Face

Release 22.0

Dec 23, 2022

Contents:

face is a Pythonic microframework for building command-line applications:

- First-class subcommand support
- Powerful middleware architecture
- Separate Parser layer
- Built-in flagfile support
- Handy testing utilities
- Themeable help display

CHAPTER 1

Installation

face is pure Python, and tested on Python 3.7+, as well as PyPy. Installation is easy:

```
pip install face
```

Then get to building your first application!

```python
from face import Command, echo

def hello_world():
    "A simple greeting command."
    echo('Hello, world!')

cmd = Command(hello_world)

cmd.run()

"""
# Here's what the default help looks like at the command-line:

$ cmd --help
Usage: cmd [FLAGS]

  A simple greeting command.


Flags:

  --help / -h   show this help message and exit
"""
```

CHAPTER 2

Getting Started

Check out our *Tutorial* for more.

2.1 Tutorial

Contents

– *Montage Admin Tools*

2.1.1 Part I: Say

The field of overdone versions of `echo` has been too long dominated by Big GNU. Today, we start taking back the power. We will implement the `say` command.

Positional arguments

While face offers a Parser interface underneath, the canonical way to create even the simplest CLI is with the Command object.

To demonstrate, we'll start with the basics, positional arguments. `say hello world` should print `hello world`:

```
from face import Command, echo


def main():
    cmd = Command(say, posargs=True)  # posargs=True means we accept positional␣
↪arguments
    cmd.run()


def say(posargs_):  # positional arguments are passed through the posargs_ parameter
    echo(' '.join(posargs_))  # our business logic


if __name__ == '__main__':  # standard fare Python: https://stackoverflow.com/
↪questions/419163
    main()
```

A basic Command takes a single function entrypoint, in our case, the `say` function.

Note: Face's `echo()` function is a version of `print()` with improved options and handling of console states, ideal for CLIs.

Flags

Let's give `say` some options:

`say --upper hello world` or `say -U hello world` should print `HELLO WORLD`.

```
...

 def main():
     cmd = Command(say, posargs=True)
     cmd.add('--upper', char='-U', parse_as=True, doc='uppercase all output')
     cmd.run()


 def say(posargs_, upper):  # our --upper flag is bound to the upper parameter
     args = posargs_
```

(continues on next page)

(continued from previous page)

```
    if upper:
        args = [a.upper() for a in args]
    echo(' '.join(args))

...
```

The parse_as keyword argument being set to True means that the presence of the flag results in the True value itself. As we'll see below, flags can take arguments, too.

Flags with values

Let's add more flags, this time ones that take values.

`say --separator . hello world` will print `hello.world`. Likewise, `say --count 2 hello world` will repeat it twice: `hello world hello world`

```
...

def main():
    cmd = Command(say, posargs=True)
    cmd.add('--upper', char='-U', parse_as=True, doc='uppercase all output')
    cmd.add('--separator', missing=' ', doc='text to put between arguments')
    cmd.add('--count', parse_as=int, missing=1, doc='how many times to repeat')
    cmd.run()


def say(posargs_, upper, separator, count):
    args = posargs_ * count
    if upper:
        args = [a.upper() for a in args]
    echo(separator.join(args))

...
```

Now we can see that `parse_as`:

- Can take a value (e.g., `True`), which make the flag no-argument
- Can take a callable (e.g., `int`), which is used to convert the single argument
- Defaults to `str` (as used by `separator`)

We can also see the `missing` keyword argument, which specifies the value to be passed to the Command's handler function if the flag is absent. Without this, `None` is passed.

Note: Face also supports required flags, though they are not an ideal CLI UX best practice. Simply set `missing` to `face.ERROR`.

More Interesting Flag Types

`say --multi-separator=@,# hello wonderful world` prints `hello@wonderful#world` (The separators repeat)

`say --from-file=fname` reads the file and adds all words from it to its output `say --animal=dog|cat|cow` will prepend “woof”, “meow”, or “moo” respectively.

2.1.2 Part II: Calc

(Details TBD!)

With `echo` having met its match, we are on to bigger and better: this time, with math

```
$ num
<Big help text>
```

Add and Multiply

```
$ num add 1 2
3
```

```
$ num mul 3 5
15
```

Subtract

```
$ num sub 10 5
5
$ num sub 5 10
Error: can't substract
$ num --allow-negatives 5 10
-5
```

Divide

```
$ num div 2 3
0.6666666666666666
$ num div --int 2 3
0
```

Precision support

```
$ num add 0.1 0.2
0.30000000000000004
$ num add --precision=3 0.1 0.2
0.3
```

Oh, now let’s add it to all subcommands.

2.1.3 Part III: Middleware

(Details TBD!)

