry to the to-do list.
  list Lists the existing entries.

```

Often, you will need to share a few characteristics, for instance a set of parameters, between multiple functions. See how Clize helps you do that in *Function compositing*.

2.3 Function compositing

One of Python's strengths is how easy it is to manipulate functions and combine them. However, this typically breaks tools such as Clize which try to inspect the resulting callable and only get vague information. Fortunately, using the functions found in `sigtools`, we can overcome this drawback.

Let's look at how you can create decorators that work well with Clize.

Creating decorators is useful if you want to share behaviour across multiple functions passed to `run`, such as extra parameters or input/output formatting.

2.3.1 Using a decorator to add new parameters and modify the return value

Let's create a decorator that transforms the output of the wrapped function when passed a specific flag.

```
from sigtools.modifiers import autokwargs
from sigtools.wrappers import wrapper_decorator

@wrapper_decorator
@autokwargs
def with_uppercase(wrapped, uppercase=False, *args, **kwargs):
    """
    Formatting options:

    uppercase: Print output in capitals
    """
    ret = wrapped(*args, **kwargs)
    if uppercase:
        return str(ret).upper()
    else:
        return ret
```

`wrapper_decorator` lets our `with_uppercase` function decorate other functions:

```
from clize import run

@with_uppercase
def hello_world(name=None):
    """Says hello world

    name: Who to say hello to
    """
    if name is not None:
        return 'Hello ' + name
    else:
        return 'Hello world!'

if __name__ == '__main__':
    run(hello_world)
```

Each time the decorated function is run, `with_uppercase` will be run with the decorated function as first argument wrapped.

`wrapper_decorator` will tell Clize that the combined function has the same signature as:

```
@kwargs('uppercase')
def hello_world(name=None, uppercase=False):
    pass
```

This is the signature you would get by “putting” the parameters of the decorated function in place of the wrapper’s `*args, **kwargs`. It is what `wrapper_decorator` expects it to do, but that can be changed.

With the correct signature advertised, the command-line interface matches it:

```
$ python examples/decorators.py --uppercase
HELLO WORLD!
$ python examples/decorators.py john
Hello john
```

```
$ python examples/decorators.py john --uppercase
HELLO JOHN
```

The help system will also pick up on the fact that the function is decorated and will read parameter descriptions from the decorator's docstring:

```
$ python decorators.py --help
Usage: decorators.py [OPTIONS] [name]

Says hello world

Positional arguments:
  name          Who to say hello to

Formatting options:
  --uppercase    Print output in capitals

Other actions:
  -h, --help     Show the help
```

2.3.2 Providing an argument using a decorator

When you're passing new arguments to the wrapped function in addition to `*args`, `**kwargs`, things get a little more complicated. You have to tell Clize that some of the wrapped function's parameters shouldn't appear in place of the wrapper's `*args`, `**kwargs`. call `wrapper_decorator` with the number of positional arguments you insert before `*args`, then the names of each named argument that you pass to the wrapped function.

See also:

Picking the appropriate arguments for forwards

The arguments for `wrapper_decorator` are the same as in `sigtools.specifiers.forwards`, so you may use this section for further information.

```
from sigtools.modifiers import autokwargs
from sigtools.wrappers import wrapper_decorator

def get_branch_object(repository, branch_name):
    return repository, branch_name

@wrapper_decorator(0, 'branch')
@autokwargs
def with_branch(wrapped,
                repository='.', branch='master',
                *args, **kwargs):
    """Decorate with this so your function receives a branch object

    repository: A directory belonging to the repository to operate on

    branch: The name of the branch to operate on
    """
    return wrapped(
        *args, branch=get_branch_object(repository, branch), **kwargs)
```

Here we pass 0, 'branch' to `wrapper_decorator` because we call wrapped with no positional arguments besides `*args`, and branch as named argument.

You can then use the decorator like before:

```
from clize import run

@with_branch
@autokwargs
def diff(branch=None):
    """Show the differences between the committed code and the working tree."""
    return "I'm different."

@with_branch
@autokwargs
def commit(branch=None, *text):
    """Commit the changes.

    text: A message to store alongside the commit
    """
    return "All saved.: " + ' '.join(text)

@with_branch
@autokwargs
def revert(branch=None):
    """Revert the changes made in the working tree."""
    return "There is no chip, John."

run(diff, commit, revert,
     description="A mockup version control system(like git, hg or bzip)")
```

2.3.3 Using a composed function to process arguments to a parameter

You can use `clize.parameters.argument_decorator` to have a second function process an argument while still being able to use parameters of its own:

```
from clize import run
from clize.parameters import argument_decorator

@argument_decorator
def read_server(arg, *, port=80, _6=False):
    """
    Options for {param}:

    port: Which port to connect on

    _6: Use IPv6?
    """
    return (arg, port, _6)

def get_page(server:read_server, path):
    """
    server: The server to contact
```

```

    path: The path of the resource to fetch
    """
    print("Connecting to", server, "to get", path)

run(get_page)

```

`read_server`'s parameters will be available on the CLI. When a value is read that would feed the server parameter, `read_server` is called with it and its collected arguments. Its return value is then used as the server parameter of `get_page`:

```

$ python argdeco.py --help
Usage: argdeco.py [OPTIONS] [--port=INT] [-6] server path

Arguments:
  server      The server to contact
  path        The path of the resource to fetch

Options for server:
  --port=INT  Which port to connect on (default: 80)
  -6          Use IPv6?

Other actions:
  -h, --help  Show the help

```

A few notes:

- Besides `arg` which receives the original value, you can only use named parameters
- The decorator's docstring is used to document its parameters. It can be preferable to use a [section](#) in order to distinguish them from other parameters.
- Appearances of `{param}` in the docstring are replaced with the parameter's name.

You can also use this on named parameters as well as on `*args`, but the names of the composited parameters must not conflict with other parameters:

```

from clize import run
from clize.parameters import argument_decorator

@argument_decorator
def read_server(arg, *, port=80, _6=False):
    """
    Options for {param}:

    port: Which port to connect on

    _6: Use IPv6?
    """
    return (arg, port, _6)

def get_page(path, *servers:read_server):
    """
    server: The server to contact

    path: The path of the resource to fetch
    """
    print("Connecting to", servers, "to get", path)

```

```
run(get_page)
```

```
$ python argdeco.py --help
Usage: argdeco.py [OPTIONS] path [--port=INT] [-6] servers...

Arguments:
  path          The path of the resource to fetch
  servers...

Options for servers:
  --port=INT    Which port to connect on (default: 80)
  -6           Use IPv6?

Other actions:
  -h, --help    Show the help
$ python argdeco.py -6 abc
argdeco.py: Missing required arguments: servers
Usage: argdeco.py [OPTIONS] path [--port=INT] [-6] servers...
$ python argdeco.py /eggs -6 abc
Connecting to (('abc', 80, True),) to get /eggs
$ python argdeco.py /eggs -6 abc def
Connecting to (('abc', 80, True), ('def', 80, False)) to get /eggs
$ python argdeco.py /eggs -6 abc def --port 8080 cheese
Connecting to (('abc', 80, True), ('def', 80, False), ('cheese', 8080, False)) to get /eggs
```

Congratulations, you've reached the end of the tutorials! You can check out the [parameter reference](#) or see how you can [extend the parser](#).

These documents document specific aspects or usages of Clize.

3.1 Extending the parser

Clize allows the parser to be extended through, most notably, new parameters. They allow you to introduce new argument parsing behaviour within the parser's constraints:

- The finality of argument parsing in Clize is to determine `func`, `args` and `kwargs` in order to do `func(*args, **kwargs)`.
- Arguments that start with `-` are looked up in the named parameters table by their first letter when there is only one dash, or until the end of the argument or `=` when there are two dashes.
- Other arguments are processed by the positional parameters in the order they are stored.

This document explains each step of the CLI inference and argument parsing process. An example shows how you can add a custom kind of parameter.

3.1.1 Execution overview

1. `clize.run` is called with either a function, or a sequence or mapping of functions.
2. It forwards that to `Clize.get_cli` in order to get a *cli object* to run.
3. If there is more than one function, `Clize.get_cli` uses `SubcommandDispatcher` to wrap it, otherwise it creates a new `Clize` instance with it and returns it.
4. `run` calls that cli object with `sys.argv` unpacked (ie. `obj(*sys.argv)`).
5. Assuming that cli object is a `Clize` instance, it will look at the function that was passed and determine a `CliSignature` object from it.
6. `Clize` calls the cli signature's `read_arguments` method with the command-line arguments, which returns a `CliBoundArguments` instance.
7. That `CliBoundArguments` instance carries information such as the arguments that the function will be called with or the instruction to replace that function with another entirely. `Clize` then runs the chosen function and `clize.run` prints its return value.

3.1.2 Parameter conversion

In step 5 above, `CliSignature.from_signature` converts each parameter. Here is the process followed for each parameter:

1. The annotation for each parameter is read as a sequence. If it isn't one then it is wrapped in one.
2. The annotations are searched for `clize.parser.Parameter.IGNORE`. If found, the parameter is dropped with no further processing.
3. The annotations are searched for a `parameter converter` function. If none is found, `default_converter` is used.
4. The parameter converter is called with the `inspect.Parameter` object representing the parameter and the sequence of annotations without the parameter converter. Its return value, expected to be a `clize.parser.Parameter` instance, is added to the list of parameters for the resulting `CliSignature` instance.

The default parameter converter

The default parameter converter works as follows:

- It looks at the parameter's type and checks whether it is a named or positional parameter. This is used to check if it is legal to assign aliases to it and to determine what cli parameter class will be used to represent it.
- It looks at the parameter's default value and extracts its type, expecting that it is a valid `value converter`. If there isn't one the parameter is marked as required.
- The annotations sequence is iterated on:
 - If the annotation is a `Parameter` instance, it is returned immediately with no processing.
 - If the annotation is a `value converter` it is used instead of the default value's type. Specifying a value converter is required when the default value's type isn't a valid one itself.
 - If it is a string, it is used as an alias unless the parameter is positional.
- Finally, depending on the above, a parameter class is picked, instantiated and returned:
 - `PositionalParameter` if the parameter was positional,
 - `ExtraPosArgsParameter` for a `*args` parameter,
 - `OptionParameter` for a named parameter that takes an argument,
 - `IntOptionParameter` if that argument is of type `int`
 - `FlagParameter` for a named parameter with `False` as default and `bool` as type,
 - An error is raised for `**kwargs` parameters, as their expected equivalent in a CLI is largely subjective. If you want to forward arguments to another function, consider using `function compositing` instead of having a CLI parameter handle it.

3.1.3 The argument parser

The argument parsing is orchestrated by `CliBoundArguments` during its initialization. For each argument of its input, it selects the appropriate `Parameter` instance to handle it. If the argument on the input starts with `-` it looks in the `CliSignature.named` dictionary. If not, it picks the next positional parameter from `CliSignature.positional`. The parameter's `read_argument` and `apply_generic_flags` methods are called.

`Parameter.read_argument(ba, i)`

Reads one or more arguments from `ba.in_args` from position `i`.

Parameters

- **ba** (`clize.parser.CliBoundArguments`) – The bound arguments object this call is expected to mutate.
- **i** (`int`) – The current position in `ba.args`.

This method is expected to mutate `ba`, an instance of `CliBoundArguments`. In particular, it should add any relevant arguments to `ba's args` and `kwargs` attributes which are used when calling the wrapped callable as in `func(*args, **kwargs)`. It can also set the `func` attribute which overrides the `Clize` object's wrapped callable.

Part of the parameter's behavior is split from `read_argument` into `apply_generic_flags` in order to facilitate subclassing:

`Parameter.apply_generic_flags(ba)`

Called after `read_argument` in order to set attributes on `ba` independently of the arguments.

Parameters **ba** (`clize.parser.CliBoundArguments`) – The bound arguments object this call is expected to mutate.

The base implementation of this method applies the `last_option` setting if applicable and discards itself from `CliBoundArguments.unsatisfied`

The both of these methods are expected to discard the parameter from `unsatisfied`, the list of still-unsatisfied required parameters, when applicable. The `sticky`, `posarg_only` and `skip` can also be modified to change the ongoing argument reading process.

3.1.4 Example: Implementing one_of

`clize.parameters.one_of` creates a parameter annotation that modifies the parameter to only allow values from a given list:

