clik

Release 0.92.5

Joe Joyce and contributors

May 23, 2019

Contents

Clik is a Python library for writing complex command-line applications with a minimum of boilerplate. Clik gives you access to the full power of `argparse`[1], but saves you the trouble of creating and managing parsers, subparsers, and all the glue in between.

To get started, take a look at the intro section linked below.

[1] https://docs.python.org/3/library/argparse.html#module-argparse

CHAPTER 1

Intro

The introductory materials revolve around a toy command-line application for managing a todo list.

The tutorial walks through development of the application step-by-step, explaining clik concepts along the way.

The screencast has roughly the same content as the tutorial but is, of course, in video form.

For a *very* quick overview of clik, take a look at the example listing linked below. It's the final code from the tutorial and, while it may not make perfect sense, it should give you a good idea of what clik applications look like.

1.1 Example

Following is the full code listing from the final step of the tutorial. See *the design sketch* (page 6) for what this code is meant to accomplish.

```
#!bin/python
# -*- coding: utf-8; mode: python -*-
import json
import os
import sys

from clik import app, args, g, parser


def print_list():
    for i, item in enumerate(g.item_list):
        print('%i. %s' % (i, item))


@app
def todo():
    """
    Simple application for managing a todo list.
```

(continues on next page)

(continued from previous page)

```
    The data is stored as a JSON file, which defaults to todo.json.
    See above for more information about arguments and subcommands.
    """
    parser.add_argument(
        '-f',
        '--file',
        default='todo.json',
        help='file in which to store data (default: %(default)s)',
    )

    yield

    g.item_list = []
    if os.path.exists(args.file):
        with open(args.file) as f:
            g.item_list = json.load(f)

    yield

    with open(args.file, 'w') as f:
        json.dump(g.item_list, f, indent=2)
        f.write('\n')


@todo.bare
def bare():
    yield
    print_list()


@todo
def add():
    """Add an item to the todo list."""
    parser.add_argument(
        'item',
        help='item to add to do the todo list',
        nargs='?',
    )

    yield

    item = args.item
    if item is None:
        item = input('Item to add: ')
    if item:
        g.item_list.append(item)
    else:
        print('error: empty item', file=sys.stderr)
        yield 1

    print()
    print('Updated list:')
    print_list()


@todo(name='list', alias='ls')
def list_():
```

(continues on next page)

(continued from previous page)

```
    """Show the items on the todo list."""
    yield
    print_list()


@todo
def done():
    """Remove an item from the todo list."""
    group = parser.add_mutually_exclusive_group()
    group.add_argument(
        '-a',
        '--all',
        action='store_true',
        default=False,
        help='mark all items as complete',
    )
    group.add_argument(
        '-i',
        '--index',
        help='integer index of the item to mark as complete',
        type=int,
    )

    yield

    if args.all:
        del g.item_list[:]
    else:
        index = args.index
        while index is None:
            print()
            print_list()
            print()
            selection = input('Index of item to delete? ')
            try:
                index = int(selection)
            except ValueError:
                print('error: invalid int value:', selection, file=sys.stderr)
        if -1 < index < len(g.item_list):
            del g.item_list[index]
        else:
            print('error: index out of bounds:', index, file=sys.stderr)
            yield 1

    print()
    print('Updated list:')
    print_list()


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

1.2 Screencast

1.3 Tutorial

The tutorial covers the step-by-step development of a toy command-line application for managing a todo list. (Original, I know.) Reading time is meant to be under an hour, assuming you play along with some of the examples. There is also *a screencast* (page 6) that roughly follows the tutorial, if videos are more your speed.

1.3.1 Tutorial 00: Design

Before writing any code, let's make a rough sketch of what the application should do.

In prose: `todo` should allow the end user to manage a todo list. Items may be added and deleted, and the current list may be displayed.

We'll keep data storage simple. The underlying data structure will be a list of strings. We'll use Python's built-in JSON module to persist the data to a file.

Roughing out the interface:

```
todo [-f/--file=todo.json] <COMMAND AND ARGS ...>
todo add ["item to add"]        # prompt if item not given
todo list                       # print a 0-indexed list of items
todo done [-i/--index|-a/--all] # if -a/--all given, all items are done
                                # else if 0-based index supplied,
                                #   delete item at that index
                                # else print list and prompt for index
```

The final implementation may not look exactly like this, but it's a good place to start!

Next, let's *set up the environment* (page 6) for the app.

1.3.2 Tutorial 01: Setup

For the purposes of the tutorial, we'll create a new virtual environment, install the `clik` package, and put the application code inside the environment's directory. For the hashbang line of the app, we'll "cheat" and use a relative path to the virtualenv's Python interpreter (so it has access to `clik`). If you were developing a "real" end-user application, you would probably want to use a proper package structure.

This means the demo app always has to be called from the working directory that contains it (i.e. `./todo.py <...>`). If you really need to run this from other working directories, change the `bin/python` in the first line of the file to an absolute path to the environment's Python interpreter.

The following commands create the basic structure:

```
$ virtualenv todo
# ... snip ...
$ cd todo
$ bin/pip install clik
# ... snip ...
$ touch todo.py
$ chmod +x todo.py
```

Open the `todo.py` file and edit it to contain:

```
#!bin/python
import clik
print('Hello, world!')
```

Now you should be able to run the script:

```
$ ./todo.py
Hello, world!
$
```

If there are no errors, cool! Everything is working. In the next step we'll tweak the file to *use the if-name-main pattern* (page 7) common for Python executables.

1.3.3 Tutorial 02: Main

The next step is to structure our file the "standard" way for Python files that are used as executables. Note that the `import clik` statement is gone; now that we've tested that we can import it, we don't need it for the moment. We'll bring it back in a couple steps.

```
#!bin/python

if __name__ == '__main__':
    print('Hello, world!')
```

`__name__` is a special variable that contains the name of the current module as a string. This is useful because Python lets a file be both "`import` -able" and executable. If we were to import the `todo.py` file from another module, `__name__` would be equal to `"todo"`.

When run directly, however, `__name__` is set to the special value `"__main__"`. By checking for that special value and running our code only when it's set, it allows our file to "do the right thing" whether it's imported or run directly:

```
$ ./todo.py
Hello, world!
$ python
# ... snip ...
>>> import todo
>>> # note that "hello world" is not printed above
>>> exit()
$
```

For the demo application, this doesn't matter very much – it is meant to always be run directly. And in fact, in the next step we'll make a change that makes it unimportable. Using this pattern is still good form, however, since it's an obvious marker for how the file is intended to be used (i.e. as an executable).

In the next step, we'll *remove the .py extension* (page 7) to make the app more caller-friendly, then we'll finally be ready to get into clik proper!

1.3.4 Tutorial 03: Filename

Personally I don't like the `.py` extension for CLI apps. And clik doesn't care. So the demo app will be renamed to just `todo`.

Since the filename no longer has an extension, we'll add a modeline to keep it editor-friendly. We'll add the encoding specifier as well, since that is good practice in general:

```
#!bin/python
# -*- coding: utf-8; mode: python -*-

if __name__ == '__main__':
    print('Hello, world!')
```

Renaming and running the app should produce the same results as before:

```
$ mv todo.py todo
$ ./todo
Hello, world!
$
```

Excellent. Now we can *start getting into clik proper* (page 8).

1.3.5 Tutorial 04: App

Note: This step is a bit longer and more involved than the others because it goes over some fundamentals that make subsequent steps smaller and easier to explain.

With all the "formalities" out of the way, we can turn the Python script into a clik app:

```
#!bin/python
# -*- coding: utf-8; mode: python -*-
from clik import app

@app
def todo():
    yield
    print('Hello, world!')

