climenu Documentation

Release 1.8.0

Timothy McFadden

Jul 31, 2023

Contents:

`climenu` was designed to easily provided developers with a command-line menu system for interactive programs.

The basic idea is that you *decorate* the functions you want to show on your menu. Everything can be done using only decorators!

CHAPTER 1

The User Guide

1.1 Introduction

1.1.1 Philsophy

`climenu` was written to make interactive command-line applications easier to write. I really liked the way Click created thier interface using decorators, and I couldn't find anything in PyPi. So I wrote one!

1.1.2 License

`climenu` is released under the MIT license as follows:

1.2 Installation

1.2.1 Install using `pip`

As long as you have `pip` installed, you can install `climenu` using:

```
$ pip install climenu
```

1.2.2 Install from source

`climenu` is hosted on GitHub and Pypi. You can download the latest release from GitHub or pypi.

Unzip the tarball, and run `setup.py`:

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

1.3 Quickstart

First, make sure that `climenu` is *installed*

The easiest way to use `climenu` is to create a *flat* menu like so:

```
import climenu

@climenu.menu()
def build_packages():
    '''Build packages'''
    # TODO: Call the build scripts here!
    print('built the packages!')

@climenu.menu()
def build_release():
    '''Build release'''
    # TODO: Call the build scripts here!
    print('built the release!')


if __name__ == '__main__':
    climenu.run()
```

When you run this code, you will see the following menu:

```
Main Menu
1 : Build packages
2 : Build release

Enter the selection (0 to exit):
```

You can see that `climenu` is using the function docstring for the menu text.

1.4 Advanced

1.4.1 Menu Groups

You can create nested menus by using `@climenu.group`, then adding menus to that group.

The *group* is just an empty function decorated by `@climenu.group`:

```
import climenu

@climenu.group()
def build_group():
    '''Build Menu'''
    pass
```

Once the group is defined, you can add menu items to it by using a decorator consisting of the function name you used to create the group, and either `.menu` or another `.group`.

```
@build_group.menu()
def build_package():
    '''Build the package'''
    # TODO: Add the actual code here
    print("I built the package!")

@build_group.menu()
def build_release():
    '''Build the release'''
    # TODO: Add the actual code here
    print("I built the release!")
```

Caution: The name of the decorator changed from @climenu.group to @build_group!

Running this code (`climenu.run()`) will show a single menu item for the *main* menu:

```
Main Menu
1 : Build Menu

Enter the selection (0 to exit):
```

Once the user enters `1`, the second menu will be shown:

```
Build Menu
1 : Build the package
2 : Build the release

Enter the selection (0 to return):
```

You can have any number of groups associated with other groups.

Menu Group Subtitles

Menu groups can have an optional subtitle. This will be displayed on the line immediately following the title.

The subtitle can be either a `callable` (which returns a string) or a string.

The subtitle will only be shown when the menu group is the main menu displayed. That is, the subtitle will not be shown when the *main* menu is a list of menu groups.

> **Caution:** The callable return will not be cached. You may want to take steps to do that in case the callable is time consuming.

```
import climenu

@climenu.group(title='Build Menu', subtitle='building stuff')
def build_group(): pass
```

1.4.2 Menu Titles

`climenu` will use the function docstring as the default text displayed for the menu items. This can be changed by using the `title` parameter to the decorator.

For example, the following code would produce the same menus as above. Especially look at `build_release()`, where both `title` and docstring are used:

```
import climenu

@climenu.group(title='Build Menu')
def build_group(): pass

@build_group.menu(title='Build the package')
def build_package():
    # TODO: Add the actual code here
    print("I built the package!")

@build_group.menu(title='Build the release')
def build_release():
    '''
    Do a bunch of stuff to build the release.

        * Copy the files
        * Run some scripts
        * Build the release
        * Copy the release to X/Y/Z
    '''
    # TODO: Add the actual code here
    print("I built the release!")

def main():
    climenu.run()

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

You may also use a `callable` object to get the title of a menu item/group. This is handy if you need to run some code in order to calculate the title.

> **Caution:** The callable must return a string!

Caution: The callable return will not be cached. You may want to take steps to do that in case the callable is time consuming.