```
from clize import run, parameters

def func(breakfast:parameters.one_of('ham', 'spam')):
    """Serves breakfast

    breakfast: what food to serve
    """
    print("{0} is served!".format(breakfast))

run(func)
```

The `breakfast` parameter now only allows ham and spam:

```
$ python breakfast.py ham
ham is served!
$ python breakfast.py spam
spam is served!
$ python breakfast.py eggs
breakfast.py: Bad value for breakfast: eggs
Usage: breakfast.py breakfast
```

A list is produced when `list` is supplied:

```
$ python breakfast.py list
breakfast.py: Possible values for breakfast:

    ham
    spam
```

Also, it hints at the `list` keyword on the help page:

```
$ python breakfast.py --help
Usage: breakfast.py breakfast

Serves breakfast

Arguments:
  breakfast    what food to serve (use "list" for options)

Other actions:
  -h, --help  Show the help
```

`one_of` is implemented in Clize as a wrapper around `mapped` which offers several more features. In this example we will only reimplement the features described above.

Creating a parameter class for us to edit

```
from clize import run, parser

class OneOfParameter(object):
    def __init__(self, values, **kwargs):
        super().__init__(**kwargs)
        self.values = values

def one_of(*values):
    return parser.use_mixin(OneOfParameter, kwargs={'values': values})

def func(breakfast:one_of('ham', 'spam')):
    """Serves breakfast

    breakfast: what food to serve
    """
    print("{0} is served!".format(breakfast))

run(func)
```

Here we used `parser.use_mixin` to implement the parameter annotation. It will create a parameter instance that inherits from both `OneOfParameter` and the appropriate class for the parameter being annotated: `PositionalParameter`, `OptionParameter` or `ExtraPosArgsParameter`. This means our class will be able to override some of those classes' methods.

For now, it works just like a regular parameter:

```
$ python breakfast.py abcdef
abcdef is served!
```

Changing `coerce_value` to validate the value

`PositionalParameter`, `OptionParameter` and `ExtraPosArgsParameter` all use `ParameterWithValue.coerce_value`. We override it to only accept the values we recorded:

```
from clize import errors

class OneOfParameter(parser.ParameterWithValue):
    def __init__(self, values, **kwargs):
        super().__init__(**kwargs)
        self.values = set(values)

    def coerce_value(self, arg, ba):
        if arg in self.values:
            return arg
        else:
            raise errors.BadArgumentFormat(arg)
```

It now only accepts the provided values:

```
$ python breakfast.py ham
ham is served!
$ python breakfast.py spam
spam is served!
$ python breakfast.py eggs
breakfast.py: Bad value for breakfast: eggs
Usage: breakfast.py breakfast
```

Displaying the list of choices

We can check if the passed value is list within `coerce_value`. When that is the case, we change `func` and swallow the following arguments. However, to ensure that the `read_argument` method doesn't alter this, we need to skip its execution. In order to do this we will raise an exception from `coerce_value` and catch it in `read_argument`:

```
class _ShowList(Exception):
    pass

class OneOfParameter(parser.ParameterWithValue):
    def __init__(self, values, **kwargs):
        super().__init__(**kwargs)
        self.values = values

    def coerce_value(self, arg, ba):
        if arg == 'list':
            raise _ShowList
        elif arg in self.values:
            return arg
        else:
            raise errors.BadArgumentFormat(arg)

    def read_argument(self, ba, i):
        try:
            super(OneOfParameter, self).read_argument(ba, i)
        except _ShowList:
```

```
        ba.func = self.show_list
        ba.args[:] = []
        ba.kwargs.clear()
        ba.sticky = parser.IgnoreAllArguments()
        ba.posarg_only = True

    def show_list(self):
        for val in self.values:
            print(val)
```

On `ba`, setting `func` overrides the function to be run (normally the function passed to `run`). `args` and `kwargs` are the positional and keyword argument that will be passed to that function. Setting `sticky` to an `IgnoreAllArguments` instance will swallow all positional arguments instead of adding them to `args`, and `posarg_only` makes keyword arguments be processed as if they were positional arguments so they get ignored too.

```
$ python breakfast.py list
ham
spam
$ python breakfast.py list --ERROR
ham
spam
```

The list is printed, even if erroneous arguments follow.

Adding a hint to the help page

Clize uses `Parameter.show_help` to produce the text used to describe parameters. It uses `Parameter.help_parens` to provide the content inside the parenthesis after the parameter description.

```
class OneOfParameter(parser.ParameterWithValue):

    ...

    def help_parens(self):
        for s in super(OneOfParameter, self).help_parens():
            yield s
        yield 'use "list" for options'
```

The help page now shows the hint:

```
$ python breakfast.py --help
Usage: breakfast.py breakfast

Serves breakfast

Arguments:
  breakfast    what food to serve (use "list" for options)

Other actions:
  -h, --help  Show the help
```

The full example is available in `examples/bfparam.py`.

3.2 Upgrading from older releases

This document will instruct you in porting applications using older clize versions to the newest version.

3.2.1 Upgrading from clize 1 and 2

Clize 3 now only treats keyword-only parameters on the function as named parameters and does not convert any parameter from keyword to positional or vice-versa, much like when the `use_kwoargs` parameter is used in Clize 2. Aliases, and other parameter-related information are now expressed exclusively through parameter annotations. Decorators from `sigtools.modifiers` are the recommended way to set these up on Python 2.

However, `clize.clize` is provided: it imitates the old behaviour but adds a deprecation warning when used.

Porting code using the `@clize` decorator with no arguments

Consider this code made to work with Clize 1 or 2:

```
from clize import clize, run

@clize
def func(positional, option=3):
    pass # ...

if __name__ == '__main__':
    run(func)
```

Here, you can drop the `@clize` line completely, but you have to convert `option` to a keyword-only parameter. You can use `sigtools.modifiers.autokwoargs` to do so:

```
from sigtools.modifiers import autokwoargs
from clize import run

@autokwoargs
def func(positional, option=3):
    pass # ...

if __name__ == '__main__':
    run(func)
```

Decorating the function with `@autokwoargs` will still let you call it normally, except that any parameter with a default value (here just `option`) will only be accepted as a named argument.

`force_positional`

`force_positional` used to let you specify parameters with defaults that you don't want as named options:

```
from clize import clize, run

@clize(force_positional=['positional_with_default'])
def func(positional, positional_with_default=3, option=6):
    pass # ...

if __name__ == '__main__':
    run(func)
```

The exceptions parameter of `autokwargs` has the same purpose here:

```
from sigtools.modifiers import autokwargs
from clize import run

@autokwargs(exceptions=['positional_with_default'])
def func(positional, positional_with_default=3, option=6):
    pass # ...