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

Running the program with no arguments produces the same results as before. If we run it with `-h` or `--help`, though, the app now has a help message! And if we run it with any other arguments, we get an error message indicating that our program is not expecting them:

```
$ ./todo
Hello, world!

$ ./todo -h
usage: todo [-h]

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

$ ./todo --help
usage: todo [-h]

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

$ ./todo --foo
```

(continues on next page)

(continued from previous page)

```
usage: todo [-h]
todo: error: unrecognized arguments: --foo

$ echo $?
1
```

Looking at the new code chunk by chunk:

```
from clik import app
```

First the `app` decorator is imported. All the clik interfaces you will interact with will be imported from the top-level `clik` package.

The `app` decorator tells clik what to call when your application is invoked. Here, we tell clik to call `todo` when the end user runs the application:

```
@app
def todo():
    # ... snip ...
```

The `app` decorator also controls the name of the application as seen in the usage and other automatically generated help messages. By default, `app` uses the name of the thing being decorated – `todo` in this case. Had we called the function something different:

```
@app
def my_todo_app():
   # ... snip ...

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

the help output would have changed accordingly:

```
$ ./todo -h
usage: my_todo_app [-h]

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

The name can also be manually specified using the `name` parameter to the `app` decorator:

```
@app(name='supercool-todo-app')
def todo():
    # ... snip ...
```

```
$ ./todo -h
usage: supercool-todo-app [-h]

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

Typically it's not necessary to manually specify `name` for the `app` decorator. But later on, when implementing subcommands, the `name` parameter makes another appearance, and is extremely useful. (Think `./todo list` – we probably don't want to define our own function named `list` since that is a *very* core built-in. . . .)

Next let's look at the function body:

```
def todo():
    yield
    print('Hello, world!')
```

Technically, `todo` is a generator – not a function. At a lower level this is an important distinction, but for our purposes it doesn't much matter. What matters are the two "rules" that being a generator implies:

1. Every clik-decorated function must have at least one `yield` statement.
2. You cannot call clik-decorated functions directly. Well, you can, but it's virtually guaranteed to do gnarly and unexpected things. Just don't do it.

In terms of program design, the second rule has important implications. clik programs usually have two layers: an internal API layer that is responsible for reading / writing / working on data and a UI layer that uses the internal API and clik to implement the end-user interface. The internal API shouldn't "know" about clik at all. (And in the spirit of tutorials everywhere, this advice will be promptly eschewed because for our demo app the logic will be simple enough to not warrant any kind of internal API.)

Within our function, there are three phases of execution:

```
def todo():
    # configure argument parser
    yield  # give control back to clik, which parses end user arguments
    # do something with parsed arguments
```

Right now our program has no arguments, so there's no code in the "configure parser" phase. clik still parses end user arguments (this is where it handles `-h` or errors out on unknown args). And the "do something" phase is where we print "hello world."

The last bit of code kicks off the whole process:

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

By default, the `main` method invokes the application with the arguments from `sys.argv`, then calls `sys.exit` with the exit code from the application. You can override these by supplying the `argv` and `exit` arguments, respectively. (This more advanced usage will not be covered in the tutorial. These arguments are mainly provided for testing purposes – to allow test code to invoke the app with a given set of arguments but not have it exit the process upon completion.)

Phew, That was a lot of words for twelve lines of code! Let's take a breather and *add some help text* (page 10) before we dive into arguments.

1.3.6 Tutorial 05: Help

Adding help text to the app is easy. Just add a docstring:

```
@app
def todo():
    """
    Command-line application for managing a todo list.

    The list is stored on disk as a simple JSON file containing an
    array of strings. The file path is controlled by the -f/--file
    argument (see documentation for that argument for more
    information).
    """
```

(continues on next page)

(continued from previous page)

```
    yield
    print('Hello, world!')
```

In argparse terms, the content before the first blank line is the `description` and all content after is the `epilog`:

```
$ ./todo -h
usage: todo [-h]

Command-line application for managing a todo list.

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

The list is stored on disk as a simple JSON file containing an array of
strings. The file path is controlled by the -f/--file argument (see
documentation for that argument for more information).
```

Easy peasy. Let's *implement the file argument* (page 11) described in the help text.

1.3.7 Tutorial 06: Arguments

clik provides two "magic" variables for configuring and accessing arguments: the aptly-named `parser` and `args`:

```
from clik import app, args, parser

@app
def todo():
    # ... snip ...
    parser.add_argument(
        '-f',
        '--file',
        default='todo.json',
        help='file in which to store data (default: %(default)s)',
    )
    yield
    print('File path is:', args.file)
```

`parser` is an `argparse.ArgumentParser`[2]. `args` is an `argparse.Namespace`[3] – the same thing you would get back from `argparse.ArgumentParser.parse_args()`[4].

This might remind you of the "execution phases" from step 4, which should make more sense now:

```
def todo():
    # configure argument parser
    yield  # give control back to clik, which parses end user arguments
    # do something with parsed arguments
```

The behavior of the application is what you probably expect:

```
$ ./todo -h
usage: todo [-h] [-f FILE]

```

(continues on next page)

[2] https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser
[3] https://docs.python.org/3/library/argparse.html#argparse.Namespace
[4] https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser.parse_args

(continued from previous page)

```
Command-line application for managing a todo list.

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  file in which to store data (default: todo.json)

The list is stored on disk as a simple JSON file containing an array of
strings. The file path is controlled by the -f/--file argument (see
documentation for that argument for more information).

$ ./todo
File is: todo.json

$ ./todo -f myfile.json
File is: myfile.json

$ ./todo -f=myfile.json
File is: myfile.json

$ ./todo -fmyfile.json
File is: myfile.json

$ ./todo --file myfile.json
File is: myfile.json

$ ./todo --file=myfile.json
File is: myfile.json
```

Of course, the code before the `yield` is not limited to simple calls to `add_argument`. It's just arbitrary Python code. As a silly example:

```
from datetime import datetime
from clik import app, args, parser

@app
def todo():
    # If it's 6PM or later, default to the "nighttime list,"
    # otherwise default to the "daytime list."
    if datetime.today().time().hour > 17:
        default = 'night.json'
    else:
        default = 'day.json'
    parser.add_argument(
        '-f',
        '--file',
        default=default,
        help='file in which to store data (default: %(default)s)',
    )
    yield
    print('File path is:', args.file)
```

As is often the case, with great power comes great responsibility. Code before the `yield` is run on every invocation of the program...

- ...regardless of whether the arguments are valid or not
- ...even if `-h/--help` is specified

- . . . or, in the case of subcommands, even if the subcommand is not called!

In other words: **don't do expensive things before the yield** or your program will feel/be unresponsive. (The Python interpreter startup time is bad enough.)

To finish this step, let's make the argument "do" something:

```
#!bin/python
# -*- coding: utf-8; mode: python -*-
import json
import os

from clik import app, args, parser


@app
def todo():
    """
    Command-line application for managing a todo list.

    The list is stored on disk as a simple JSON file containing an
    array of strings. The file path is controlled by the -f/--file
    argument (see documentation for that argument for more
    information).
    """
    parser.add_argument(
        '-f',
        '--file',
        default='todo.json',
        help='file in which to store data (default: %(default)s)',
    )

    yield

    item_list = []
    if os.path.exists(args.file):
        with open(args.file) as f:
            item_list = json.load(f)

    for item in item_list:
        print('*', item)


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

Assuming a `test.json` file with the following contents. . .

```
[
  "Pick up nails from hardware store",
  "Grab milk from the grocery",
  "Clean up the kitchen",
  "Feed the cats"
]
```

. . . the application can now print out the items in the todo list:

```
$ ./todo
$ ./todo -f test.json
```

(continues on next page)

(continued from previous page)

```
* Pick up nails from hardware store
* Grab milk from the grocery
* Clean up the kitchen
* Feed the cats
$
```

Next we'll get into the thick of what makes clik useful, and start implementing the interface we designed way back in step 0! *Get ready for subcommands!* (page 14)

1.3.8 Tutorial 07: Subcommands

Note: This step is a bit longer and more involved than the others because it tries to tie together the core concepts in clik, and emphasize the pattern that underlies the library.

