terminal Documentation

Release 0.4.0

Hsiaoming Yang

November 12, 2015

Contents

Terminal is designed for creating terminal-based tools, written in Python, for developers who like simple and beautiful things.

Python's standard **logging** and **argparse** (or optparse) modules are powerful, but the API is thoroughly ugly and hard to use. Terminal is designed to make life simpler and better.

It contains everything you need for terminal:

- Argv Parser
- ANSI Colors
- Prompts
- Logging

```
from terminal import Command, red

program = Command('foo', version='1.0.0')

@program.action
def show(color=True):
    """
    show something.

    :param color: disable or enable colors
    """
    if color:
        print(red('show'))
    else:
        print('show')
```

CHAPTER 1

User's Guide

This part of the documentation, which is mostly prose, begins with some background information about Terminal, then focuses on step-by-step instructions for getting the most out of Terminal.

1.1 Introduction

Terminal is designed for creating terminal-base tools, written in Python, for developers who like simple and beautiful things.

1.1.1 import this

Terminal was developed with a few **PEP 20** idioms in mind:

```
>>> import this
```

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Readability counts.

All contributions to Terminal should keep these important rules in mind.

1.1.2 License

A large number of open source projects in Python are BSD Licensed, and Terminal is released under BSD License too.

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.
- Neither the name of the creator nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

1.2 Installation

This part of the documentation covers the installation of Terminal. The first step to using any software package is getting it properly installed.

1.2.1 Distribute & Pip

Installing terminal is simple with pip:

```
$ pip install terminal
```

or, with easy_install:

```
$ easy_install terminal
```

But, you really shouldn't do that.

1.2.2 Cheeseshop Mirror

If the Cheeseshop is down, you can also install Terminal from one of the mirrors. Crate.io is one of them:

```
$ pip install -i http://simple.crate.io/ terminal
```

1.2.3 Get the Code

Terminal is actively developed on GitHub, where the code is always available.

You can either clone the public repository:

```
git clone git://github.com/lepture/terminal.git
```

Download the tarball:

```
$ curl -OL https://github.com/lepture/terminal/tarball/master
```

Or, download the zipball:

```
$ curl -OL https://github.com/lepture/terminal/zipball/master
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily:

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

1.3 Quickstart

Eager to get started? This page gives a good introduction in how to get started with Terminal. This assumes you already have Terminal installed. If you do not, head over to the Installation section.

First, make sure that:

- Terminal is installed

Let's get started with some simple examples.

1.3.1 Play with colors

Terminal would be better if it is colorful.

Begin by importing the Terminal module:

```
>>> import terminal
```

Now, let's play with the colors:

```
>>> print(terminal.red('red color'))
>>> print(terminal.green_bg('green background'))
```

Available colors:

```
black, red, green, yellow, blue, magenta, cyan, white
```

Available styles:

```
bold, faint, italic, underline, blink, overline, inverse
conceal, strike
```

Warning: But don't count on these styles, your terminal may be not able to show all of them.

1.3.2 Verbose logging

Terminal programs need a simple and beautiful logging system. If it has a verbose feature, it could help a lot:

```
>>> from terminal import log
```

Now, let's play with the verbose log:

```
>>> log.info('play with the log')
>>> log.verbose.info('a verbose log will not show')
>>> log.config(verbose=True)
>>> log.verbose.info('a verbose log will show')
```

We can also control the logging level:

```
>>> log.config(quiet=True)
>>> log.info('info log will not show')
>>> log.warn('but warn and error messages will show')
```

1.3.3 Prompt communication

Many terminal programs will communicate with the users, this could be easy with *prompt()*.

Let's create a prompt to ask the user's name:

```
import terminal

username = terminal.prompt('what is your name')
```

We could set a default name for the user:

```
username = terminal.prompt('what is your name', 'Kate')
```

It is not a good idea to get a password with *prompt()*, instead, terminal provided a *password()* for you:

```
password = terminal.password('what is your password')
```

Want more on prompt?

We have *terminal.confirm()* and *terminal.choose()*.

1.3.4 Command line

This is a replacement for **argparse** (or optparse).

Create a simple command parser with *Command*:

```
program = Command('foo', 'a description')
```

Add some options:

```
program.option('-f, --force', 'force to process')
program.option('-o, --output [output]', 'the output directory')
```

Let's make it work:

```
program.parse()

if program.output:
   print program.output
```

Save the code in a file (for example `foo.py`), play in the terminal:

```
$ python foo.py -h
$ python foo.py -o src
$ python foo.py --output=src
$ python foo.py --output src
```

However, when creating a terminal tool, a subcommand is usually needed, we can add subcommands via *Command.action()*:

```
program = Command('foo', 'a description')
program.option('-v, --verbose', 'show more logs')

subcommand = Command('build', 'build the site')
subcommand.option('-o, --output [output]', 'the output directory')

program.action(subcommand)

program.parse()

if program.verbose:
    terminal.log.config(verbose=True)
```

Let's play with the more complex one:

```
$ python foo.py -h
$ python foo.py build -h
```

Ready for more? Check out the Advanced Usage section.

1.4 Advanced Usage

This document covers some of Terminal more advanced features.

1.4.1 Colorful Life

If you are not satisfied with 8-bit colors, we can have more. The *colorize()* function can paint 256 colors if your terminal supports 256 colors:

```
>>> from terminal import colorize
>>> print colorize('text', 'red')        # color name
text
>>> print colorize('text', '#ff0000')    # hex color with #
text
>>> print colorize('text', 'ff0000')     # hex color without #
text
>>> print colorize('text', (255, 0, 0))  # rgb color
text
```

Note: If your terminal can not show 256 colors, maybe you should change your terminal profile, claim that it supports 256 colors.

We can also paint the background of the text with *colorize()*:

```
>>> print colorize('text', 'ff0000', background=True)
```

The source engine of *colorize()*, *red()*, *cyan()* and etc. is *Color* object. Let's dig into it:

```
>>> from terminal import Color
>>> s = Color('text')
>>> s.fgcolor = 1
>>> print s
text
```

```
>>> s.bgcolor = 2
>>> print s
text
>>> s.styles = [1, 4]
>>> print s
text
```

But it is not fun to play with ANSI code, we like something that we can read:

```
>>> s = Color('text')
>>> s.fgcolor = 'red'
>>> s.bgcolor = 'green'
>>> print s
```

This is not good enough, if we want to paint a text in red text, green background, bold and underline styles, we could:

```
>>> from terminal import *
>>> print underline(bold(green_bg(red('text'))))
text
>>> # this is a disaster, we can do better
...
>>> s = Color('text')
>>> print s.red.green_bg.bold.underline
text
```

Important: If you are colorizing non-ASCII charaters, and you are not on Python 3. Please do wrap the charaters with **u**:

```
>>> terminal.red(u'')
```

1.4.2 Decorator Command

Command is inspired by commander.js, with the magic power of Python, we can do better. Let's have a look at the basic usage of *Command*:

```
from terminal import Command

program = Command('pip', description='.....', version='1.2.1')
program.option('--log [filename]', 'record the log into a file')
program.option(
    '--timeout [seconds]',
    'the socket timeout, default: 15',
    resolve=int
)

# let's create a subcommand
installer = Command('install', description='install packages')
installer.option('-t, --target [dir]', 'Install packages into dir')

# let's add the subcommand
program.action(installer)

program.parse()
```

The magic of decorator makes it easier to add a subcommand:

```
@program.action
def install(target=None):
    """
    install packages

    :param target: Install packages into dir
    """
    do_something(target)
```

The decorator will create a subcommand for you automatically. It will get the options from parameters, it will get the description from the docstring.

The example above equals something like:

```
def install(target=None):
    do_something(target)

installer = Command('install', description='install packages', func=install)
installer.option('-t, --target [target]', 'Install packages into dir')
program.action(installer)
```

The option `-t, --target [target]` is generated from params and docstring. You can define the option yourself:

```
@program.action
def install(target=None):
    """
    install packages

    :param target: Install packages into dir
    :option target: -g, --target [dir]
    """
    do_something(target)
```

The `arguments` parameter was added in 0.4.0, and it can be generated from a function:

```
@program.action
def install(target):
    """install a package"""
    do_something(target)
```

The usage will be:

```
$ pip install <target>
```

If the `target` has a description, it will be a required option:

```
@program.action
def install(target):
    """
    install packages

    :param target: Install packages into dir
    """
    do_something(target)
```

The usage will be:

```
$ pip install -t <target>
```

Options

If you defined a subcommand with the decorator magic, *Command* will auto detect if the option is required or not, if it is a Boolean or not.

If you do like the raw option, we can learn from these examples:

```
# boolean, default is False
program.option('-f, --force', 'force to do something')
# program.force is False

# boolean, default is True
program.option('-C, --no-color', 'do not paint')
# program.color is True

# required, no default value
program.option('-o, --output <dir>', 'output directory')

# required, default is dist
program.option('-o, --output <dir>', 'output directory, default: dist')
# if it has a default value, it is optional actually.