if __name__ == '__main__':
    run(func)
```

Porting code that used `alias` or `coerce`

The `alias` and `coerce` were used in order to specify alternate names for options and functions to convert the value of arguments, respectively:

```
from clize import clize, run

@clize(
    alias={'two': ['second'], 'three': ['third']},
    coerce={'one': int, 'three': int})
def func(one, two=2, three=None):
    pass # ...

if __name__ == '__main__':
    run(func)
```

You now pass these as annotations on the corresponding parameter. To keep compatibility with Python 2, we use `sigtools.modifiers.annotate`:

```
from sigtools.modifiers import annotate, autokwargs
from clize import run

@autokwargs
@annotate
def func(one=int, two='second', three=('third', int)):
    pass # ...

if __name__ == '__main__':
    run(func)
```

`require_excess`

Indicating that an `*args`-like parameter is required is now done by annotating the parameter with `Parameter.REQUIRED` or `Parameter.R` for short:

```
from sigtools.modifiers import annotate
from clize import run, Parameter

@annotate(args=Parameter.R)
def func(*args):
    pass # ...

if __name__ == '__main__':
    run(func)
```

extra and make_flag

Alternate actions as shown in Clize 2's tutorial are now done by passing the function directly to *run as shown in the tutorial*. Unlike previously, the alternate command function is passed to the clizer just like the main one.

For other use cases, you should find the appropriate parameter class from `clize.parser` or subclass one, instantiate it and pass it in a sequence as the `extra` parameter of `Clize` or `run`. If the parameter matches one actually present on the source function, annotate that parameter with your `Parameter` instance instead of passing it to `extra`.

3.3 Interoperability with arbitrary callables

Clize operates as a callable that takes each item from `sys.argv` or something supposed to replace it. It is therefore easy to substitute it with another callable that has such parameters.

3.3.1 Avoiding automatic CLI inference

When an object is passed to `run`, either as sole command, one in many subcommands or as an alternative action, it attempts to make a *CLI object* out of it if it isn't one already. It simply checks if there is a `cli` attribute and uses it, or it wraps it with `Clize`.

To insert an arbitrary callable, you must therefore place it as the `cli` attribute of whatever object you pass to `run`.

`clize.Clize.as_is` does exactly that. You can use it as a decorator or when passing it to `run`:

```
import argparse

from clize import Clize, parameters, run

@Clize.as_is
def echo_argv(*args):
    print(*args, sep=' | ')

def using_argparse(name: parameters.pass_name, *args):
    parser = argparse.ArgumentParser(prog=name)
    parser.add_argument('--ham')
    ns = parser.parse_args(args=args)
    print(ns.ham)

run(echo_argv, Clize.as_is(using_argparse))
```

```
$ python interop.py echo-argv ab cd ef
interop.py echo-argv | ab | cd | ef
$ python interop.py using-argparse --help
usage: interop.py using-argparse [-h] [--ham HAM]

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

$ python interop.py using-argparse spam
usage: interop.py using-argparse [-h] [--ham HAM]
interop.py using-argparse: error: unrecognized arguments: spam
$ python interop.py using-argparse --ham spam
spam
```

3.3.2 Providing a description in the parent command's help

If you try to access the above program's help screen, Clize will just leave the description for each external command empty:

```
: .tox/docs/bin/python interop.py --help
Usage: interop.py command [args...]

Commands:
  echo-argv
  using-argparse
```

Clize expects to find a description as `cli.helper.description`. You can either create an object there or let `Clize.as_is` do it for you:

```
@Clize.as_is(description="Prints argv separated by pipes")
def echo_argv(*args):
    print(*args, sep=' | ')

...

run(echo_argv,
    Clize.as_is(using_argparse,
        description="Prints the value of the --ham option"))
```

```
$ python interop.py --help
Usage: interop.py command [args...]

Commands:
  echo-argv          Prints argv separated by pipes
  using-argparse     Prints the value of the --ham option
```

3.3.3 Advertising command usage

To produce the `--help --usage` output, Clize uses `cli.helper.usages()` to produce an iterable of (command, usage) pairs. When it can't determine it, it shows a generic usage signature instead.

You can use `Clize.as_is`'s `usages=` parameter to provide it:

```
run(echo_argv,
    Clize.as_is(using_argparse,
        description="Prints the value of the --ham option"),
    usages=['--help', '--ham HAM'])
```

```
$ python interop.py --help --usage
interop.py --help [--usage]
interop.py echo-argv [args...]
interop.py using-argparse --help
interop.py using-argparse [--ham HAM]
```

The example is available as `examples/interop.py`.

Reference

The user reference lists all capabilities of each kind of parameter. The API reference comes in handy if you're extending clize.

4.1 User reference

Clize deduces what kind of parameter to pick for the CLI depending on what kind of parameter is found on the Python function as well as its annotations.

Note: Python 2 compatibility

In this document we will be showing examples that use Python 3 syntax such as annotations and keyword-only parameters for conciseness. To translate those into Python 2, you can use `sigtools.modifiers.kwoargs` and `sigtools.modifiers.annotate`.

For instance, given this Python 3 function:

```
def func(ab:int, *, cd:'c'=2):  
    pass
```

You would write in Python 2:

```
from sigtools import modifiers  
  
@modifiers.kwoargs('cd')  
@modifiers.annotate(ab=int, cd='c')  
def func(ab, cd=2):  
    pass
```

You can pass annotations as a sequence:

```
def func(*, param:('p', int)): pass  
def func(*, param:['p', int]): pass
```

When only one annotation is needed, you can omit the sequence:

```
def func(*, param:'p'): pass  
def func(*, param:('p',)): pass  
def func(*, param:['p']): pass
```

4.1.1 Annotations for parameters that handle a value

The positional and options parameters detailed later both handle the following features:

Specifying a value converter

A function or callable decorated with `parser.value_converter` passed as annotation will be used during parsing to convert the value from the string found in `sys.argv` into a value suitable for the annotated function.

```
from clize import run, parser

@parser.value_converter
def wrap_xy(arg):
    return 'x' + arg + 'y'

def func(a, b:wrap_xy):
    print(repr(a), repr(b))

run(func)
```

```
$ python valconv.py abc def
'abc' 'xdefy'
```

`def` was transformed into `xdefy` because of the value converter.

Besides callables decorated with `parser.value_converter`, the built-in functions `int`, `float` and `bool` are also recognized as value converters.

Specifying a default value

The parameter's default value is used without conversion. If no *value converter* is specified, its type is used instead. When a default value is specified, the parameter becomes optional.

```
>>> def func(par=3):
...     print(repr(par))
...
>>> run(func, exit=False, args=['func', '46'])
46
>>> run(func, exit=False, args=['func'])
3
```

Therefore, be careful not to use values of types for which the constructor does not handle strings, unless you specify a converter:

```
>>> from datetime import datetime
>>> now = datetime.utcnow()
>>> def fail(par=now):
...     print(repr(par))
...
>>> run(fail, exit=False, args=['func', '...'])
Traceback (most recent call last):
...
TypeError: an integer is required (got type str)
>>> from dateutil.parser import parse
>>> from datetime import datetime
>>> now = datetime.utcnow()
```

```
>>> def func(par:parse=now):
...     print(par)
...
>>> run(func, exit=False, args=['func', '1/1/2016'])
2016-01-01 00:00:00
```

Ignoring the source parameter's default value

`Parameter.REQUIRED`

Annotate a parameter with this to force it to be required, even if there is a default value in the source.

```
>>> from clize import run, Parameter
>>> def func(par:Parameter.REQUIRED=3):
...     pass
...
>>> run(func, exit=False, args=['func'])
func: Missing required arguments: par
Usage: func par
```

4.1.2 Positional parameters

Normal parameters in Python are turned into positional parameters on the CLI. Plain arguments on the command line (those that don't start with a `-`) are processed by those and assigned in the order they appear:

```
from clize import run

def func(posparam1, posparam2, posparam3):
    print('posparam1', posparam1)
    print('posparam2', posparam2)
    print('posparam3', posparam3)

run(func)
```

```
$ python posparams.py one two three
posparam1 one
posparam2 two
posparam3 three
```

It also shares the features detailed in *Annotations for parameters that handle a value*.

4.1.3 Parameter that collects remaining positional arguments

An `*args`-like parameter in Python becomes a repeatable positional parameter on the CLI:

```
from clize import run

def func(param, *args):
    print('param', param)
    print('args', args)

run(func)
```

```
$ python argslike.py one two three
param one
args ('two', 'three')
```

Parameter.REQUIRED

When used on an `*args` parameter, requires at least one value to be provided.

You can use `clize.parameters.multi` for more options.

4.1.4 Named parameters

Clize treats keyword-only parameters as named parameters, which, instead of their position, get designated by they name preceeded by `--`, or by `-` if the name is only one character long.

There are a couple kinds of named parameters detailed along with examples in the sections below.

They all understand annotations to specify alternate names for them: Simply pass them as strings in the parameter's annotation:

```
from clize import run

def func(*, par:('p', 'param')):
    print('par', par)

run(func)
```

```
$ python named.py --par value
par value
$ python named.py -p value
par value
$ python named.py --param value
par value
```

All parameter names are converted by removing any underscores (`_`) off the extremities of the string and replacing the remaining ones with dashes (`-`).

Python 2 support for named parameters

Python 2 has no keyword-only parameters. To fill that gap, you can use the decorators from `sigtools.modifiers` to emulate them.