The important code updates are all in the first listing. The rest of the step explains what's going on using silly, verbose examples for the sake of illustration.

To this point, everything we have done would be just as easy using stock argparse. clik really starts to shine when we introduce subcommands:

```
@app
def todo():
    # ... snip ...

    # The following lines have been deleted from the example.
    # for item in item_list:
    #     print('*', item)

@todo
def add():
    """Add an item to the list."""
    yield
    print('hello from add')

@todo(name='list')
def list_():
    """Show the items on the list."""
    yield
    print('hello from list')

@todo
def done():
    """Remove an item from the list."""
    yield
    print('hello from done')
```

Poking around at the application:

```
$ ./todo -h
usage: todo [-h] [-f FILE] {add,list,done} ...

Command-line application for managing a todo list.
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  file in which to store data (default: todo.json)

subcommands:
  {add,list,done}
    add                 Add an item to the list.
    list                Show the items on the list.
    done                Remove an item from the list.

The list is stored on disk as a simple JSON file containing an array of
strings. The file path is controlled by the -f/--file argument (see
documentation for that argument for more information).

$ ./todo
usage: todo [-h] [-f FILE] {add,list,done} ...
todo: error: the following arguments are required: {add,list,done}

$ ./todo add -h
usage: todo add [-h]

Add an item to the list.

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

$ ./todo add
hello from add

$ ./todo list -h
usage: todo list [-h]

Show the items on the list.

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

$ ./todo list
hello from list

$ ./todo done -h
usage: todo done [-h]

Remove an item from the list.

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

$ ./todo done
hello from done
```

Neat-o! Gluing all that together with argparse would have been straightforward, but would have involved quite a bit of ceremony and boilerplate.

Subcommands look a lot like the `app` we've been working on to this point. (There is a reason for this – under the covers they're actually the same thing. *clik.app.App* (page 39) is a subclass of *clik.command.Command* (page 41)!)

The function decorated by `app` (`todo` in our case) can itself be used as a decorator to register a subcommand:

```
@todo
def xyz():
    # ... do subcommand stuff ...
```

Note: Once a single subcommand has been registered, it is no longer valid for end users to invoke the application without a subcommand. (Unless a "bare" subcommand has been registered – more on that later.)

Subcommands can also be used as decorators to register sub-subcommands. It's "turtles all the way down." An example, with a slew of dummy sub- (and sub-sub- and sub-sub-sub-) commands:

```
@todo
def foo():
    yield

@foo
def spam():
    yield
    print('hai from foo spam')

@foo
def ham():
    yield
    print('hai from foo ham')

@foo
def eggs():
    yield

@eggs
def alpha():
    yield
    print('hai from foo eggs alpha')

@eggs
def bravo():
    yield
    print('hai from foo eggs bravo')

@eggs
def charlie():
    yield
    print('hai from foo eggs charlie')
```

Poking around in the shell:

```
$ ./todo foo -h
usage: todo foo [-h] {spam,ham,eggs} ...

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

subcommands:
  {spam,ham,eggs}
    spam
```

(continues on next page)

(continued from previous page)

```
    ham
    eggs

$ ./todo foo
usage: todo foo [-h] {spam,ham,eggs} ...
todo foo: error: the following arguments are required: {spam,ham,eggs}

$ ./todo foo spam
hai from foo spam

$ ./todo foo ham
hai from foo ham

$ ./todo foo eggs
usage: todo foo eggs [-h] {alpha,bravo,charlie} ...
todo foo eggs: error: the following arguments are required: {alpha,bravo,charlie}

$ ./todo foo eggs -h
usage: todo foo eggs [-h] {alpha,bravo,charlie} ...

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

subcommands:
  {alpha,bravo,charlie}
    alpha
    bravo
    charlie

$ ./todo foo eggs alpha -h
usage: todo foo eggs alpha [-h]

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

$ ./todo foo eggs alpha
hai from foo eggs alpha

$ ./todo foo eggs bravo
hai from foo eggs bravo

$ ./todo foo eggs charlie
hai from foo eggs charlie
```

Like the `app`, the name for the subcommand defaults to the name of the function being decorated and can be overridden by passing the `name` parameter to the decorator.

This is useful for our `list` command since it's a bad idea to redefine built-in functions (which `list` is). We use `list_` as the function name, and pass `"list"` to clik as the name *it* should use:

```
@todo(name='list')
def list_():
    """Show the items on the list."""
    yield
    print('hello from list')
```

The app, of course, works the same as it did before:

```
$ ./todo -h
usage: todo [-h] [-f FILE] {add,list,done} ...

Command-line application for managing a todo list.

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  file in which to store data (default: todo.json)

subcommands:
  {add,list,done}
    add                 Add an item to the list.
    list                Show the items on the list.
    done                Remove an item from the list.

The list is stored on disk as a simple JSON file containing an array of
strings. The file path is controlled by the -f/--file argument (see
documentation for that argument for more information).

$ ./todo list -h
usage: todo list [-h]

Show the items on the list.

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

$ ./todo list
hello from list
```

As you've probably noticed, help messages are taken from docstrings. Like the `app`, content before the blank line is the `description` and everything after is the `epilog`. As an example, let's "lorem ipsum" the help for `add`:

```
@todo
def add():
    """
    Add an item to the list.

    Lorem ipsum dolor sit amet, consectetur adipiscing elit. In
    congue porttitor ornare. Aenean ac diam ipsum. Sed sit amet
    libero ut ligula pretium consectetur eu quis justo. Integer
    sollicitudin velit et nunc suscipit laoreet.
    """
    yield
    print('hello from add')
```

Predictably, the help text is:

```
$ ./todo -h
usage: todo [-h] [-f FILE] {add,list,done} ...

Command-line application for managing a todo list.

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  file in which to store data (default: todo.json)
```

(continues on next page)

(continued from previous page)

```
subcommands:
  {add,list,done}
    add                 Add an item to the list.
    list                Show the items on the list.
    done                Remove an item from the list.

The list is stored on disk as a simple JSON file containing an array of
strings. The file path is controlled by the -f/--file argument (see
documentation for that argument for more information).

$ ./todo add -h
usage: todo add [-h]

Add an item to the list.

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

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In congue porttitor
ornare. Aenean ac diam ipsum. Sed sit amet libero ut ligula pretium
consectetur eu quis justo. Integer sollicitudin velit et nunc suscipit
laoreet.
```

Nice! We are moving right along. Next we'll take *a quick look at aliases* (page 19) before circling back to arguments for our subcommands.

1.3.9 Tutorial 08: Aliases

For discoverability, it's always a good idea to give your commands descriptive names. For commands that are commonly used, though, this can be a burden on end users. clik allows you to define aliases for these commands, giving your application the best of both worlds: discoverability for new users and concision for power users.

Let's say the end users of our todo program are grumpy systems administrators that are used to typing `ls` instead of `list`. The extra `i` and `t` are causing a serious problem for todo-related productivity.

An alias makes everyone happy:

```
@todo(name='list', alias='ls')
def list_():
    """Show the items on the list."""
    yield
    print('hello from list')
```

In the shell:

```
$ ./todo -h
usage: todo [-h] [-f FILE] {add,list,done} ...

Command-line application for managing a todo list.

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  file in which to store data (default: todo.json)

subcommands:
  {add,list,done}
```

(continues on next page)

(continued from previous page)

```
    add                 Add an item to the list.
    list (ls)           Show the items on the list.
    done                Remove an item from the list.

The list is stored on disk as a simple JSON file containing an array of
strings. The file path is controlled by the -f/--file argument (see
documentation for that argument for more information).

$ ./todo list -h
usage: todo list [-h]

Show the items on the list.

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

$ ./todo ls -h
usage: todo list [-h]

Show the items on the list.

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

$ ./todo list
hello from list

$ ./todo ls
hello from list
```

If a command has multiple aliases, supply the `aliases` argument instead. For example, if we wanted `list` to be aliased to `ls` and `l`:

```
@todo(name='list', aliases=('ls', 'l'))
def list_():
    """Show the items on the list."""
    yield
    print('hello from list')
```

Note: it's perfectly valid to supply both `alias` and `aliases`. The reason there are two separate parameters is simply to make calling code read more naturally.