# optional, no default value
program.option('-o, --output [dir]', 'output directory')
```

You can learn from the example that required options are surrounded by <>, and optional ones are surrounded by `[]`. The parser can parse a default value from the description.

1.4.3 Builtin Engines

We do like colorful things, but we are too lazy to do any customization. The default one is pretty enough, (if you don't think so, let's improve it).

Get the power from **terminal.builtin**:

```
from terminal.builtin import Command, log
```

CHAPTER 2

API Documentation

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 Developer Interface

This part of the documentation covers the interface of terminal. Terminal only depends on Python's built-in batteries, it is lightweight.

2.1.1 Command

Command is the interface which you can create a command parser.

class terminal.**Command**(*name*, *description=None*, *version=None*, *usage=None*, *title=None*, *func=None*, *help_footer=None*, *arguments=None*)

The command interface.

New in version 0.4: The *arguments* parameters were added.

Parameters

- **name** – name of the program
- **description** – description of the program
- **version** – version of the program
- **usage** – usage of the program
- **title** – title of the program
- **func** – command function to be invoked
- **arguments** – positional arguments

Create a *Command* instance in your cli file:

```
from terminal import Command
command = Command('name', 'description', 'version')
```

The command will add a default help option. If you set a version parameter, it will add a default version option too. You can add more options:

```
command.option('-v, --verbose', 'show more logs')

# required option in < and >
command.option('-o, --output <output>', 'the output file')

# optional option in [ and ]
command.option('-o, --output [output]', 'the output file')

# set a default value in description
command.option('-o, --source [dir]', 'the output file, default: src')
```

Usually you will need a subcommand feature, add a subcommand like this:

```
subcommand = Command('build', 'build command')
subcommand.option('--source <source>', 'source directory')

command.action(subcommand)
```

A subcommand is a full featured Command, which means you can add a subcommand to the subcommand too. We can also add a subcommand with the decorator feature:

```
@command.action
def build(output='_build'):
    '''
    generate the documentation.

    :param output: the output directory
    '''
    do_something(output)
```

After defining the command, we can parse the command line argv:

```
command.parse()
```

If we have pased `--verbose` in the terminal, we can get:

```
assert command.verbose is True
```

action(*command*)

Add a subcommand. (Alias *subcommand*).

Parameters **command** – a function or a Command

You can add a *Command* as an action, or a function:

```
command.action(subcommand)
```

If you prefer a decorator:

```
@command.action
def server(port=5000):
    '''
    docstring as the description

    :param port: description of port
    '''
    start_server(port)
```

It will auto generate a subcommand from the `server` method.

args
argv that are not for options.

get(*key*)
Get parsed result.

After `parse()` the argv, we can get the parsed results:

```
# command.option('-f', 'description of -f')
command.get('-f')

command.get('verbose')
# we can also get ``verbose``: command.verbose
```

option(*name*, *description=None*, *action=None*, *resolve=None*)
Add or get option.

Here are some examples:

```
command.option('-v, --verbose', 'show more log')
command.option('--tag <tag>', 'tag of the package')
command.option('-s, --source <source>', 'the source repo')
```

Parameters

- **name** – arguments of the option
- **description** – description of the option
- **action** – a function to be invoked
- **resolve** – a function to resolve the data

parse(*argv=None*)
Parse argv of terminal

Parameters **argv** – default is sys.argv

parse_options(*arg*)
Parse options with the argv

Parameters **arg** – one arg from argv

print_help()
Print the help menu.

print_title(*title*)
Print output the title.

You can create a colorized title by:

```
class MyCommand(Command):

    def print_title(self, title):
        if 'Options' in title:
            print(terminal.magenta(title))
        elif 'Commands' in title:
            print(terminal.green(title))
        return self
```

You can get the color function with `terminal`.

print_version()
Print the program version.

subcommand(*command*)
Alias for Command.action.

validate_options()
Validate options

class terminal.**Option**(*name*, *description=None*, *action=None*, *resolve=None*)
The Option object for Command.

Parameters

- **name** – the name of the option, usually like `-v, --verbose`
- **description** – the description of this option
- **action** – a function to be invoke when this option is found
- **resolve** – a function to transform the option data

parse(*name*, *description*)
Parse option name.

Parameters

- **name** – option's name
- **description** – option's description

Parsing acceptable names:

•-f: shortname

•–force: longname

•-f, –force: shortname and longname

•-o <output>: shortname and a required value

•-o, –output [output]: shortname, longname and optional value

Parsing default value from description:

•source directory, default: src

•source directory, default: [src]

•source directory, default: <src>

to_python(*value=None*)
Transform the option value to python data.

2.1.2 Logger

Logger is the interface which you can create a logging instance. We do have a initialized instance on terminal:

```
>>> terminal.log.info('this is the default log instance')
```

class terminal.**Logger**(***kwargs*)
The Logger interface.

Parameters

- **verbose** – control if the log to show verbose log

- **quiet** – control if the log to show debug and info log
- **indent** – control the indent level, default is zero

Play with *Logger*, it supports nested logging:

```
log = Logger()
log.info('play with log')

log.start('start a indent level')
log.info('this log will indent two spaces')
log.end('close a indent level')
```

config(***kwargs*)
Config the behavior of *Logger*.

Control the output to show *Logger.verbose* log:

```
log.config(verbose=True)
```

Control the output to show only the *Logger.warn* and *Logger.error* log:

```
log.config(quiet=True)
```

debug(**args*)
The debug level log.

end(**args*)
End a nested log.

error(**args*)
The error level log.

info(**args*)
The info level log.

message(*level*, **args*)
Format the message of the logger.

You can rewrite this method to format your own message:

```
class MyLogger(Logger):

    def message(self, level, *args):
        msg = ' '.join(args)

        if level == 'error':
            return terminal.red(msg)
        return msg
```

start(**args*)
Start a nested log.

verbose
Make it the verbose log.

A verbose log can be only shown when user want to see more logs. It works as:

```
log.verbose.warn('this is a verbose warn')
log.verbose.info('this is a verbose info')
```

warn(**args*)
The warn level log.

2.1.3 Prompt

Prompt are functions to communicate with the terminal.

terminal.**prompt**(*name*, *default=None*)
Grab user input from command line.

Parameters

- **name** – prompt text
- **default** – default value if no input provided.

terminal.**password**(*name*, *default=None*)
Grabs hidden (password) input from command line.

Parameters

- **name** – prompt text
- **default** – default value if no input provided.

terminal.**confirm**(*name*, *default=False*, *yes_choices=None*, *no_choices=None*)
Grabs user input from command line and converts to boolean value.

Parameters

- **name** – prompt text
- **default** – default value if no input provided.
- **yes_choices** – default 'y', 'yes', '1', 'on', 'true', 't'
- **no_choices** – default 'n', 'no', '0', 'off', 'false', 'f'

terminal.**choose**(*name*, *choices*, *default=None*, *resolve=None*, *no_choice=('none',)*)
Grabs user input from command line from set of provided choices.

Parameters

- **name** – prompt text
- **choices** – list or tuple of available choices. Choices may be single strings or (key, value) tuples.
- **default** – default value if no input provided.
- **no_choice** – acceptable list of strings for "null choice"

2.1.4 Colors

class terminal.**Color**(**items*)
Color object for painters.

You should always use the high-level API of colors and styles, such as *red* and *bold*.

But if you are so interested in this module, you are welcome to use some advanced features:

```
s = Color('text')
print(s.bold.red.italic)
```

All ANSI colors and styles are available on Color.

terminal.**colorize**(*text*, *color*, *background=False*)
Colorize text with hex code.

Parameters

- **text** – the text you want to paint
- **color** – a hex color or rgb color
- **background** – decide to colorize background