```
@DECORATOR
def func(ab, cd, de=None, fg=None, hi=None):
    ...
```

@DECORATOR	Parameters that become keyword-only
@kwoargs('cd', 'fg')	cd and fg
@kwoargs(start='fg')	fg and all following parameters
@autokwoargs	All parameters with default values
@autokwoargs(exceptions=['fg'])	Same, except for fg

4.1.5 Named parameters that take an argument

Keyword-only parameters in Python become named parameters on the CLI: They get designated by their name rather than by their position:

```
from clize import run

def func(arg, *, o, par):
    print('arg', arg)
```

```
print('o', o)
print('par', par)

run(func)
```

```
$ python opt.py -o abc def --par ghi
arg def
o abc
par ghi
$ python opt.py -oabc --par=def ghi
arg ghi
o abc
par def
```

The parameter is designated by prefixing its name with two dashes (eg. `--par`) or just one dash if the name is only one character long (eg. `-o`). The value is given either as a second argument (first example) or glued to the parameter name for the short form, glued using an equals sign (`=`) for the long form.

It shares the features of *Annotations for parameters that handle a value* and *Named parameters*.

4.1.6 Named parameters that take an integer argument

A variant of *Named parameters that take an argument* used when the value type from *Annotations for parameters that handle a value* is `int`. The only difference is that when designating the parameter using the short glued form, you can chain other short named parameters afterwards:

```
from clize import run

def func(*, i: int, o):
    print('i', i)
    print('o', o)

run(func)
```

```
$ python intopt.py -i42o abcdef
i 42
o abcdef
```

4.1.7 Flag parameters

Flag parameters are named parameters that unlike *options* do not take an argument. Instead, they set their corresponding parameter in Python to `True` if mentionned.

You can create them by having a keyword-only parameter take `False` as default value:

```
from clize import run

def func(*, flag=False):
    print('flag', flag)

run(func)
```

```
$ python flag.py
flag False
$ python flag.py --flag
flag True
```

```
$ python flag.py --flag=0
flag False
```

Additionally, you can chain their short form on the command line with other short parameters:

```
from clize import run

def func(*, flag:'f'=False, other_flag:'g'=False, opt:'o'):
    print('flag', flag)
    print('other_flag', other_flag)
    print('opt', opt)

run(func)
```

```
$ python glueflag.py -fgo arg
flag True
other_flag True
opt arg
$ python glueflag.py -fo arg
flag True
other_flag False
opt arg
```

4.1.8 Mapped parameters

`clize.parameters.mapped(values, *, list_name='list', case_sensitive=None)`

Creates an annotation for parameters that maps input values to Python objects.

Parameters

- **values** (*sequence*) – A sequence of `pyobj`, `names`, `description` tuples. For each item, the user can specify a name from `names` and the parameter will receive the corresponding `pyobj` value. `description` is used when listing the possible values.
- **list_name** (*str*) – The value the user can use to show a list of possible values and their description.
- **case_sensitive** (*bool*) – Force case-sensitiveness for the input values. The default is to guess based on the contents of values.

```
greeting = parameters.mapped([
    ('Hello', ['hello', 'hi'], 'A welcoming message'),
    ('Goodbye', ['goodbye', 'bye'], 'A parting message'),
])

@modifiers.kwoargs('kind')
@modifiers.annotate(kind=('k', greeting))
def main(name='world', kind='Hello'):
    """
    name: Who is the message for?

    kind: What kind of message should be given to name?
    """
    return '{0} {1}!'.format(kind, name)
```

```
$ python -m examples.mapped -k list
python -m examples.mapped: Possible values for --kind:
```

```

    hello, hi      A welcoming message
    goodbye, bye   A parting message
$ python -m examples.mapped -k hello
Hello world!
$ python -m examples.mapped -k hi
Hello world!
$ python -m examples.mapped -k bye
Goodbye world!
$ python -m examples.mapped
Hello world!

```

`clize.parameters.one_of(*values, case_sensitive=None, list_name='list')`

Creates an annotation for a parameter that only accepts the given values.

Parameters

- **values** – value, description tuples, or just the accepted values
- **list_name** (*str*) – The value the user can use to show a list of possible values and their description.
- **case_sensitive** (*bool*) – Force case-sensitiveness for the input values. The default is to guess based on the contents of values.

4.1.9 Repeatable parameters

`clize.parameters.multi(min=0, max=None)`

For option parameters, allows the parameter to be repeated on the command-line with an optional minimum or maximum. For `*args`-like parameters, just adds the optional bounds.

```

@modifiers.kwoargs('listen')
@modifiers.annotate(listen=('l', parameters.multi(min=1, max=3)))
def main(listen):
    """Listens on the given addresses

    listen: An address to listen on.
    """
    for address in listen:
        print('Listening on {}'.format(address))

```

```

$ python -m examples.multi -l bacon
Listening on bacon
$ python -m examples.multi -l bacon -l ham -l eggs
Listening on bacon
Listening on ham
Listening on eggs
$ python -m examples.multi -l bacon -l ham -l eggs -l spam
python -m examples.multi: Received too many values for --listen
Usage: python -m examples.multi [OPTIONS]
$ python -m examples.multi
python -m examples.multi: Missing required arguments: --listen
Usage: python -m examples.multi [OPTIONS]

```

4.1.10 Decorated arguments

`clize.parameters.argument_decorator(f)`

Decorates a function to create an annotation for adding parameters to qualify another.

```
@argument_decorator
def capitalize(arg, *, capitalize:('c', 'upper')=False, reverse:'r'=False):
    """
    Options to qualify {param}:

    capitalize: Make {param} uppercased

    reverse: Reverse {param}
    """
    if capitalize:
        arg = arg.upper()
    if reverse:
        arg = arg[::-1]
    return arg

def main(*args:capitalize):
    """
    args: stuff
    """
    return ' '.join(args)
```

```
$ python -m examples.argdeco --help
Usage: python -m examples.argdeco [OPTIONS] [[-c] [-r] args...]

Arguments:
  args...      stuff

Options to qualify args:
  -c, --capitalize, --upper
                  Make args uppercased
  -r, --reverse  Reverse args

Other actions:
  -h, --help      Show the help
$ python -m examples.argdeco abc -c def ghi
abc DEF ghi
```

4.1.11 Annotations that work on any parameter

The following objects can be used as annotation on any parameter:

Parameter converters

Callables decorated with `clize.parser.parameter_converter` are used instead of the *default converter* to construct a CLI parameter for the annotated Python parameter.

The callable can return a *Parameter* instance or *Parameter.IGNORE* to instruct clize to drop the parameter.

```
>>> from clize import run, parser
>>> @parser.parameter_converter
```

```

... def print_and_ignore(param, annotations):
...     print(repr(param), annotations)
...     return parser.Parameter.IGNORE
...
>>> def func(par:(print_and_ignore,int)):
...     pass
...
>>> run(func, exit=False, args=['func', '--help'])
<Parameter at 0x7fc6b0c3dae8 'par'> (<class 'int'>,)
Usage: func

Other actions:
  -h, --help  Show the help

```

Unless you are creating new kinds of parameters this should not be useful to you directly. Note that custom parameter converters are likely to use different conventions than those described in this reference.

See also:

Extending the parser

Parameter instances

A *Parameter* instance seen in an annotation will be used to represent that parameter on the CLI, without any further processing. Using a *parameter converter* is recommended over this.

Skipping parameters

`Parameter.IGNORE = clize.Parameter.IGNORE`

Annotate a parameter with this and it will be dropped from the resulting CLI signature.

Note that it is dangerous to use this on anything except:

- On `*args` and `**kwargs`-like parameters,
- On keyword parameters with defaults.

For instance, clize's default converter does not handle `**kwargs`:

```

>>> from clize import run, Parameter
>>> def fail(**kwargs):
...     pass
...
>>> run(fail, exit=False)
Traceback (most recent call last):
...
ValueError: This converter cannot convert parameter 'kwargs'.

```

However, if we decorate that parameter with `Parameter.IGNORE`, clize ignores it:

```

>>> def func(**kwargs:Parameter.IGNORE):
...     pass
...
>>> run(func, exit=False)
>>> run(func, exit=False, args=['func', '--help'])
Usage: func

Other actions:
  -h, --help  Show the help

```

Hiding parameters from the help

`Parameter.UNDOCUMENTED = clize.Parameter.UNDOCUMENTED`

Parameters annotated with this will be omitted from the documentation (`--help`).

```
>>> from clize import run, Parameter
>>> def func(*, o1, o2:Parameter.UNDOCUMENTED):
...     pass
...
>>> run(func, exit=False, args=['func', '--help'])
Usage: func [OPTIONS]

Options:
  --o1=STR

Other actions:
  -h, --help  Show the help
```

Forcing arguments to be treated as positional

`Parameter.LAST_OPTION = clize.Parameter.LAST_OPTION`

Annotate a parameter with this and all following arguments will be processed as positional.

```
>>> from clize import run, Parameter
>>> def func(a, b, c, *, d:Parameter.LAST_OPTION, e=''):
...     print("a:", a)
...     print("b:", b)
...     print("c:", c)
...     print("d:", d)
...     print("e:", e)
...
Usage: [OPTIONS] a b c
>>> run(func, exit=False, args=['func', 'one', '-d', 'alpha', '-e', 'beta'])
a: one
b: -e
c: beta
d: alpha
e:
```

Here, `-e beta` was received by the `b` and `c` parameters rather than `e`, because it was processed after `-d alpha`, which triggered the parameter `d` which had the annotation.

Retrieving the executable name

`clize.parameters.pass_name (param, annotations)`

Parameters decorated with this will receive the executable name as argument.

This can be either the path to a Python file, or `python -m some.module`. It is also appended with sub-command names.