```
import climenu

def myfunction():
    return 'Build Menu'

@climenu.menu(title=myfunction)
def a_function(): pass
```

1.4.3 Mutliple Files

You can split up your menu files into multiple Python files. This is useful if you have lots of menus, or the menu actions are complex.

One example layout like so:

```
| main.py
| build_menu.py
| test_menu.py
```

main.py:

```
import build_menu
import test_menu
import climenu

if __name__ == '__main__':
    climenu.run()
```

build_menu.py

```
import climenu

@climenu.group(title='Build Menu')
def build_menu(): pass

@build_menu.menu(title='Build package')
def build_package():
    pass

@build_menu.menu(title='Build release')
def build_release():
    pass
```

test_menu.py

```
import climenu

@climenu.group(title='Test Menu')
def test_menu(): pass

@test_menu.menu(title='Run test #1')
```

(continues on next page)

(continued from previous page)

```
def test_one():
    pass

@test_menu.menu(title='Run test #2')
def test_two():
    pass
```

1.4.4 Dynamic Menu Items

A `MenuGroup` is made up of `Menu` items. Normally, you create these menu items using the `@climenu.menu` decorator. However, sometimes you don't know what these items should be until runtime.

In this case, you can use the `items_getter`, `items_getter_args`, and `items_getter_kwargs` parameters to the `@group` decorator.

`items_getter` is a callback function that returns a list of tuples in the form `(<item-title>, <callback-function>)`. If this function takes arguments, you'll need to also use some combination of `items_getter_args` (a list of arguments to pass to the callback function) and `items_getter_kwargs` (a dictionary of keyword arguments).

Note: You will need to use `functools.partial()` if the function(s) you are returning from `items_getter` takes any arguments. See example below.

`dynamic-group.py`

```
from functools import partial
import climenu


def print_var(variable):
    '''print the variable'''
    print(str(variable))


def build_items(count):
    # In this example, we're generating menu items based on some
    # thing that's determined at runtime (e.g. files in a directory).

    # For this case, we're simply using `xrange` to generate a range of
    # items.  The function that eventually gets called takes 1 argument.
    # Therefore, we need to use ``partial`` to pass in those arguments at
    # runtime.

    items = []
    for index in xrange(count):
        items.append(
            (
                'Run item %i' % (index + 1),
                partial(print_var, 'Item %i' % (index + 1))
            )
        )

    return items
```

(continues on next page)

(continued from previous page)

```
@climenu.menu(title='Do the first thing')
def first_thing():
    # A simple menu item
    print('Did the first thing!')


@climenu.group(items_getter=build_items, items_getter_kwargs={'count': 7})
def build_group():
    '''A dynamic menu'''
    # This is just a placeholder for a MenuGroup.  The items in the menu
    # will be dymanically generated when this module loads by calling
    # `build_items`.
    pass


if __name__ == '__main__':
    climenu.run()
```

1.5 Colors

climenu supports basic ANSI colors, including foreground, background, and *bright*.

Caution: Windows may have some strange affects when using ANSI formatting codes. It may not always work! YMMV

Caution: There's no way of knowing what color scheme your users are using in thier terminal. It's best to use colors sparingly, and perhaps always define a background.

1.5.1 ANSI Color Table

You can read more about the ANSI color table on wikipedia

The supported colors are: black, red, green, yellow, blue, magenta, cyan, and white.

1.5.2 Using Colors

You can use the colors by *wrapping* some text with the color codes. This is done using the `climenu.colors` object.

```
import climenu

# Print red text on the default background.
print(
    climenu.colors.red("Hello World!")
)
```

1.5.3 Nesting

The functions in `climenu.colors.*` return strings that contain the ANSI codes needed to format your text. This also means that you can *nest* these calls, or make multiple calls to create multi-formatted strings (e.g. bright-blue text on a yellow background).