Next, we'll *add arguments to the subcommands* (page 20) and finally have a UI that looks like the one we sketched out in step 0!

1.3.10 Tutorial 09: Arguments, Again

Adding arguments to the subcommands should look familiar:

```
@todo
def add():
    """Add an item to the list."""
    parser.add_argument(
        'item',
        default=None,
```

(continues on next page)

(continued from previous page)

```
        help='item to add (prompts if not supplied)',
        nargs='?',
    )
    yield
    print('item:', args.item)

# ... snip ...

@todo
def done():
    """
    Remove an item from the list.

    If no arguments are supplied, the current list is printed and
    the program prompts for the index of the item to remove.
    """
    group = parser.add_mutually_exclusive_group()
    group.add_argument(
        '-a',
        '--all',
        action='store_true',
        default=False,
        help='remove all items from the list',
    )
    group.add_argument(
        '-i',
        '--index',
        default=None,
        help='0-based index of the item to remove',
        type=int,
    )
    yield
    print('index:', args.index)
    print('all:', args.all)
```

We use the same `parser` and `args` variables to configure and access arguments in subcommands! This is part of the "magic" provided by clik. When used in the `todo` function, `parser` refers to the top-level parser for the app. When used in a subcommand, `parser` refers to the subparser for that subcommand.

Note: This type of magic will turn off some Pythonistas; it's totally cool if you feel a little dirty and maybe a little angry right now. I mean, I wrote the thing and I'm still not totally sure how to feel about this part of it. All I can say is: I've been using this pattern for a few years now as clik has taken this final shape and, except for the occasional ringing of "explicit is better than implicit" in my head, it's been quite pleasant.

The `todo` app now has the UI we sketched out at the very beginning:

```
$ ./todo add -h
usage: todo add [-h] [item]

Add an item to the list.

positional arguments:
  item        item to add (prompts if not supplied)

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help  show this help message and exit

$ ./todo add
item: None

$ ./todo add "Wash the car"
item: Wash the car

$ ./todo done -h
usage: todo done [-h] [-a | -i INDEX]

Remove an item from the list.

optional arguments:
  -h, --help            show this help message and exit
  -a, --all             remove all items from the list
  -i INDEX, --index INDEX
                        0-based index of the item to remove

If no arguments are supplied, the current list is printed and the program
prompts for the index of the item to remove.

$ ./todo done
index: None
all: False

$ ./todo done -i 2
index: 2
all: False

$ ./todo done -a
index: None
all: True

$ ./todo done -a -i 2
usage: todo done [-h] [-a | -i INDEX]
todo done: error: argument -i/--index: not allowed with argument -a/--all
```

To show that `parser` is, in fact, different for the different subcommands, let's try to use an argument in `done` that is defined in `add`:

```
@todo
def done():
    # ... snip ...
    yield
    print('item:', args.item)  # defined in add
```

In the shell:

```
$ ./todo done
Traceback (most recent call last):
  File "./todo", line 83, in <module>
    todo.main()

# ... snip traceback ...

AttributeError: 'Namespace' object has no attribute 'item'
```

So that's most of what makes clik clik: `parser`, `args`, and subcommands.

We're now in the home stretch! Just a couple more steps and the application will be ready to ship.

(Also, I'd like to take this chance to thank you for continuing to read. I didn't know whether you'd be angry about that magic globals thing, and honestly I was a little afraid to bring it up with you. But now that we have that behind us and we're all cool, let's finish up this app shall we?)

Let's circle back and *make the list command print the items* (page 23) loaded from the data file.

1.3.11 Tutorial 10: Global

Right now, the todo items are "locked up" inside the `todo` function:

```
@app
def todo():
    # ... snip ...
    yield

    item_list = []
    if os.path.exists(args.file):
        with open(args.file) as f:
            item_list = json.load(f)
```

`add`, `list`, and `done` all need to access/modify `item_list`. How?

This is a common need for applications. The top-level app object reads data/configuration/etc, or opens a database connection, or sets up a client for a remote service – or whatever – and the subcommands use those "handles" to do their work.

clik provides a "global" object, `g`, to facilitate passing around global data/connection handles/etc:

```
from clik import app, args, g, parser  # note the g

@app
def todo():
    # ... snip ...
    yield

    g.item_list = []
    if os.path.exists(args.file):
        with open(args.file) as f:
            g.item_list = json.load(f)

# ... snip ...

@todo(name='list', alias='ls')
def list_():
    """Show the items on the list."""
    yield
    for i, item in enumerate(g.item_list):
        print('%i. %s' % (i, item))
```

Assuming the `test.json` file from before (with the following contents)...

```
[
  "Pick up nails from hardware store",
  "Grab milk from the grocery",
```

(continues on next page)

(continued from previous page)

```
  "Clean up the kitchen",
  "Feed the cats"
]
```

... `list` now prints our 0-indexed list of items:

```
$ ./todo -f test.json list
0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Clean up the kitchen
3. Feed the cats
```

Under the covers, `g` is just a dictionary that allows you to access values using attributes instead of brackets. The following sets of operations are identical:

```
g.foo = 'bar'
g['foo'] = 'bar'

g.foo
g['foo']

del g.foo
del g['foo']
```

We already know we'll need to print the 0-indexed list output inside `done`, so let's factor it out into a function:

```
def print_list():
    for i, item in enumerate(g.item_list):
        print('%i. %s' % (i, item))

# ... snip ...

@todo(name='list', alias='ls')
def list_():
    """Show the items on the list."""
    yield
    print_list()
```

Since we're thinking about it, let's go ahead and implement `done`:

```
import sys

# ... snip ...

@todo
def done():
    # ... snip ...
    yield

    if args.all:
        del g.item_list[:]
    else:
        index = args.index
        while index is None:
            print()
            print_list()
```

(continues on next page)

(continued from previous page)

```
            print()
            selection = input('Item number to remove? ')
            try:
                index = int(selection)
            except ValueError:
                print('error: not an integer:', selection, file=sys.stderr)
        if -1 < index < len(g.item_list):
            del g.item_list[index]
        else:
            print('error: index out of range:', index, file=sys.stderr)

    print()
    print('Updated list:')
    print_list()
```

This is all straightforward Python code; going over the details of the implementation is beyond the scope of this tutorial.