```
from clize import run, parameters

def main(name:parameters.pass_name, arg):
```

```

print('name:', name)
print('arg:', arg)

def alt(arg, *, name:parameters.pass_name):
    print('arg:', arg)
    print('name:', name)

run(main, alt=alt)

```

```

$ python pn.py ham
name: pn.py
arg: ham
$ python -m pn ham
name: python -m pn
arg: ham
$ python pn.py --alt spam
arg: spam
name: pn.py --alt
$ python -m pn --alt spam
arg: spam
name: python -m pn --alt

```

Inserting arbitrary values

`clize.parameters.value_inserter` (*value_factory*)

Create an annotation that hides a parameter from the command-line and always gives it the result of a function.

Parameters `value_factory` (*function*) – Called to determine the value to provide for the parameter. The current `parser.CliBoundArguments` instance is passed as argument, ie. `value_factory(ba)`.

```

from clize import run, parameters

@parameters.value_inserter
def insert_ultimate_answer(ba):
    return 42

def main(arg, ans:insert_ultimate_answer):
    print('arg:', arg)
    print('ans:', ans)

run(main)

```

```

$ python ins.py eggs
arg: eggs
ans: 42

```

4.1.12 Customizing the help using the docstring

Clize draws the text of the `--help` output from the wrapped function's docstring as well as of its `sigtools.wrappers.wrapper_decorator`-based decorators.

While it allows some amount of customization, the input must follow certain rules and the output is formatted by Clize.

The docstring is divided in units of paragraphs. Each paragraph is separated by two newlines.

Documenting positional parameters

To document a parameter, start a paragraph with the name of the parameter you want to document followed by a colon (:), followed by text:

```
from clize import run

def func(one, and_two):
    """
    one: Documentation for the first parameter.

    and_two: Documentation for the second parameter.
    """

run(func)
```

```
$ python docstring.py --help
Usage: docstring.py one and-two

Arguments:
  one           Documentation for the first parameter.
  and-two       Documentation for the second parameter.

Other actions:
  -h, --help    Show the help
```

Documentation for positional parameters is always shown in the order they appear in the function signature.

Description and footnotes

You can add a description as well as footnotes:

```
from clize import run

def func(one, and_two):
    """
    This is a description of the program.

    one: Documentation for the first parameter.

    and_two: Documentation for the second parameter.

    These are footnotes about the program.
    """

run(func)
```

```
$ python docstring.py --help
Usage: docstring.py one and-two

This is a description of the program.

Arguments:
  one           Documentation for the first parameter.
  and-two       Documentation for the second parameter.

Other actions:
  -h, --help    Show the help
```

These are footnotes about the program.

Adding additional information

If you wish, you may add additional information about each parameter in a new paragraph below it:

```
from clize import run

def func(one, and_two):
    """
    This is a description of the program.

    one: Documentation for the first parameter.

    More information about the first parameter.

    and_two: Documentation for the second parameter.

    More information about the second parameter.

    _:_

    These are footnotes about the program.
    """

run(func)
```

To distinguish `and_two`'s information and the footnotes, we inserted a dummy parameter description between them (`_:_`).

```
$ python docstring.py --help
Usage: docstring.py one and-two

This is a description of the program.

Arguments:
  one          Documentation for the first parameter.

More information about the first parameter.

  and-two      Documentation for the second parameter.

More information about the second parameter.

Other actions:
  -h, --help  Show the help

These are footnotes about the program.
```

Ordering named parameters

Unlike positional parameters, named parameters will be shown in the order they appear in the docstring:

```
from clize import run

def func(*, one, and_two):
```

```
"""
    and_two: Documentation for the second parameter.

    one: Documentation for the first parameter.
"""

run(func)
```

```
$ python docstring.py --help
Usage: docstring.py [OPTIONS]

Options:
  --and-two=STR  Documentation for the second parameter.
  --one=STR      Documentation for the first parameter.

Other actions:
  -h, --help    Show the help
```

Creating sections

Named parameters can be arranged into sections. You can create a section by having a paragraph end with a colon (:) before a parameter definition:

```
from clize import run

def func(*, one, and_two, three):
    """
    Great parameters:

    and_two: Documentation for the second parameter.

    one: Documentation for the first parameter.

    Not-so-great parameters:

    three: Documentation for the third parameter.
    """

run(func)
```

```
$ python docstring.py --help
Usage: docstring.py [OPTIONS]

Great parameters:
  --and-two=STR  Documentation for the second parameter.
  --one=STR      Documentation for the first parameter.

Not-so-great parameters:
  --three=STR    Documentation for the third parameter.

Other actions:
  -h, --help    Show the help
```

Unformatted paragraphs

You can insert unformatted text (for instance, code examples) by finishing a paragraph with a colon (:) and starting the unformatted text with at least one space of indentation:

```
from clize import run

def func():
    """
    This      text
    is      automatically formatted.
    However, you may present code blocks like this:

        Unformatted      text

    More      unformatted
    text.

    """
```

```
$ python docstring.py --help
Usage: docstring.py

This text is automatically formatted. However, you may present code blocks
like this:

    Unformatted      text

    More      unformatted
    text.

Other actions:
  -h, --help  Show the help

run(func)
```

A paragraph with just a colon (:) will be omitted from the output, but still trigger unformatted text.

4.2 API Reference

4.2.1 Running functions

`clize.run(*fn, args=None, catch=(), exit=True, out=None, err=None, **kwargs)`

Runs a function or *CLI object* with `args`, prints the return value if not `None`, or catches the given exception types as well as `clize.UserError` and prints their string representation, then exit with the appropriate status code.

Parameters

- **args** (*sequence*) – The arguments to pass the CLI, for instance (`'./a_script.py', 'spam', 'ham'`). If unspecified, uses `sys.argv`.
- **catch** (*sequence of exception classes*) – Catch these exceptions and print their string representation rather than letting Python print an uncaught exception traceback.
- **exit** (*bool*) – If true, exit with the appropriate status code once the function is done.

- **out** (*file*) – The file in which to print the return value of the command. If unspecified, uses `sys.stdout`
- **err** (*file*) – The file in which to print any exception text. If unspecified, uses `sys.stderr`.

class `clize.Clize` (*fn=None, **kwargs*)

Wraps a function into a CLI object that accepts command-line arguments and translates them to match the wrapped function's parameters.

Parameters

- **alt** (*sequence*) – Alternate actions the CLI will handle.
- **help_names** (*sequence of strings*) – Names to use to trigger the help.
- **helper_class** (a type like `ClizeHelp`) – A callable to produce a helper object to be used when the help is triggered. If unset, uses `ClizeHelp`.
- **hide_help** (*bool*) – Mark the parameters used to trigger the help as undocumented.

parameters ()

Returns the parameters used to instantiate this class, minus the wrapped callable.

classmethod **keep** (*fn=None, **kwargs*)

Instead of wrapping the decorated callable, sets its `cli` attribute to a `Clize` instance. Useful if you need to use the decorator but must still be able to call the function regularly.

classmethod **as_is** (*obj=None, *, description=None, usages=None*)

Returns a CLI object which uses the given callable with no translation.

The following parameters improve the decorated object's compatibility with Clize's help output:

Parameters

- **description** – A description for the command.
- **usages** – A list of usages for the command.

See also:

Interoperability with arbitrary callables

classmethod **get_cli** (*obj, **kwargs*)

Makes an attempt to discover a command-line interface for the given object. The process used is as follows:

- 1.If the object has a `cli` attribute, it is used with no further transformation.
- 2.If the object is callable, `Clize` or whichever object this class method is used from is used to build a CLI. `**kwargs` are forwarded to its initializer.
- 3.If the object is iterable, `SubcommandDispatcher` is used on the object, and its `cli` method is used.

Most notably, `clize.run` uses this class method in order to interpret the given object(s).

cli

Returns the object itself, in order to be selected by `get_cli`

helper

A cli object(usually inherited from `help.Help`) when the user requests a help message. See the constructor for ways to affect this attribute.

signature

The `parser.CliSignature` object used to parse arguments.

read_commandline (*args*)

Reads the command-line arguments from *args* and returns a tuple with the callable to run, the name of the program, the positional and named arguments to pass to the callable.

Raises *ArgumentError*

class `clize.SubcommandDispatcher` (*commands=()*, *description=None*, *footnotes=None*)

clizer

alias of `Clize`

cli

4.2.2 Parser

class `clize.parser.CliSignature` (*parameters*)

A collection of parameters that can be used to translate CLI arguments to function arguments.

Parameters *parameters* (*iterable*) – The parameters to use.

converter = `clize.parser.default_converter`

The converter used by default in case none is present in the annotations.

parameters

An ordered dict of all parameters of this cli signature.

positional

List of positional parameters.

alternate

List of parameters that initiate an alternate action.

named

List of named parameters that aren't in *alternate*.

aliases = {}

Maps parameter names to *NamedParameter* instances.

required = `set()`

A set of all required parameters.

classmethod `from_signature` (*sig*, *extra=()*, ***kwargs*)

Takes a signature object and returns an instance of this class deduced from it.

Parameters

- **sig** (*inspect.Signature*) – The signature object to use.
- **extra** (*iterable*) – Extra parameter instances to include.

classmethod `convert_parameter` (*param*)

Convert a Python parameter to a CLI parameter.

read_arguments (*args*, *name*)

Returns a *CliBoundArguments* instance for this CLI signature bound to the given arguments.

Parameters

- **args** (*sequence*) – The CLI arguments, minus the script name.
- **name** (*str*) – The script name.

`clize.parser.parameter_converter(obj)`

Decorates a callable to be interpreted as a parameter converter when passed as an annotation.

It will be called with an `inspect.Parameter` object and a sequence of objects passed as annotations, without the parameter converter itself. It is expected to return a `clize.parser.Parameter` instance or `Parameter.IGNORE`.

`clize.parser.default_converter(param, annotations)`

The default parameter converter. It is described in detail in *The default parameter converter*.