```
colorize('hello', 'ff0000')
colorize('hello', '#ff0000')
colorize('hello', (255, 0, 0))
```

Front colors

terminal.**black**(*text*)
Black color.

terminal.**red**(*text*)
Red color.

terminal.**green**(*text*)
Green color.

terminal.**yellow**(*text*)
Yellow color.

terminal.**blue**(*text*)
Blue color.

terminal.**magenta**(*text*)
Magenta color.

terminal.**cyan**(*text*)
Cyan color.

terminal.**white**(*text*)
White color.

Background colors

terminal.**black_bg**(*text*)
Black background.

terminal.**red_bg**(*text*)
Red background.

terminal.**green_bg**(*text*)
Green background.

terminal.**yellow_bg**(*text*)
Yellow background.

terminal.**blue_bg**(*text*)
Blue background.

terminal.**magenta_bg**(*text*)
Magenta background.

terminal.**cyan_bg**(*text*)
Cyan background.

terminal.**white_bg**(*text*)
White background.

Styles

terminal.**bold**(*text*)
Bold style.

terminal.**faint**(*text*)
Faint style.

terminal.**italic**(*text*)
Italic style.

terminal.**underline**(*text*)
Underline style.

terminal.**blink**(*text*)
Blink style.

terminal.**overline**(*text*)
Overline style.

terminal.**inverse**(*text*)
Inverse style.

terminal.**conceal**(*text*)
Conceal style.

terminal.**strike**(*text*)
Strike style.

CHAPTER 3

Additional Notes

Contribution guide, legal information and changelog are here.

3.1 Contributing

First, please do contribute! There are more than one way to contribute, and I will appreciate any way you choose.

- introduce Terminal to your friends, let Terminal to be known
- discuss Terminal, and submit bugs with GitHub issues
- improve documentation for Terminal
- send patch with GitHub pull request

English and Chinese issues are acceptable, talk in your favorite language.

Pull request and git commit message **must be in English**, if your commit message is in other language, it will be rejected.

3.1.1 Issues

When you submit an issue, please format your content, a readable content helps a lot. You should have a little knowledge on Markdown.

Code talks. If you can't make yourself understood, show me the code. Please make your case as simple as possible.

3.1.2 Codebase

The codebase of Terminal is highly tested and **PEP 8** compatible, as a way to guarantee functionality and keep all code written in a good style.

You should follow the code style, and if you add any code, please add test cases for them.

Here are some tips to make things simple:

- When you cloned this repo, run `make`, it will prepare everything for you
- Check the code style with `make lint`
- Check the test cases with `make test`
- Check test coverage with `make coverage`

3.1.3 Git Help

Something you should know about git.

- don't add any code on the master branch, create a new one
- don't add too many code in one pull request
- all featured branches should be based on the master branch

Take an example, if you want to add feature A and feature B, you should have two branches:

```
$ git branch feature-A
$ git checkout feature-A
```

Now code on feature-A branch, and when you finish feature A:

```
$ git checkout master
$ git branch feature-B
$ git checkout feature-B
```

All branches must be based on the master branch. If your feature-B needs feature-A, you should send feature-A first, and wait for its merging. We may reject feature-A, and you should stop feature-B.

3.2 Changelog

Here you can see the full list of changes between each Terminal release.

3.2.1 Version 0.4.0

Release on Dec 20, 2013. Beta release.

- Add alias *Command.subcommand()*
- Add *arguments* parameter for *Command*
- Reture False and True for *Command.parse()*

3.2.2 Version 0.3.0

Released on May 7, 2013. Beta release.

- various bugfixed
- delete an newline in print help
- remove support for pure function as action
- config option in docstring
- add help_footer option

3.2.3 Version 0.2.0

Released on April 23, 2013. Beta release.

- various bugfixed

- don't sys.exit on *Command.print_help()* and *Command.print_version()*
- add *Option*, use *Option* on *Command.option()*
- remove the colorful message on *Logger.message()*
- disable *Logger.verbose()* log on *Logger.start()* and *Logger.end()*
- add **builtin.Command** and **builtin.Logger**
- Option parser has required and boolean flags, Option parser has default value
- record warn and error count on *Logger*

3.2.4 Version 0.1.0

First public preview release.

3.3 Authors

Terminal is written and maintained by Hsiaoming Yang <me@lepture.com>.

3.3.1 Contributors

People who send patches and suggestions:

- Hsiaoming Yang http://github.com/lepture
- Xiao Meng https://github.com/reorx
- James Rowe https://github.com/JNRowe

Find more contributors on GitHub.

t