`add` is simpler and shorter than `done`:

```
@todo
def add():
    # ... snip ...
    yield
    item = args.item
    if item is None:
        item = input('Item to add: ') or None
    if item:
        g.item_list.append(item)
        print()
        print('Updated list:')
        print_list()
    else:
        print('error: empty item', file=sys.stderr)
```

Playing with the new commands and the `test.json` file, we see that things are generally working. Changes are not persisted to disk, but we'll tackle that problem in the next step.

```
$ ./todo -f test.json ls
0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Clean up the kitchen
3. Feed the cats

$ ./todo -f test.json add "Hang picture on the wall"

Updated list:
0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Clean up the kitchen
3. Feed the cats
4. Hang picture on the wall

$ ./todo -f test.json add
Item to add: Hang picture on the wall

Updated list:
```

(continues on next page)

(continued from previous page)

```
0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Clean up the kitchen
3. Feed the cats
4. Hang picture on the wall

$ ./todo -f test.json add ""
error: empty item

$ ./todo -f test.json add ""
Item to add:
error: empty item

$ ./todo -f test.json ls
0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Clean up the kitchen
3. Feed the cats

$ ./todo -f test.json done -i 2

Updated list:
0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Feed the cats

$ ./todo -f test.json done -a

Updated list:

$ ./todo -f test.json done

0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Clean up the kitchen
3. Feed the cats

Item number to remove? 3

Updated list:
0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Clean up the kitchen

$ ./todo -f test.json done -i 10
error: index out of range: 10

Updated list:
0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Clean up the kitchen
3. Feed the cats

$ ./todo -f test.json done

0. Pick up nails from hardware store
1. Grab milk from the grocery
```

(continues on next page)

(continued from previous page)

```
2. Clean up the kitchen
3. Feed the cats

Item number to remove? foo
error: not an integer: foo

0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Clean up the kitchen
3. Feed the cats

Item number to remove? 12
error: index out of range: 12

Updated list:
0. Pick up nails from hardware store
1. Grab milk from the grocery
2. Clean up the kitchen
3. Feed the cats
```

Nice! The application has really started to take shape. Next we'll save the changes to disk using *cleanup code in the app function* (page 27).

Note: `g` (along with the magic `parser` and `args` variables) is the other design decision experienced Pythonistas might (rightfully) sneer at. Global variables are generally discouraged in Python, and `g` actively encourages their use (even if veiled behind a not-technically-a-global-depending-on-how-you-look-at-it proxy object).

The justification is the same as for `parser` / `args`. This "`g` pattern" is one I've used extensively (in clik and in Flask) and, while it may be against the Zen of Python, it's damn useful. *Used judiciously*, it can be a real boon to productivity and overall code clarity.

1.3.12 Tutorial 11: Cleanup

There's a final phase to execution that we haven't discussed yet: cleanup. For command functions with subcommands (like our `todo` function), cleanup is an optional third block of code that gets run after child commands have run:

```
@app
def todo():
    # configure argument parser
    yield  # give control back to clik, which parses end user arguments
    # do something with parsed arguments
    yield  # give control back to clik, which runs child commands
    # clean up
```

This is where we'll persist the changes that subcommands make to `g.item_list`:

```
@app
def todo():
    # ... snip ...
    yield

    # Same as before
    g.item_list = []
```

(continues on next page)

(continued from previous page)

```
    if os.path.exists(args.file):
        with open(args.file) as f:
            g.item_list = json.load(f)

    # New stuff
    yield

    with open(args.file, 'w') as f:
        json.dump(g.item_list, f, indent=2)
        f.write('\n')
```

And that's it! The app now does what we sketched out in the first step:

```
$ ls
bin             lib                         test.json
include         pip-selfcheck.json          todo

$ ./todo ls

$ ls
bin             pip-selfcheck.json          todo.json
include         test.json
lib             todo

$ cat todo.json
[]

$ ./todo add "Pick up nails from the hardware store"

Updated list:
0. Pick up nails from the hardware store

$ cat todo.json
[
  "Pick up nails from the hardware store"
]

$ ./todo ls
0. Pick up nails from the hardware store

$ ./todo add
Item to add: Grab milk from the grocery

Updated list:
0. Pick up nails from the hardware store
1. Grab milk from the grocery

$ cat todo.json
[
  "Pick up nails from the hardware store",
  "Grab milk from the grocery"
]

$ ./todo ls
0. Pick up nails from the hardware store
1. Grab milk from the grocery
```

(continues on next page)

(continued from previous page)

```
$ ./todo add "Clean up the kitchen"

Updated list:
0. Pick up nails from the hardware store
1. Grab milk from the grocery
2. Clean up the kitchen

$ ./todo add "Feed the cats"

Updated list:
0. Pick up nails from the hardware store
1. Grab milk from the grocery
2. Clean up the kitchen
3. Feed the cats

$ cat todo.json
[
  "Pick up nails from the hardware store",
  "Grab milk from the grocery",
  "Clean up the kitchen",
  "Feed the cats"
]

$ ./todo done -i 2

Updated list:
0. Pick up nails from the hardware store
1. Grab milk from the grocery
2. Feed the cats

$ cat todo.json
[
  "Pick up nails from the hardware store",
  "Grab milk from the grocery",
  "Feed the cats"
]

$ ./todo done

0. Pick up nails from the hardware store
1. Grab milk from the grocery
2. Feed the cats

Item number to remove? 1

Updated list:
0. Pick up nails from the hardware store
1. Feed the cats

$ cat todo.json
[
  "Pick up nails from the hardware store",
  "Feed the cats"
]

$ ./todo done -a
```

(continues on next page)

(continued from previous page)

```
Updated list:

$ cat todo.json
[]
```

It works. Cool.

In the next couple steps we'll put on some finishing touches by *implementing error codes* (page 30) for our couple user-input-error situations, then adding one final tweak to let users simply run `./todo` to get a list of items (instead of `./todo list`).

1.3.13 Tutorial 12: Exit Code

There are a lot of places execution can go wrong. clik lets you bail out at any point by yielding a non-`None` value. The value yielded is used as the exit code for the invocation.

...well, kind of. When a subcommand yields non-`None`, clik immediately starts "unwinding" through the cleanup blocks of the parent commands. Parent commands can override the exit code from children.

...and you really can unwind from anywhere. That first yield separating the "parser config" block from the "do stuff" block? If you yield non-`None` from there the application will exit before the parser is even fully configured. Meaning that it's a *hard* fail, and end users won't even be able to use `-h/--help`. In general this isn't what you want, but consider a situation where a default value to a critical argument (say, "database host") is expected to be in an environment variable. You control all the machines the app runs on, and it really is a situation where, if that variable is not set, all kinds of stuff is wrong. In that case it might be perfectly appropriate to hard fail without even initializing the parser. The point is: clik gives you the choice and makes it easy to do that if you want to.

Back to our application:

```
@todo
def add():
    # ... snip ...
    if item:
        g.item_list.append(item)
        # ... snip ...
    else:
        print('error: empty item', file=sys.stderr)
        yield 1  # <--- !!! new code !!! ---

# ... snip ...

@todo
def done():
    # ... snip ...
    if args.all:
        del g.item_list[:]
    else:
        # ... snip ...
        if -1 < index < len(g.item_list):
            del g.item_list[index]
        else:
            print('error: index out of range:', index, file=sys.stderr)
            yield 1  # <--- !!! new code !!! ---
    # ... snip ...
```

The application now has an exit code of `1` when the user provides invalid input and, as a nice side effect, the "Updated list" is no longer printed when the list is not actually updated:

```
$ ./todo add ""
error: empty item

$ echo $?
1

$ ./todo add
Item to add:
error: empty item

$ echo $?
1

$ ./todo add "Pick up nails from the hardware store"

Updated list:
0. Pick up nails from the hardware store

$ echo $?
0

$ ./todo done -i asdf
usage: todo done [-h] [-a | -i INDEX]
todo done: error: argument -i/--index: invalid int value: 'asdf'

$ echo $?
1

$ ./todo done -i -1
error: index out of range: -1

$ echo $?
1

$ ./todo done

0. Pick up nails from the hardware store

Item number to remove? 5
error: index out of range: 5

$ echo $?
1

$ ./todo done -i 0

Updated list:

$ echo $?
0
```

One final tweak (page 31) and the tutorial is complete!

1.3.14 Tutorial 13: Bare

It turns out the grumpy systems administrators from *Tutorial 08: Aliases* (page 19) were right. Printing the todo list is such a common operation that the extra characters (even with the shortened `ls` alias) are having company-wide

impacts on todo-related productivity.

It would make a lot of sense if running `./todo` without any arguments just printed the todo list.