```
import climenu

# Print bright-blue text on a yellow background
print(
    climenu.colors.yellow(
        climenu.colors.blue("Hello World!", bright=True),
        bg=True)
)

# Another way to do it
text = climenu.colors.blue("Hello World!", bright=True)
text = climenu.colors.yellow(text, bg=True)
```

1.5.4 Disabling Colors

You can disable colors completely by using `climenu.settings.disable_colors = True` in your code. This would only be needed in cases where you are using colors in some parts of the code, but you know you are running on a platform that doesn't support ANSI color codes.

1.5.5 Why is "yellow" showing up as orange?

That threw me too... A short explaination is given on wikipedia. If any of this make sense to you, good on ya!

> On terminals based on CGA compatible hardware, such as ANSI.SYS running on DOS, this normal intensity foreground color is rendered as Orange. CGA RGBI monitors contained hardware to modify the dark yellow color to an orange/brown color by reducing the green component

1.6 Settings

You can use the `settings` object to change the behavior and text displayed by climenu.

You would normally change the settings after importing the library like so:

```
import climenu
climenu.settings.clear_screen = True
```

1.6.1 Changing Behavior

The following parameters can be used to change how climenu operates

clear_screen (`bool`): Whether or not climenu clears the screen when displaying the menu.

back_values (`list`): The list of values entered by the user that will cause the menu to go back one level (if in a `group`), or exit the application.

This value defaults to `['0']`. If you want an empty input to also result in *back*, change this to `['0', '']`. This makes it so the user can keep pressing the `Enter` key to exit the application from any submenu.

quit_value (`string`): This value causes the menu system to exit immediately. It's also formatted in the defaults for `text['submenu_prompt']` and `text['main_menu_prompt']`.

For example, the default settings produce a main menu prompt that looks like:

```
Enter the selection ([0, q] to quit):
```

and a submenu prompt that looks like:

```
Enter the selection ([0] to return, q to quit):
```

quit_exit_code (`int`): This is the exit code of the python process that is returned when the user *quits* the menu.

breadcrumbs (`bool`): This controls whether or not *breadcrumbs* for the titles is enabled. When this setting is `True`, the title of the displayed menu will be changed to `Main Menu > Submenu 1 > Submenu 2`.

breadcrumb_join (`string`): This is the string that is used to *join* the menu titles. This is only used when `settings.breadcrumbs = True`.

1.6.2 Changing Displayed Text

The `settings` object has a `text` attribute that is a dictionary. The values of this dictionary are the text displayed during the course of menu operation (not to be confused with the text displayed for a menu item).

main_menu_title [This is the string that is displayed at the top of the first] menu.

main_menu_prompt [This is the string displayed at the bottom of the first menu] asking the user to either select and item or 'q' to exit the application. This includes a special formatter `{q}` that displays the text used to quit the application.

submenu_prompt [This is almost the same as `main_menu_prompt`, except the user] is prompted to select '0' to return (as opposed to *exit*). This includes two special formatting fields: `{q}`, that displays the text used to quit the application, and `{back}`, which is used to show the user what to enter to go back **up** on level of menu.

invalid_selection [This is the text presented to the user if they make an invalid] selection.

continue [This is the text presented to the user after a menu item has been] executed.

disable_color [Set this to *True* if you have code that uses colors, but] you want to disable it globally. This can be useful if you have code that runs on multiple platforms but you need to disable colors on an unsupported machine.

> **Caution:** Disabling colors should come prior to any color use! For example, if you use colors in the other settings (*text*, *main_menu_title*, etc).

1.6.3 Example

Here's an example showing all of the options:

```
import climenu
climenu.settings.clear_screen = False
climenu.settings.back_values = ['0', '', 'argh!']
climenu.settings.text['main_menu_title'] = 'My Sweet Application'
climenu.settings.text['main_menu_prompt'] = 'Enter the selection (0 to exit the␣
↪application): '
climenu.settings.text['submenu_prompt'] = 'Enter the selection (0 to return to␣
↪previous menu): '
```

(continues on next page)

(continued from previous page)

```
climenu.settings.text['invalid_selection'] = "WTH?  I don't understand... try again! "
climenu.settings.text['continue'] = 'All done with that; Press enter to go again. '

@climenu.menu(title='Item 1')
def item_1(): pass
```