Doing math locally is all well and good, but sometimes we need to use the web.

We will add an "expression" sub-command to num that uses `https://api.mathjs.org/v4/`. But since we want to unit test it, we will create the `httpx.Client` in a middleware.

```
$ num expression "1 + (2 * 3)"
7
```

But we can also write a unit test that does not touch the web:

```
$ pytest test_num.py
```

2.1.4 Part IV: Examples

There are more realistic examples of *face* usage out there, that can serve as a reference.

Cut MP4

The script cut_mp4 is a quick but useful tool to cut recordings using `ffmpeg`. I use it to slice and dice the Python meetup recordings. It does not have subcommands or middleware, just a few flags.

Glom

Glom is a command-line interface front end for the `glom` library. It does not have any subcommands, but does have some middleware usage.

Pocket Protector

Pocket Protector is a secrets management tool. It is a medium-sized application with quite a few subcommands for manipulating a YAML file.

Montage Admin Tools

Montage Admin Tools is a larger application. It has nested subcommands and a database connection. It is used to administer a web application.

2.2 Command

class `face.`**`Command`**(*func*, *name=None*, *doc=None*, ***kwargs*)

The central type in the face framework. Instantiate a Command, populate it with flags and subcommands, and then call command.run() to execute your CLI.

Note that only the first three constructor arguments are positional, the rest are keyword-only.

Parameters

- **func** (*callable*) – The function called when this command is run with an argv that contains no subcommands.
- **name** (*str*) – The name of this command, used when this command is included as a subcommand. (Defaults to name of function)
- **doc** (*str*) – A description or message that appears in various help outputs.

- **flags** (*list*) – A list of Flag instances to initialize the Command with. Flags can always be added later with the .add() method.
- **posargs** (*bool*) – Pass True if the command takes positional arguments. Defaults to False. Can also pass a PosArgSpec instance.
- **post_posargs** (*bool*) – Pass True if the command takes additional positional arguments after a conventional '–' specifier.
- **help** (*bool*) – Pass False to disable the automatically added –help flag. Defaults to True. Also accepts a HelpHandler instance, see those docs for more details.
- **middlewares** (*list*) – A list of @face_middleware decorated callables which participate in dispatch. Also addable via the .add() method. See Middleware docs for more details.

add (**a*, ***kw*)

Add a flag, subcommand, or middleware to this Command.

If the first argument is a callable, this method contructs a Command from it and the remaining arguments, all of which are optional. See the Command docs for for full details on names and defaults.

If the first argument is a string, this method constructs a Flag from that flag string and the rest of the method arguments, all of which are optional. See the Flag docs for more options.

If the argument is already an instance of Flag or Command, an exception is only raised on conflicting subcommands and flags. See add_command for details.

Middleware is only added if it is already decorated with @face_middleware. Use .add_middleware() for automatic wrapping of callables.

add_command (*subcmd*)

Add a Command, and all of its subcommands, as a subcommand of this Command.

Middleware from the current command is layered on top of the subcommand's. An exception may be raised if there are conflicting middlewares or subcommand names.

add_middleware (*mw*)

Add a single middleware to this command. Outermost middleware should be added first. Remember: first added, first called.

get_dep_names (*path=()*)

Get a list of the names of all required arguments of a command (and any associated middleware).

By specifying *path*, the same can be done for any subcommand.

get_flag_map (*path=()*, *with_hidden=True*)

Command's get_flag_map differs from Parser's in that it filters the flag map to just the flags used by the endpoint at the associated subcommand *path*.

prepare (*paths=None*)

Compile and validate one or more subcommands to ensure all dependencies are met. Call this once all flags, subcommands, and middlewares have been added (using .add()).

This method is automatically called by .run() method, but it only does so for the specific subcommand being invoked. More conscientious users may want to call this method with no arguments to validate that all subcommands are ready for execution.

run (*argv=None*, *extras=None*, *print_error=None*)

Parses arguments and dispatches to the appropriate subcommand handler. If there is a parse error due to invalid user input, an error is printed and a CommandLineError is raised. If not caught, a CommandLineError will exit the process, typically with status code 1. Also handles dispatching to the appropriate HelpHandler, if configured.

Defaults to handling the arguments on the command line (`sys.argv`), but can also be explicitly passed arguments via the *argv* parameter.

Parameters

- **`argv`** (*`list`*) – A sequence of strings representing the command-line arguments. Defaults to `sys.argv`.
- **`extras`** (*`dict`*) – A map of additional arguments to be made available to the subcommand's handler function.
- **`print_error`** (*`callable`*) – The function that formats/prints error messages before program exit on CLI errors.

Note: For efficiency, `run()` only checks the subcommand invoked by *argv*. To ensure that all subcommands are configured properly, call `prepare()`.