Using clik, it can! "Bare commands" (named as such because I spent two weeks thinking about it and couldn't come up with anything better) allow a command with subcommands – which would normally require one of the subcommands to be supplied – to be invoked "bare" (i.e. without a subcommand).

There are some (serious) limitations:

- Positional arguments are not allowed for bare commands (if the user runs `./app foo` is `foo` a subcommand or a positional argument?)
- Mutually exclusive groups are not allowed (this is an internal limitation)
- Unknown arguments are not allowed (similar rationale to positional arguments) *(note: unknown arguments are not covered in the tutorial)*

So it's far from perfect, but it's better than nothing.

Implementing the "bare" command for our `todo` app:

```
@todo.bare
def bare():
    yield
    print_list()
```

Poking around in the shell (note the updated usage statement):

```
$ ./todo -h
usage: todo [-h] [-f FILE] {add,list,done} ...
       todo [-h] [-f FILE]

Command-line application for managing a todo list.

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  file in which to store data (default: todo.json)

subcommands:
  {add,list,done}
    add                 Add an item to the list.
    list (ls)           Show the items on the list.
    done                Remove an item from the list.

The list is stored on disk as a simple JSON file containing an array of
strings. The file path is controlled by the -f/--file argument (see
documentation for that argument for more information).

$ ./todo add "Pick up nails from the hardware store"
# ... snip ...

$ ./todo
0. Pick up nails from the hardware store

$ ./todo add "Grab milk from the grocery"
# ... snip ...

$ ./todo
0. Pick up nails from the hardware store
```

(continues on next page)

(continued from previous page)

```
1. Grab milk from the grocery

$ ./todo ls
0. Pick up nails from the hardware store
1. Grab milk from the grocery
```

And so, the demo application is finally complete. Ship it!

CHAPTER 2

Development

2.1 Quickstart

2.1.1 Prerequisites

Requirements:

- Common development tools like git, make, a C compiler, etc
- Python 3.6 – this is the "main" interpreter used for development
- Virtualenv

Recommendations:

- All supported Python interpreters
  - Python 2.6
  - Python 2.7
  - Python 3.3
  - Python 3.4
  - Python 3.5
  - Python 3.6
  - PyPy
  - PyPy 3

Suggestions:

- LaTeX – for building the documentation as a PDF

2.1.2 Setup

The repository can be cloned anywhere you like on your local machine. At any time, you can delete the entire project and its environment by `rm -rf` -ing the local directory.

The following instructions clone the repository to `~/clik`:

```
cd
git clone https://github.com/decafjoe/clik.git
cd clik
make env
```

Wait ~10m and you should be good to go!

Note that all dependencies are installed underneath the repository directory (take a peek at `.env/`). To delete the development environment artifacts, you can run `make pristine` (see below). To delete everything, simply `rm -rf` the clone.

2.1.3 Tooling

Clik's developer tooling is exposed via `make`. Run `make` with no arguments to get a list of available targets. All targets except `make release` are idempotent, so they can be run at any time.

Environment:

- `make env` installs the development environment; subsequent runs update the environment if required
- `make clean` deletes build artifacts like .pyc files, sdists, etc
- `make pristine` kills the local development environment
- `make check-update` checks for updates to Python packages installed in the development environment

Documentation:

- `make html` generates the HTML documentation to `doc/_build/html/`
- `make pdf` generates the PDF documentation to `doc/_build/latex/clik.pdf`
- `make docs` builds both HTML and PDF documentation to their respective locations

Build:

- `make dist` builds a sdist into `dist/`
- `make release` builds a clean sdist, uploads it to PyPI, tags the commit with the current version number, bumps the version, then commits the new version number and pushes it up to GitHub (this is largely implemented by the `tool/pre-release` and `tool/post-release` scripts)

QA:

- `make lint` runs the Flake8 linter on the Python files in the project
- `make test` runs the functional and unit test suites against the "main" development interpreter
- `make test-all` runs the linter, runs the functional and unit test suites against all supported interpreters, and generates a coverage report to `coverage/`

Be aware of `tool/test`. It allows for precise selection of what tests to run. It's a time-saver when working on a small part of the codebase. Instead of running the entire test suite after every change, you can simply run the relevant tests. See `tool/test -h` for more information.

2.2 Workflow

Note: Clik is currently maintained by a single person. For now, I don't want to put any hard and fast rules on how development should be done. What follows is a sketch.

Master must always be stable, working, QA-ed code. That is, at all times master must:

- Be free of linter violations
- Pass the full test suite on all supported interpreters
- Maintain 100% code coverage (exceptions may be made)
- Contain approprite documentation for any changes (in end user documentation, docstrings, developer documentation, etc)
- Have a descriptive commit messages

Development must happen off-master. In other words, you should almost *never be committing or pushing directly to master.*

Once the patch is working, **history must be rewritten to be linear and neat, and must be rebased off of the current master.** Group changes logically. Larger groups of smaller commits are preferable to smaller groups of larger commits.

With the patch and history ready, **submit a pull request**. The pull request must provide a general description of the change and the rationale for why clik needs the code. Commit messages are the "what", pull request messages are the "why".

Submitting a pull request will automatically start a Travis CI run to check for linter violations or test failures. If the test run passes, a clik committer will review your changes for a possible merge into master.

2.3 Internals

2.3.1 clik

The command line interface kit.

Clik is a tool for writing complex command-line interfaces with minimal boilerplate and bookkeeping.

This top-level package pulls together the end user API from the various modules within clik.

author Joe Joyce <joe@decafjoe.com>

copyright Copyright (c) Joe Joyce and contributors, 2009-2019.

license BSD

`clik.__version__ = '0.92.5'`
Current version.

Type `str`[5]

[5] https://docs.python.org/3/library/stdtypes.html#str

2.3.2 clik.app

Top-level *App* (page 39) class and helpers.

author Joe Joyce <joe@decafjoe.com>

license BSD

clik.app.**current_app**
Magic variable containing the active application instance.

Type *clik.magic.Magic* (page 46) – *clik.app.App* (page 39)

clik.app.**parser**
Magic variable containing the current parser.

Type *clik.magic.Magic* (page 46) – *clik.argparse.ArgumentParser* (page 40)

clik.app.**args**
Magic variable containing parsed arguments.

Type *clik.magic.Magic* (page 46) – argparse.Namespace[6]

clik.app.**g**
Magic variable containing globals.

Type *clik.magic.Magic* (page 46) – clik.app.AttributeDict

clik.app.**run_children**
Magic variable containing the function to run child commands.

Type *clik.magic.Magic* (page 46)

clik.app.**unknown_args**
Magic variable containing unknown arguments.

Type *clik.magic.Magic* (page 46) – list[7]

clik.app.**app**(*fn=None*, *name=None*)
Decorate the main application generator function.

If the decorator is given no arguments, the name of the application is the name of the decorated generator function:

```
# Application will be named 'myapp' in this case
@app
def myapp():
    yield
```

The application name can be set by passing a string to name:

```
# Application will be named 'theapp' in this case
@app(name='theapp')
def myapp():
    yield
```

Parameters

[6] https://docs.python.org/3/library/argparse.html#argparse.Namespace
[7] https://docs.python.org/3/library/stdtypes.html#list

- **fn** – Main application generator function. Name of the application will be set to `fn.__name__`.
- **name** – Overrides name of application. Must not be used with the `fn` argument (if used with `fn`, `name` is ignored).

Returns *App* (page 39) if `fn` is `None`, otherwise a decorator returning *App* (page 39).

class clik.app.**App**(*fn*, *name=None*)

Bases: *clik.command.Command* (page 41)

`clik.Command` subclass that implements the *main()* (page 39) method.

main() (page 39) is the user-level API for starting the application.

main(*argv=None*, *exit=<built-in function exit>*)

Start the application.