```
clize.parser.use_class(*, pos=<function unimplemented_parameter at 0x7fda10ed58c8>,
                      varargs=<function unimplemented_parameter at 0x7fda10ed58c8>,
                      named=<function unimplemented_parameter at 0x7fda10ed58c8>,
                      varkwargs=<function unimplemented_parameter at 0x7fda10ed58c8>,
                      kwargs={})
```

Creates a parameter converter similar to the default converter that picks one of 4 factory functions depending on the type of parameter.

Parameters

- **pos** (callable that returns a `Parameter` instance) – The parameter factory for positional parameters.
- **varargs** (callable that returns a `Parameter` instance) – The parameter factory for `*args`-like parameters.
- **named** (callable that returns a `Parameter` instance) – The parameter factory for keyword parameters.
- **varkwargs** (callable that returns a `Parameter` instance) – The parameter factory for `**kwargs`-like parameters.
- **kwargs** (`collections.abc.Mapping`) – additional arguments to pass to the chosen factory.

`clize.parser.use_mixin(cls, *, kwargs={})`

Like `use_class`, but creates classes inheriting from `cls` and one of `PositionalParameter`, `ExtraPosArgsParameter`, and `OptionParameter`

Parameters

- **cls** – The class to use as mixin.
- **kwargs** (`collections.abc.Mapping`) – additional arguments to pass to the chosen factory.

class `clize.parser.CliBoundArguments(sig, args, name)`

Command line arguments bound to a `CliSignature` instance.

Parameters

- **sig** (`CliSignature`) – The signature to bind against.
- **args** (`sequence`) – The CLI arguments, minus the script name.
- **name** (`str`) – The script name.

sig

The signature being bound to.

in_args

The CLI arguments, minus the script name.

name

The script name.

args = []

List of arguments to pass to the target function.

kwargs = {}

Mapping of named arguments to pass to the target function.

meta = {}

A dict for parameters to store data for the duration of the argument processing.

func = None

If not `None`, replaces the target function.

post_name = []

List of words to append to the script name when passed to the target function.

The following attributes only exist while arguments are being processed:

posparam = iter(sig.positional)

The iterator over the positional parameters used to process positional arguments.

sticky = None

If not `None`, a parameter that will keep receiving positional arguments.

posarg_only = False

Arguments will always be processed as positional when this is set to `True`.

skip = 0

Amount of arguments to skip.

unsatisfied = set(<required parameters>)

Required parameters that haven't yet been satisfied.

class `clize.parser.Parameter` (*display_name*, *undocumented=False*, *last_option=None*)

Bases: `object`

Represents a CLI parameter.

Parameters

- **display_name** (*str*) – The ‘default’ representation of the parameter.
- **undocumented** (*bool*) – If true, hides the parameter from the command help.
- **last_option** – If `True`, the parameter will set the `posarg_only` flag on the bound arguments.

Also available as `clize.Parameter`.

LAST_OPTION = clize.Parameter.LAST_OPTION

Annotate a parameter with this and all following arguments will be processed as positional.

IGNORE = clize.Parameter.IGNORE

Annotate a parameter with this and it will be dropped from the resulting CLI signature.

UNDOCUMENTED = clize.Parameter.UNDOCUMENTED

Parameters annotated with this will be omitted from the documentation (`--help`).

REQUIRED = clize.Parameter.REQUIRED

Annotate a parameter with this to force it to be required.

Mostly only useful for `*args` parameters. In other cases, simply don't provide a default value.

required = False

Is this parameter required?

is_alternate_action = False

Should this parameter appear as an alternate action or as a regular parameter?

extras = ()

Iterable of extra parameters this parameter incurs

display_name = None

The name used in printing this parameter.

undocumented = None

If true, this parameter is hidden from the documentation.

last_option = None

If true, arguments after this parameter is triggered will all be processed as positional.

read_argument (ba, i)

Reads one or more arguments from `ba.in_args` from position `i`.

Parameters

- **ba** (`clize.parser.CliBoundArguments`) – The bound arguments object this call is expected to mutate.
- **i** (`int`) – The current position in `ba.args`.

apply_generic_flags (ba)

Called after `read_argument` in order to set attributes on `ba` independently of the arguments.

Parameters **ba** (`clize.parser.CliBoundArguments`) – The bound arguments object this call is expected to mutate.

The base implementation of this method applies the `last_option` setting if applicable and discards itself from `CliBoundArguments.unsatisfied`

unsatisfied (ba)

Called after processing arguments if this parameter required and not discarded from `CliBoundArguments.unsatisfied`.

post_parse (ba)

Called after all arguments are processed successfully.

get_all_names ()

Return a string with all of this parameter's names.

get_full_name ()

Return a string that designates this parameter.

show_help (desc, after, f, cols)

Called by `ClizeHelp` to produce the parameter's description in the help output.

show_help_parens ()

Return a string to complement a parameter's description in the `--help` output.

help_parens ()

Return an iterable of strings to complement a parameter's description in the `--help` output. Used by `show_help_parens`

prepare_help (helper)

Called by `ClizeHelp` to allow parameters to complement the help.

Param `clize.help.ClizeHelp helper`: The object charged with displaying the help.

class `clize.parser.ParameterWithSourceEquivalent (argument_name, **kwargs)`

Bases: `clize.parser.Parameter`

Parameter that relates to a function parameter in the source.

Parameters `argument_name` (*str*) – The name of the parameter.

class `clize.parser.HelperParameter` (***kwargs*)

Bases: `clize.parser.Parameter`

Parameter that doesn't appear in CLI signatures but is used for instance as the `.sticky` attribute of the bound arguments.

class `clize.parser.ParameterWithValue` (*conv=<function identity at 0x7fa10ed4f28>, default=<unset>, **kwargs*)

Bases: `clize.parser.Parameter`

A parameter that takes a value from the arguments, with possible default and/or conversion.

Parameters

- **conv** (*callable*) – A callable to convert the value or raise `ValueError`. Defaults to `identity`.
- **default** – A default value for the parameter or `util.UNSET`.

conv = None

The function used for coercing the value into the desired format or type.

default = None

The default value used for the parameter, or `util.UNSET` if there is no default value. Usually only used for displaying the help.

required

Tells if the parameter has no default value.

coerce_value (*arg, ba*)

Coerces *arg* using the *conv* function. Raises `errors.BadArgumentFormat` if the coercion function raises `ValueError`.

get_value (*ba, i*)

Retrieves the “value” part of the argument in *ba* at position *i*.

help_parens ()

Shows the default value in the parameter description.

clize.parser.value_converter (*func=None, *, name=None*)

Callables decorated with this can be used as a value converter.

See *Specifying a value converter*.

class `clize.parser.NamedParameter` (*aliases, **kwargs*)

Bases: `clize.parser.Parameter`

Equivalent of a keyword-only parameter in Python.

Parameters `aliases` (*sequence of strings*) – The arguments that trigger this parameter.

The first alias is used to refer to the parameter. The first one is picked as *display_name* if unspecified.

aliases = None

The parameter's aliases, eg. “-option” and “-o”.

classmethod alias_key (*name*)

Sort key function to order aliases in source order, but with short forms(one dash) first.

get_all_names ()

Retrieves all aliases.

short_name

Retrieves the shortest alias for displaying the parameter signature.

get_full_name()

Uses the shortest name instead of the display name.

redispatch_short_arg(*rest, ba, i*)

Processes the rest of an argument as if it was a new one prefixed with one dash.

For instance when `-a` is a flag in `-abcd`, the object implementing it will call this to proceed as if `-a -bcd` was passed.

get_value(*ba, i*)

Fetches the value after the `=` (`--opt=val`) or in the next argument (`--opt val`).

class `clize.parser.FlagParameter`(*value, **kwargs*)

Bases: `clize.parser.OptionParameter`

A named parameter that takes no argument.

Parameters

- **value** – The value when the argument is present.
- **false_value** – The value when the argument is given one of the false value triggers using `--param=xyz`.

required = False

false_triggers = ('0', 'n', 'no', 'f', 'false')

Values for which `--flag=X` will consider the argument false and will pass `false_value` to the function. In all other cases `value` is passed.

value = None

The value passed to the function if the flag is triggered without a specified value.

read_argument(*ba, i*)

Overrides `NamedParameter`'s value-getting behavior to allow no argument to be passed after the flag is named.

format_argument(*long_alias*)

class `clize.parser.OptionParameter`(*aliases, **kwargs*)

Bases: `clize.parser.NamedParameter`, `clize.parser.ParameterWithValue`, `clize.parser.ParameterWithSourceEquivalent`

A named parameter that takes an argument.

read_argument(*ba, i*)

Stores the argument in `CliBoundArguments.kwargs` if it isn't already present.

format_type()

Returns a string designation of the value type.

format_argument(*long_alias*)**get_all_names**()

Appends the value type to all aliases.

get_full_name()

Appends the value type to the shortest alias.

class `clize.parser.IntOptionParameter`(*aliases, **kwargs*)

Bases: `clize.parser.OptionParameter`

A named parameter that takes an integer as argument. The short form of it can be chained with the short form of other named parameters.

read_argument (*ba*, *i*)

Handles redispatching after a numerical value.

class `clize.parser.PositionalParameter` (*conv=<function identity at 0x7fda10ed4f28>*, *default=<unset>*, ***kwargs*)

Bases: `clize.parser.ParameterWithValue`, `clize.parser.ParameterWithSourceEquivalent`

Equivalent of a positional-only parameter in Python.

read_argument (*ba*, *i*)

Stores the argument in `CliBoundArguments.args`.

help_parens ()

Puts the value type in parenthesis since it isn't shown in the parameter's signature.

class `clize.parser.MultiParameter` (*min*, *max*, ***kwargs*)

Bases: `clize.parser.ParameterWithValue`

Parameter that can collect multiple values.

min = None

The minimum amount of values this parameter accepts.

max = None

The maximum amount of values this parameter accepts.

required

Returns if there is a minimum amount of values required.

get_collection (*ba*)

Return an object that new values will be appended to.

read_argument (*ba*, *i*)

Adds passed argument to the collection returned by `get_collection`.

apply_generic_flags (*ba*)

Doesn't automatically mark the parameter as satisfied.

unsatisfied (*ba*)

Lets `errors.MissingRequiredArguments` be raised or raises `errors.NotEnoughValues` if arguments were passed but not enough to meet *min*.

get_full_name ()

Adds an elipsis to the parameter name.

class `clize.parser.ExtraPosArgsParameter` (*required=False*, *min=None*, *max=None*, ***kwargs*)

Bases: `clize.parser.MultiParameter`, `clize.parser.PositionalParameter`

Parameter that forwards all remaining positional arguments to the callee.

Used to convert **args*-like parameters.

get_collection (*ba*)

Uses `CliBoundArguments.args` to collect the remaining arguments.

apply_generic_flags (*ba*)

Sets itself as sticky parameter so that `errors.TooManyArguments` is not raised when processing further parameters.

class `clize.parser.AppendArguments` (***kwargs*)

Bases: `clize.parser.HelperParameter`, `clize.parser.MultiParameter`

Helper parameter that collects multiple values to be passed as positional arguments to the callee.

Similar to *ExtraPosArgsParameter* but does not correspond to a parameter in the source.

get_collection (*ba*)

Uses *CliBoundArguments.args* to collect the remaining arguments.

class *clize.parser.IgnoreAllArguments* (***kwargs*)

Bases: *clize.parser.HelperParameter*, *clize.parser.Parameter*

Helper parameter for *FallbackCommandParameter* that ignores the remaining arguments.

read_argument (*ba*, *i*)

Does nothing, ignoring all arguments processed.

class *clize.parser.FallbackCommandParameter* (*func*, ***kwargs*)

Bases: *clize.parser.NamedParameter*

Parameter that sets an alternative function when triggered. When used as an argument other than the first all arguments are discarded.

is_alternate_action = **True**

func = **None**

The function that will be called if this parameter is mentionned.

description

Use *func*'s docstring to provide the parameter description.

read_argument (*ba*, *i*)

Clears all processed arguments, sets up *func* to be called later, and lets all remaining arguments be collected as positional if this was the first argument.

class *clize.parser.AlternateCommandParameter* (*func*, ***kwargs*)

Bases: *clize.parser.FallbackCommandParameter*

Parameter that sets an alternative function when triggered. Cannot be used as any argument but the first.

read_argument (*ba*, *i*)

Raises an error when this parameter is used after other arguments have been given.

4.2.3 Exceptions

exception *clize.errors.UserError*

Bases: *ValueError*

An error to be printed to the user.

exception *clize.errors.ArgumentError* (*message=None*)

Bases: *clize.errors.UserError*

An error related to the arguments passed through the command-line interface

exception *clize.errors.MissingRequiredArguments* (*missing*)

Bases: *clize.errors.ArgumentError*

Raised when required parameters have not been provided an argument

exception *clize.errors.TooManyArguments* (*extra*)

Bases: *clize.errors.ArgumentError*

Raised when too many positional arguments have been passed for the parameters to consume.

exception `clize.errors.DuplicateNamedArgument` (*message=None*)

Bases: `clize.errors.ArgumentError`

Raised when a named option or flag has been passed twice.

exception `clize.errors.UnknownOption` (*name*)

Bases: `clize.errors.ArgumentError`

Raised when a named argument has no matching parameter.

exception `clize.errors.MissingValue` (*message=None*)

Bases: `clize.errors.ArgumentError`

Raised when an option received no value.

exception `clize.errors.NotEnoughValues` (*message=None*)

Bases: `clize.errors.ArgumentError`

Raised when `MultiOptionParameter` is given less values than its min parameter.

exception `clize.errors.TooManyValues` (*message=None*)

Bases: `clize.errors.ArgumentError`

Raised when `MultiOptionParameter` is given more values than its max parameter.

exception `clize.errors.CliValueError`

Bases: `ValueError`

Specialization of `ValueError` for showing a message to the user along with the error rather than just the incorrect value.

exception `clize.errors.BadArgumentFormat` (*text*)

Bases: `clize.errors.ArgumentError`

Raised when an argument cannot be converted to the correct format.

exception `clize.errors.ArgsBeforeAlternateCommand` (*param*)

Bases: `clize.errors.ArgumentError`

Raised when there are arguments before a non-fallback alternate command.

class `clize.errors.SetErrorContext` (*exc_type, **attributes*)

Bases: `object`

Context manager that sets attributes on exceptions that are raised past it

Parameters

- **exc_type** – The exception type to operate on.
- **attributes** – The attributes to set on the matching exceptions. They will only be set if yet unset on the exception.

4.2.4 Compatibility with older clize releases

`clize.clize` (*fn=None, *, alias={}, force_positional=(), coerce={}, require_excess=False, extra=(), use_kwoargs=None*)

Compatibility with `clize<3.0` releases. Decorates a function in order to be passed to `clize.run`. See *Upgrading from clize 1 and 2*.

`clize.make_flag` (*source, names, default=False, type=None, help='', takes_argument=0*)

Compatibility with `clize<3.0` releases. Creates a parameter instance. See *Upgrading from clize 1 and 2*.

Project documentation

Information on how Clize is organized as a project.

5.1 Release notes

5.1.1 3.0 (2015-05-13)

Version 3.0 packs a full rewrite. While it retains backwards-compatibility, the old interface is deprecated. See *Upgrading from clize 1 and 2*.

- The argument parsing logic has been split between a loop over the parameters and parameter classes. New parameter classes can be made to implement custom kinds of parameters.
- The CLI inference is now based on `sigtools.specifiers.signature` rather than `inspect.getfullargspec`. This enables a common interface for the function signature to be manipulated prior to being passed to Clize. Namely, the `__signature__` attribute can be overridden or `sigtools`'s lazy `forger_function` method can be employed.
- The `@clize` decorator is deprecated in favor of directly passing functions to `run`, thus leaving the original function intact.
- Named parameters are now obtained exclusively through keyword-only parameters. Other information about each parameter is communicated through parameter annotations. `sigtools.modifiers` provides backwards-compatibility for Python 2.
- As a result of implementing the function signature-based abstraction, there are *ways to set up decorators that work with Clize*.
- The help system now accepts *subsection headers for named parameters*.
- *Coercion functions* have been renamed to *value converters*. Except for a few notable exceptions, they must be *tagged with a decorator*. This also applies to the type of supplied default values.
- *Alternate actions* (for instance `--version`) can be supplied directly to `run`.
- Several *Parameter converter* annotations have been added, including parameters with a limited choice of values, repeatable parameters, parameters that always supply the same value, and more.

5.1.2 2.0 (2012-10-07)

This release and earlier were documented post-release.

Version 2.0 adds subcommands and support for function annotations and keyword-only parameters.

5.1.3 1.0 (2011-04-04)

Initial release.

5.2 Contributing

Thanks for considering helping out. We don't bite. :-)

5.2.1 Reporting issues

Bugs and other tasks are tracked on GitHub.

- Check whether the issue exists already. (Be sure to also check closed ones.)
- Report which version of Clize the issue appears on. You can obtain it using:

```
pip show clize
```

For documentation-related bugs, you can either look at the version in the page URL, click the “Read the docs” insignia in the bottom-left corner or the hamburger menu on mobile.

- When applicable, show how to trigger the bug and what was expected instead. Writing a testcase for it is welcome, but not required.

5.2.2 Submitting patches

Patches are submitted for review through GitHub pull requests.

- Follow [PEP 8](#).
- When fixing a bug, include a test case in your patch. Make sure correctly tests against the bug: It must fail without your fix, and succeed with it. See [Running the test suite](#).
- Submitting a pull request on GitHub implies your consent for merging, therefore authorizing the maintainer(s) to distribute your modifications under the project's license.

5.2.3 Implementing new features

Before implementing a new feature, please open an issue on GitHub to discuss it. This ensures you do not work on a feature that would be refused inclusion.

Add tests for your feature to the test suite and make sure it *completes on all supported versions of Python*. Make sure it is fully tested using the `cover` target.

Feel free to submit a pull request as soon as you have changes you need feedback on. In addition, TravisCI will run the test suite on all supported platforms and will perform coverage checking for you on the pull request page.

5.2.4 Running the test suite

The test suite can be run across all supported versions using, `tox`:

```
pip install --user tox
tox
```

If you do not have all required Python versions installed or wish to save time when testing you can specify one version of Python to test against:

```
tox -e pyXY
```

Where `X` and `Y` designate a Python version as in `X.Y`. For instance, the following command runs the test suite against Python 3.4 only:

```
tox -e py34
```

Branches linked in a pull request will be run through the test suite on TravisCI and the results are linked back in the pull request. You can use this if installing all supported Python versions is impractical for you.

`coverage.py` is used to measure code coverage. New code is expected to have full code coverage. You can run the test suite through it using:

```
tox -e cover
```

It will print the measured code coverage and generate webpages with line-by-line coverage information in `htmlcov`. Note that the `cover` target requires Python 3.4.

5.2.5 Documentation

The documentation is written using `sphinx` and lives in `docs/` from the project root. It can be built using:

```
tox -e docs
```

This will produce documentation in `build/sphinx/html/`. Note that Python 3.4 must be installed to build the documentation.

C

`clize.errors`, 58
`clize.legacy`, 59
`clize.parser`, 25
`clize.runner`, 49

A

`alias_key()` (clize.parser.NamedParameter class method), 55
`aliases` (clize.parser.CliSignature attribute), 51
`aliases` (clize.parser.NamedParameter attribute), 55
`alternate` (clize.parser.CliSignature attribute), 51
`AlternateCommandParameter` (class in clize.parser), 58
`AppendArguments` (class in clize.parser), 57
`apply_generic_flags()` (clize.parser.ExtraPosArgsParameter method), 57
`apply_generic_flags()` (clize.parser.MultiParameter method), 57
`apply_generic_flags()` (clize.parser.Parameter method), 27, 54
`args` (clize.parser.CliBoundArguments attribute), 53
`ArgsBeforeAlternateCommand`, 59
`argument_decorator()` (in module clize.parameters), 42
`ArgumentError`, 58
`as_is()` (clize.Clize class method), 50

B

`BadArgumentFormat`, 59

C

`cli` (clize.Clize attribute), 50
`cli` (clize.SubcommandDispatcher attribute), 51
`CliBoundArguments` (class in clize.parser), 52
`CliSignature` (class in clize.parser), 51
`CliValueError`, 59
`Clize` (class in clize), 50
`clize()` (in module clize), 59
`clize.errors` (module), 58
`clize.legacy` (module), 59
`clize.parser` (module), 25, 51
`clize.runner` (module), 49
`clizer` (clize.SubcommandDispatcher attribute), 51
`code examples`, 49
`coerce_value()` (clize.parser.ParameterWithValue method), 55
`conv` (clize.parser.ParameterWithValue attribute), 55

`convert_parameter()` (clize.parser.CliSignature class method), 51
`converter` (clize.parser.CliSignature attribute), 51

D

`default` (clize.parser.ParameterWithValue attribute), 55
`default value`, 36
`default_converter()` (in module clize.parser), 52
`description` (clize.parser.FallbackCommandParameter attribute), 58
`display_name` (clize.parser.Parameter attribute), 54
`DRY`, 10
`DuplicateNamedArgument`, 58

E

`ExtraPosArgsParameter` (class in clize.parser), 57
`extras` (clize.parser.Parameter attribute), 54

F

`FallbackCommandParameter` (class in clize.parser), 58
`false_triggers` (clize.parser.FlagParameter attribute), 56
`FlagParameter` (class in clize.parser), 56
`format_argument()` (clize.parser.FlagParameter method), 56
`format_argument()` (clize.parser.OptionParameter method), 56
`format_type()` (clize.parser.OptionParameter method), 56
`from_signature()` (clize.parser.CliSignature class method), 51
`func` (clize.parser.CliBoundArguments attribute), 53
`func` (clize.parser.FallbackCommandParameter attribute), 58

G

`get_all_names()` (clize.parser.NamedParameter method), 55
`get_all_names()` (clize.parser.OptionParameter method), 56
`get_all_names()` (clize.parser.Parameter method), 54
`get_cli()` (clize.Clize class method), 50

`get_collection()` (clize.parser.AppendArguments method), 58

`get_collection()` (clize.parser.ExtraPosArgsParameter method), 57

`get_collection()` (clize.parser.MultiParameter method), 57

`get_full_name()` (clize.parser.MultiParameter method), 57

`get_full_name()` (clize.parser.NamedParameter method), 56

`get_full_name()` (clize.parser.OptionParameter method), 56

`get_full_name()` (clize.parser.Parameter method), 54

`get_value()` (clize.parser.NamedParameter method), 56

`get_value()` (clize.parser.ParameterWithValue method), 55

H

`help_parens()` (clize.parser.Parameter method), 54

`help_parens()` (clize.parser.ParameterWithValue method), 55

`help_parens()` (clize.parser.PositionalParameter method), 57

`helper` (clize.Clize attribute), 50

`HelperParameter` (class in clize.parser), 55

I

`IGNORE` (clize.parser.Parameter attribute), 43, 53

`IgnoreAllArguments` (class in clize.parser), 58

`in_args` (clize.parser.CliBoundArguments attribute), 52

`IntOptionParameter` (class in clize.parser), 56

`is_alternate_action` (clize.parser.FallbackCommandParameter attribute), 58

`is_alternate_action` (clize.parser.Parameter attribute), 53

K

`keep()` (clize.Clize class method), 50

`kwargs` (clize.parser.CliBoundArguments attribute), 53

L

`LAST_OPTION` (clize.parser.Parameter attribute), 44, 53

`last_option` (clize.parser.Parameter attribute), 54

M

`make_flag()` (in module clize), 59

`mapped()` (in module clize.parameters), 40

`max` (clize.parser.MultiParameter attribute), 57

`meta` (clize.parser.CliBoundArguments attribute), 53

`min` (clize.parser.MultiParameter attribute), 57

`MissingRequiredArguments`, 58

`MissingValue`, 59

`multi()` (in module clize.parameters), 41

`MultiParameter` (class in clize.parser), 57

N

`name` (clize.parser.CliBoundArguments attribute), 52

`named` (clize.parser.CliSignature attribute), 51

`NamedParameter` (class in clize.parser), 55

`NotEnoughValues`, 59

O

`one_of()` (in module clize.parameters), 41

`OptionParameter` (class in clize.parser), 56

P

`Parameter` (class in clize.parser), 53

`parameter converter`, 42

`parameter_converter()` (in module clize.parser), 51

`parameters` (clize.parser.CliSignature attribute), 51

`parameters()` (clize.Clize method), 50

`ParameterWithSourceEquivalent` (class in clize.parser), 54

`ParameterWithValue` (class in clize.parser), 55

`pass_name()` (in module clize.parameters), 44

`posarg_only` (clize.parser.CliBoundArguments attribute), 53

`positional` (clize.parser.CliSignature attribute), 51

`PositionalParameter` (class in clize.parser), 57

`posparam` (clize.parser.CliBoundArguments attribute), 53

`post_name` (clize.parser.CliBoundArguments attribute), 53

`post_parse()` (clize.parser.Parameter method), 54

`prepare_help()` (clize.parser.Parameter method), 54

Python Enhancement Proposals

PEP 3102, 7

PEP 3107, 6

PEP 8, 62

R

`read_argument()` (clize.parser.AlternateCommandParameter method), 58

`read_argument()` (clize.parser.FallbackCommandParameter method), 58

`read_argument()` (clize.parser.FlagParameter method), 56

`read_argument()` (clize.parser.IgnoreAllArguments method), 58

`read_argument()` (clize.parser.IntOptionParameter method), 57

`read_argument()` (clize.parser.MultiParameter method), 57

`read_argument()` (clize.parser.OptionParameter method), 56

`read_argument()` (clize.parser.Parameter method), 26, 54

`read_argument()` (clize.parser.PositionalParameter method), 57

`read_arguments()` (clize.parser.CliSignature method), 51

`read_commandline()` (clize.Clize method), 50

`redispatch_short_arg()` (clize.parser.NamedParameter method), 56

`required` (clize.parser.CliSignature attribute), 51

required (clize.parser.FlagParameter attribute), [56](#)
required (clize.parser.MultiParameter attribute), [57](#)
REQUIRED (clize.parser.Parameter attribute), [37](#), [53](#)
required (clize.parser.Parameter attribute), [53](#)
required (clize.parser.ParameterWithValue attribute), [55](#)
run() (in module clize), [49](#)

S

SetErrorContext (class in clize.errors), [59](#)
short_name (clize.parser.NamedParameter attribute), [55](#)
show_help() (clize.parser.Parameter method), [54](#)
show_help_parens() (clize.parser.Parameter method), [54](#)
sig (clize.parser.CliBoundArguments attribute), [52](#)
signature (clize.Clize attribute), [50](#)
skip (clize.parser.CliBoundArguments attribute), [53](#)
sticky (clize.parser.CliBoundArguments attribute), [53](#)
SubcommandDispatcher (class in clize), [51](#)

T

TooManyArguments, [58](#)
TooManyValues, [59](#)

U

UNDOCUMENTED (clize.parser.Parameter attribute),
[44](#), [53](#)
undocumented (clize.parser.Parameter attribute), [54](#)
UnknownOption, [59](#)
unsatisfied (clize.parser.CliBoundArguments attribute),
[53](#)
unsatisfied() (clize.parser.MultiParameter method), [57](#)
unsatisfied() (clize.parser.Parameter method), [54](#)
use_class() (in module clize.parser), [52](#)
use_mixin() (in module clize.parser), [52](#)
UserError, [58](#)

V

value (clize.parser.FlagParameter attribute), [56](#)
value conversion, [36](#)
value converter, [36](#)
value_converter() (in module clize.parser), [55](#)
value_inserter() (in module clize.parameters), [45](#)