2.2.1 Command Exception Types

In addition to all the Parser-layer exceptions, a command or user endpoint function can raise:

class `face.`**`CommandLineError`**(*msg*, *code=1*)

A `FaceException` and `SystemExit` subtype that enables safely catching runtime errors that would otherwise cause the process to exit.

If instances of this exception are left uncaught, they will exit the process.

If raised from a `run()` call and `print_error` is True, face will print the error before reraising. See `face.Command.run()` for more details.

class `face.`**`UsageError`**(*msg*, *code=1*)

Application developers should raise this `CommandLineError` subtype to indicate to users that the application user has used the command incorrectly.

Instead of printing an ugly stack trace, Face will print a readable error message of your choosing, then exit with a nonzero exit code.

2.3 Middleware

Coming soon!

- Dependency injection (like pytest!)
- autodoc * inventory of all production-grade middlewares

2.4 Testing

Face provides a full-featured test client for maintaining high-quality command-line applications.

class `face.`**`CommandChecker`**(*cmd*, *env=None*, *chdir=None*, *mix_stderr=False*, *reraise=False*)

Face's main testing interface.

Wrap your `Command` instance in a `CommandChecker`, `run()` commands with arguments, and get `RunResult` objects to validate your Command's behavior.

Parameters

- **`cmd`** – The *`Command`* instance to test.
- **`env`** (*`dict`*) – An optional base environment to use for subsequent calls issued through this checker. Defaults to `{}`.
- **`chdir`** (*`str`*) – A default path to execute this checker's commands in. Great for temporary directories to ensure test isolation.
- **`mix_stderr`** (*`bool`*) – Set to `True` to capture stderr into stdout. This makes it easier to verify order of standard output and errors. If `True`, this checker's results' error_bytes will be set to `None`. Defaults to `False`.
- **`reraise`** (*`bool`*) – Reraise uncaught exceptions from within *cmd*'s endpoint functions, instead of returning a `RunResult` instance. Defaults to `False`.

`run`(*args*, *input=None*, *env=None*, *chdir=None*, *exit_code=0*)

The *`run()`* method acts as the primary entrypoint to the *`CommandChecker`* instance. Pass arguments as a list or string, and receive a `RunResult` with which to verify your command's output.

If the arguments do not result in an expected *exit_code*, a `CheckError` will be raised.

Parameters

- **`args`** – A list or string representing arguments, as one might find in `sys.argv` or at the command line.
- **`input`** (*`str`*) – A string (or list of lines) to be passed to the command's stdin. Used for testing *`prompt()`* interactions, among others.
- **`env`** (*`dict`*) – A mapping of environment variables to apply on top of the *`CommandChecker`*'s base env vars.
- **`chdir`** (*`str`*) – A string (or stringifiable path) path to switch to before running the command. Defaults to `None` (runs in current directory).
- **`exit_code`** (*`int`*) – An integer or list of integer exit codes expected from running the command with *args*. If the actual exit code does not match *exit_code*, `CheckError` is raised. Set to `None` to disable this behavior and always return `RunResult`. Defaults to `0`.

Note: At this time, *`run()`* interacts with global process state, and is not designed for parallel usage.

`fail`(**a*, ***kw*)

Convenience method around *`run()`*, with the same signature, except that this will raise a `CheckError` if the command completes with exit code `0`.

`fail_X`()

Test that a command fails with exit code `X`, where `X` is an integer.

For testing convenience, any method of pattern `fail_X()` is the equivalent to `fail(exit_code=X)`, and `fail_X_Y()` is equivalent to `fail(exit_code=[X, Y])`, providing `X` and `Y` are integers.

class `face.testing.`**`RunResult`**(*args*, *input*, *exit_code*, *stdout_bytes*, *stderr_bytes*, *exc_info=None*, *checker=None*)

Returned from `CommandChecker.run()`, complete with the relevant inputs and outputs of the run.

Instances of this object are especially valuable for verifying expected output via the *`stdout`* and *`stderr`* attributes.

API modeled after `subprocess.CompletedProcess` for familiarity and porting of tests.

args
The arguments passed to `run()`.

input
The string input passed to the command, if any.

exit_code
The integer exit code returned by the command. `0` conventionally indicates success.

stdout
The text output ("stdout") of the command, as a decoded string. See `stdout_bytes` for the bytestring.

stderr
The error output ("stderr") of the command, as a decoded string. See `stderr_bytes` for the bytestring. May be `None` if *mix_stderr* was set to `True` in the `CommandChecker`.

stdout_bytes
The output ("stdout") of the command, as an encoded bytestring. See `stdout` for the decoded text.