Parameters

- **argv** – Optional list of command-line arguments. If not supplied, this defaults to `sys.argv`.
- **exit** (`fn(integer_exit_code)`) – Optional function to call on exit. If not supplied, this defaults to `sys.exit()`[8].

Returns Return value of `exit`

2.3.3 clik.argparse

Most of the hackery that makes clik tick.

author Joe Joyce <joe@decafjoe.com>

license BSD

clik.argparse.**ALLOW_UNKNOWN = '_clik_unknown'**

Name of the argument that contains whether to allow unknown arguments.

Type `str`[9]

exception clik.argparse.**ArgumentParserExit**(*code*)

Raised instead of allowing `argparse`[10] to call `sys.exit()`[11].

code = None

Exit code.

Type `int`[12]

exception clik.argparse.**BareUnsupportedFeatureError**(*feature*)

Raised when using a feature that is not supported by bare commands.

feature = None

Feature that is unsupported.

Type `str`[13]

[8] https://docs.python.org/3/library/sys.html#sys.exit
[9] https://docs.python.org/3/library/stdtypes.html#str
[10] https://docs.python.org/3/library/argparse.html#module-argparse
[11] https://docs.python.org/3/library/sys.html#sys.exit
[12] https://docs.python.org/3/library/functions.html#int
[13] https://docs.python.org/3/library/stdtypes.html#str

class clik.argparse.**HelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: argparse.HelpFormatter

Format usage with no trailing newline on usage.

class clik.argparse.**ArgumentParser**(**args*, ***kwargs*)

Bases: argparse.ArgumentParser[14]

argparse.ArgumentParser[15] specialized for clik.

_clik_bare_arguments()

Context manager for bare command mode.

When the parser is in bare command mode, it disallows certain features (like positional args and mutually exclusive groups). In addition, the argument destinations are recorded in order to do some post-processing before running the selected command.

add_argument(**args*, ***kwargs*)

Override default behavior to disallow posargs in bare commands.

Raise *BareUnsupportedFeatureError* (page 39) if adding a positional argument to a bare command.

add_mutually_exclusive_group(**args*, ***kwargs*)

Override default behavior to disallow mutex groups in bare commands.

Raise *BareUnsupportedFeatureError* (page 39) if adding a mutually exclusive group to a bare command.

allow_unknown_args()

Allow unknown arguments, putting them in clik.unknown_args.

Raise UnknownArgsUnsupportedError if this parser has subparsers – unknown arguments are allowed *only* on leaf commands.

exit(*status=0*, *message=None*)

Override default behavior to avoid interpreter exit.

By default, the parser calls sys.exit()[16]. In certain situations – namely testing – we don't actually want to exit the Python interpreter.

So instead of exiting, this throws an *ArgumentParserExit* (page 39) exception which can be caught by the caller.

Parameters

- **status** (*int*[17]) – Exit status.
- **message** (*str*[18]) – Optional message. If supplied, will be printed to sys.stderr[19] before raising the exception.

Raise *ArgumentParserExit* (page 39)

format_help()

Override default behavior to support formatting bare commands.

format_usage()

Override default behavior to use clik's formatter.

[14] https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser
[15] https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser
[16] https://docs.python.org/3/library/sys.html#sys.exit
[17] https://docs.python.org/3/library/functions.html#int
[18] https://docs.python.org/3/library/stdtypes.html#str
[19] https://docs.python.org/3/library/sys.html#sys.stderr

2.3.4 clik.command

All the recursive, argument-parsin', context-managin' goodness.

author Joe Joyce <joe@decafjoe.com>

copyright Copyright (c) Joe Joyce and contributors, 2009-2019.

license BSD

clik.command.**catch = <object object>**
Globally-unique value used by commands to indicate they want clik to send the exception (if one occurred) in addition to the child exit code in response to a `yield`.

Type object[20]

clik.command.**ARGS = 'args'**
Name of the magic variable containing the parsed arguments.

Type str[21]

clik.command.**STACK = '_clik_stack'**
Name of the argument that contains the stack of commands to be run.

Type str[22]

clik.command.**SHOW_SUBCOMMANDS = 3**
If a parent has more than this number of subcommands, the help output will `{command}` instead of the full list of subcommands.

Type int[23]

exception clik.command.**BareAlreadyRegisteredError**(*command*)
Raised when a bare command has already been registered.

command = None
Command caller was trying to register as the bare command.

Type *Command* (page 41)

class clik.command.**Command**(*ctx*, *fn*, *name=None*, *alias=None*, *aliases=None*)
Bases: object[24]

The invisible backend driving most of what the user interacts with.

__init__(*ctx*, *fn*, *name=None*, *alias=None*, *aliases=None*)
Initialize the command object.

Parameters

- **ctx** (*clik.context.Context* (page 44)) – Context object.
- **fn** – Generator function – the actual command.
- **name** (*str*[25]) – Name of the command or `None`. If `None`, name will be `fn.__name__`.
- **alias** (*str*[26]) – Command alias or `None`. If `None`, the command has no aliases. If this and `aliases` are both supplied, `alias` will be prepended to the `aliases` list.

[20] https://docs.python.org/3/library/functions.html#object
[21] https://docs.python.org/3/library/stdtypes.html#str
[22] https://docs.python.org/3/library/stdtypes.html#str
[23] https://docs.python.org/3/library/functions.html#int
[24] https://docs.python.org/3/library/functions.html#object
[25] https://docs.python.org/3/library/stdtypes.html#str
[26] https://docs.python.org/3/library/stdtypes.html#str

- **aliases** (list[27] or None) – List of additional aliases for the command or None.

__call__(*fn=None*, *name=None*, *alias=None*, *aliases=None*)

Add subcommands to this command.

Basic use:

```
@myapp
def mysubcommand():
    yield
```

Customizing the subcommand:

```
@myapp(name='subcommand', alias='sub', aliases=['s'])
def mysubcommand():
    yield
```

Parameters

- **fn** – Generator function or None. If fn is supplied, all other arguments are ignored.
- **name** (*str*[28]) – Name of the command or None.
- **alias** (*str*[29]) – Command alias. See *__init__()* (page 41) for information on how aliases are handled.
- **aliases** (list[30] or None) – List of additional aliases for the command or None.

_configure_parser(*parser*, *parent=None*, *stack=None*)

_check_bare_arguments()

_run(*child=False*)

_aliases = None

Tuple of aliases for this command.

Type tuple[31] of str[32]

_bare = None

Command (page 41) instance for the bare command or None if bare command is not set.

Type *Command* (page 41) or None

_children = None

List of child commands.

Type list[33] of *Command* (page 41) instances

_ctx = None

Context object for this command. This context is shared between all command instances associated with a *clik.app.App* (page 39).

Type *clik.context.Context* (page 44)

[27] https://docs.python.org/3/library/stdtypes.html#list
[28] https://docs.python.org/3/library/stdtypes.html#str
[29] https://docs.python.org/3/library/stdtypes.html#str
[30] https://docs.python.org/3/library/stdtypes.html#list
[31] https://docs.python.org/3/library/stdtypes.html#tuple
[32] https://docs.python.org/3/library/stdtypes.html#str
[33] https://docs.python.org/3/library/stdtypes.html#list

_fn = None
Generator function – the actual command.

Type generator function

_generator = None
In-progress generator for *_fn* (page 42). This is the object that we call `generator.send()`[34] and `generator.next()` on.

Type `generator`

_name = None
Canonical name of the command.

Type `str`[35]

_parent = None
Parent command. For the root *clik.app.App* (page 39) instance, this is `None`. Set in *_configure_parser()* (page 42).

Type *Command* (page 41) or `None`

_parser = None
Parser for this command. Set in *_configure_parser()* (page 42).

Type *clik.argparse.ArgumentParser* (page 40)

2.3.5 clik.compat

Python compatibility helpers.

author Joe Joyce <joe@decafjoe.com>

copyright Copyright (c) Joe Joyce and contributors, 2009-2019.

license BSD

`clik.compat.`**PY2**
Indicates whether we are on Python 2.

Type `bool`[36]

`clik.compat.`**PY26**
Indicates whether we are on Python 2.6.

Type `bool`[37]

`clik.compat.`**PY33**
Indicates whether we are on Python 3.3.

Type `bool`[38]

2.3.6 clik.context

Manage bindings for *clik.magic.Magic* (page 46) variables.

author Joe Joyce <joe@decafjoe.com>

[34] https://docs.python.org/3/reference/expressions.html#generator.send
[35] https://docs.python.org/3/library/stdtypes.html#str
[36] https://docs.python.org/3/library/functions.html#bool
[37] https://docs.python.org/3/library/functions.html#bool
[38] https://docs.python.org/3/library/functions.html#bool

copyright Copyright (c) Joe Joyce and contributors, 2009-2019.

license BSD

exception clik.context.**LockedMagicError**(*name*)
Raised when trying to acquire a magvar that is already acquired.

name = None
Name of the magic variable that is locked.

Type str[39]

exception clik.context.**MagicNameConflictError**(*name*)
Raised when trying to register an already-registered magvar.

name = None
Name of the magic variable that is already registered.

Type str[40]

exception clik.context.**UnregisteredMagicNameError**(*name*)
Raised when trying to access an unregistered magic variable.

name = None
Name of the magic variable that is unregistered.

Type str[41]

exception clik.context.**UnboundMagicError**(*name*)
Raised when trying to access a magic variable that is not bound.

name = None
Name of the magic variable that is unbound.

Type str[42]

class clik.context.**Context**
Bases: object[43]

Bindings manager for magic variables.

__call__(***kwargs*)
Context manager for *push()* (page 45) -ing kwargs during a code block.

Example:

```
context = Context()
context.register('foo')
with context(foo='bar'):
    pass  # do some stuff
```

Before the block, each key/value pair in kwargs is passed to *push()* (page 45). After the block, each key is *pop()* (page 45) -ped.

acquire(**magic_variables*)
Context manager to lock magic_variables from use by other contexts.

Only one context at a time is allowed to control the binding of a magic variable. Using this context manager ensures the caller can safely manipulate the binding without interference from other contexts:

[39] https://docs.python.org/3/library/stdtypes.html#str
[40] https://docs.python.org/3/library/stdtypes.html#str
[41] https://docs.python.org/3/library/stdtypes.html#str
[42] https://docs.python.org/3/library/stdtypes.html#str
[43] https://docs.python.org/3/library/functions.html#object

```
foo = Magic('foo')
context1 = Context()
context1.register('foo')
context2 = Context()
context2.register('foo')
with context1.acquire(foo):
    with context1(foo='bar'):
        pass  # do some stuff
    with context2.acquire(foo):
        # BOOM! LockedMagicError gets thrown
```

Before the block, each of the `magic_variables` is checked to see if it currently has a context. If so, *LockedMagicError* (page 44) is thrown. Otherwise, the context is set to this instance.

After the block, the context for each magic variable is reset to `None`, freeing it up for use by other contexts.

Raise *LockedMagicError* (page 44) if one of `magic_variables` is already acquired

get(*name*)

Return currently-bound value of magic variable named `name`.

Parameters **name** (*str*[44]) – Name of magic variable.

Raise *UnregisteredMagicNameError* (page 44) if `name` is not registered

Raise *UnboundMagicError* (page 44) if variable is not currently bound

pop(*name*)

Pop and return the current value off the variable's stack.

This rebinds the variable to the next-highest item on the stack.

Parameters **name** (*str*[45]) – Name of magic variable.

Raise *UnregisteredMagicNameError* (page 44) if `name` is not registered

Raise *UnboundMagicError* (page 44) if variable is not currently bound

push(*name*, *obj*)

Push a value on to a variable's stack, rebinding its current value.

Parameters

- **name** (*str*[46]) – Name of magic variable.
- **obj** – New value to push on to the stack.

Raise *UnregisteredMagicNameError* (page 44) if `name` is not registered

register(*name*)

Register a magic variable name.

Requiring registration prevents accidental conflicts between modules. If two modules (which may not know about each other) both try to register the same magic variable, clik will thrown an exception.

Parameters **name** (*str*[47]) – Name of magic variable.

Raise *MagicNameConflictError* (page 44) if `name` is already registered

unregister(*name*)

Unregister a magic variable name.

[44] https://docs.python.org/3/library/stdtypes.html#str
[45] https://docs.python.org/3/library/stdtypes.html#str
[46] https://docs.python.org/3/library/stdtypes.html#str
[47] https://docs.python.org/3/library/stdtypes.html#str

Parameters **name** (*str*[48]) – Name of magic variable.

Raise *UnregisteredMagicNameError* (page 44) if name is not registered

2.3.7 clik.magic

Slightly adapted version of Werkzeug's werkzeug.local.LocalProxy.

Original code licensed from the Werkzeug Team under the following terms:

> Copyright (c) 2014 by the Werkzeug Team, see AUTHORS for more details.
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
> - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
> - The names of the contributors may not be used to endorse or promote products derived from this software without specific prior written permission.
>
> THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
>
> **copyright** Copyright (c) 2014 by the Werkzeug Team
>
> **authors** See Werkzeug's AUTHORS file
>
> **license** BSD

class clik.magic.**Magic**(*name*)

2.3.8 clik.util

The ever-present utilities module.

> **author** Joe Joyce <joe@decafjoe.com>
>
> **copyright** Copyright (c) Joe Joyce and contributors, 2009-2019.
>
> **license** BSD

class clik.util.**AttributeDict**

Bases: dict[49]

Simple dict[50] wrapper that allows key access via attribute.

[48] https://docs.python.org/3/library/stdtypes.html#str
[49] https://docs.python.org/3/library/stdtypes.html#dict
[50] https://docs.python.org/3/library/stdtypes.html#dict

Example:

```
d = AttributeDict(foo='bar', baz='qux')
d['foo']       # 'bar'
d.foo          # 'bar'
d['baz']       # 'qux'
d.baz          # 'qux'
d.foo = 'bup'
d['foo']       # 'bup'
d.foo          # 'bup'
del d.foo
d.foo          # KeyError
```

__delattr__(*name*)
Delete via attribute name.

__getattr__(*name*)
Get via attribute name.

__setattr__(*name*, *value*)
Set via attribute name.

2.4 Changelog

2.4.1 0.92.5 – unreleased

2.4.2 0.92.4 – 2019-05-23

- The `__doc__` attribute of `Command` instances were changed to be the decorated function's `__doc__` rather than the generic `Command` docstring.

2.4.3 0.92.3 – 2018-11-01

- Fixed logic bug that, when an end user is running Python 2.7 and does not supply a required positional argument, causes an exception within clik instead of printing an error message.

2.4.4 0.92.2 – 2017-12-19

- Added introductory documentation: example code, tutorial, and screencast.

2.4.5 0.92.1 – 2017-11-28

- Fixed incorrect `__version__` attribute.

2.4.6 0.92.0 – 2017-11-23

- Moved internal `AttributeDict` class from `clik.app` to `clik.util`.

2.4.7 0.91.0 – 2017-11-06

- Added a facility for handling unknown arguments.

2.4.8 0.90.2 – 2017-10-12

- Calling `run_children()` when there are no children no longer raised an exception; it simply returned 0 (i.e. no error in the children).

2.4.9 0.90.1 – 2017-09-06

- Updated PyPI trove classifier `Development Status` from `1 - Planning` to `3 - Alpha`.

2.4.10 0.90.0 – 2017-09-05

- Initial public re-release.

2.4.11 Pre-0.90.0

The Dark Old Days.

Python Module Index

C

Index

Symbols

A

B

C

E

F

G