stderr_bytes
The error output ("stderr") of the command, as an encoded bytestring. See `stderr` for the decoded text. May be `None` if *mix_stderr* was set to `True` in the `CommandChecker`.

returncode
Alias of `exit_code`, for parity with `subprocess.CompletedProcess`

exc_info
A 3-tuple of the internal exception, in the same fashion as `sys.exc_info()`, representing the captured uncaught exception raised by the command function from a `CommandChecker` with *reraise* set to `True`. For advanced use only.

exception
Exception instance, if an uncaught error was raised. Equivalent to `run_res.exc_info[1]`, but more readable.

exception `face.testing.`**CheckError**(*result*, *exit_codes*)
Rarely raised directly, `CheckError` is automatically raised when a `CommandChecker.run()` call does not terminate with an expected error code.

This error attempts to format the stdout, stderr, and stdin of the run for easier debugging.

2.5 Input / Output

Face includes a variety of utilities designed to make it easy to write applications that adhere to command-line conventions and user expectations.

`face.`**echo**(*msg*, ***kw*)
A better-behaved `print()` function for command-line applications.

Writes text or bytes to a file or stream and flushes. Seamlessly handles stripping ANSI color codes when the output file is not a TTY.

```
>>> echo('test')
test
```

Parameters

- **msg** (*str*) – A text or byte string to echo.
- **err** (*bool*) – Set the default output file to `sys.stderr`

- **file** (*file*) – Stream or other file-like object to output to. Defaults to `sys.stdout`, or `sys.stderr` if *err* is True.
- **nl** (*bool*) – If `True`, sets *end* to `'\n'`, the newline character.
- **end** (*str*) – Explicitly set the line-ending character. Setting this overrides *nl*.
- **color** (*bool*) – Set to `True`/`False` to always/never echo ANSI color codes. Defaults to inspecting whether *file* is a TTY.

`face.`**`echo_err`**(**a*, ***kw*)

A convenience function which works exactly like `echo()`, but always defaults the output *file* to `sys.stderr`.

`face.`**`prompt`**(*label*, *confirm=None*, *confirm_label=None*, *hide_input=False*, *err=False*)

A better-behaved `input()` function for command-line applications.

Ask a user for input, confirming if necessary, returns a text string. Handles Ctrl-C and EOF more gracefully than Python's built-ins.

Parameters

- **label** (*str*) – The prompt to display to the user.
- **confirm** (*bool*) – Pass `True` to ask the user to retype the input to confirm it. Defaults to False, unless *confirm_label* is passed.
- **confirm_label** (*str*) – Override the confirmation prompt. Defaults to "Retype *label*" if *confirm* is `True`.
- **hide_input** (*bool*) – If `True`, disables echoing the user's input as they type. Useful for passwords and other secret entry. See `prompt_secret()` for a more convenient interface. Defaults to `False`.
- **err** (*bool*) – If `True`, prompts are printed on `sys.stderr`. Defaults to `False`.

`prompt()` is primarily intended for simple plaintext entry. See `prompt_secret()` for handling passwords and other secret user input.

Raises `UsageError` if *confirm* is enabled and inputs do not match.

`face.`**`prompt_secret`**(*label*, ***kw*)

A convenience function around `prompt()`, which is preconfigured for secret user input, like passwords.

All arguments are the same, except *hide_input* is always `True`, and *err* defaults to `True`, for consistency with `getpass.getpass()`.

2.5.1 TODO

- TODO: InputCancelled exception, to be handled by .run()
- TODO: stuff for prompting choices
- TODO: pre-made –color flag(s) (looks at isatty)

2.6 Face FAQs

TODO

2.6.1 What sets Face apart from other CLI libraries?

In the Python world, you certainly have a lot of choices among argument parsers. Software isn't a competition, but there are many good reasons to choose face.

- Rich dependency semantics guarantee that endpoints and their dependencies line up before the Command will build to start up.
- Streamlined, Pythonic API.
- Handy testing tools
- Focus on CLI UX (arg order, discouraging required flag)
- TODO: contrast with argparse, optparse, click, etc.

2.6.2 Why is Face so picky about argument order?

In short, command-line user experience and history hygiene. While it's easy for us to be tempted to add flags to the ends of commands, anyone reading that command later is going to suffer:

```
cmd subcmd posarg1 --flag arg posarg2
```

Does `posarg2` look more like a positional argument or an argument of `--flag`?

This is also why Face doesn't allow non-leaf commands to accept positional arguments (is it a subcommand or an argument?), or flags which support more than one whitespace-separated argument.

2.6.3 Any recommended patterns for laying out CLI code?

- Dedicated cli.py which constructs commands.
- main function should take argv as an argument
- `if __name__ == '__main__':  main(sys.argv)`
- Entrypoints are nicer than `-m`

Index