Chalky Documentation

Release 1.0.0

Stephen Bunn <stephen@bunn.io>

Jan 19, 2021

CONTENTS

Yet another terminal text coloring library...

Why? Because, I like certain things and I hate certain things about the currently available solutions. This here is my attempt to build an interface for simply applying ANSI escape sequences to strings that I enjoy and can update at my own free will. That is it, there is nothing new or interesting that this packages adds. Thanks

```
from chalky import bg, fg, hex, rgb, sty

# compose some styles together
print(fg.red & sty.bold | "Bold and red text")
print(bg.blue & fg.white & sty.italic | "White italic text on a blue background")

# store a style for later use
success_style = fg.green
print(success_style | "Success message")
print(success_style & sty.underline | "Underlined success message")

# build some true colors as well
print(rgb(255, 9, 255) | "Purply text")
print(hex("#ffdd00") & sty.bold | "Bold yellowy text")
```

To get started using this package, please see the *Getting Started* page!

CHAPTER

ONE

GETTING STARTED

Welcome to Chalky!
This page should hopefully provide you with enough information to get you started using Chalky.

1.1 Installation and Setup

Installing the package should be super duper simple as we utilize Python's setuptools.

```
$ poetry add chalky
$ # or if you're old school...
$ pip install chalky
```

Or you can build and install the package from the git repo.

```
$ git clone https://github.com/stephen-bunn/chalky.git
$ cd ./chalky
$ python setup.py install
```

1.2 Usage

Now with Chalky installed we can start applying some styles to text. Styles and colors are applied to text through a single *Chalk* instance that contains the desired format for styling a string. This instance is reusable and does not require the user to manually define when reset escape sequences need to be sent.

1.2.1 Creating Chalk

The *Chalk* class is simply a container storing the basic styles and colors that can be applied to a string. The stored rules try as best as possible to be agnostic to the `interfaces` the styles are going to be built with.

Chalk instances can contain:

- A set of *Style*
- A foreground color (*Color* or *TrueColor*)
- A background color (*Color* or *TrueColor*)

Constructing instances is pretty straightforward:

```
from chalky import Chalk
from chalky.style import Style
from chalky.color import Color

my_chalk = Chalk(
    styles={Style.BOLD, Style.ITALIC},
    foreground=Color.GREEN,
    background=Color.WHITE
)
```

1.2.2 Applying Chalk to Strings

Now that you have a *Chalk* instance to work with, you can apply it to a string using either the | or + operators. Or you can simply call the chalk instance with the desired string.

```
print(my_chalk | "Hello, World!")
print(my_chalk + "Hello, World!")
print(my_chalk("Hello, World!"))
```

When applying the chalk instance to a string, it will build the appropriate ANSI escape sequences to style the string and automatically add the reset sequence to the end of the string.

1.2.3 Composing Chalk

These *Chalk* instances can be composed together using the & or + operators.

```
from chalky import Chalk
from chalky.style import Style
from chalky.color import Color

my_chalk = Chalk(style={Style.BOLD}) & Chalk(foreground=Color.RED)
my_chalk = Chalk(style={Style.BOLD}) + Chalk(foreground=Color.RED)
```

The styles provided in the instance **being** applied will override any existing styles on the starting instance.

```
my_chalk = Chalk(foreground=Color.RED) & Chalk(foreground=Color.BLUE)
assert my_chalk.foreground == Color.BLUE
```

1.2.4 Chaining Chalk

Chaining together multiple styles and colors is another typical interface that people like to use for text coloring. We provide a *Chain* class that produces a *Chalk* for quick and easy production:

```
from chalky import chain

print(chain.green.bold | "I'm bold green text")
print(chain.italic.white.bg.blue | "I'm italic white text on blue background")
```

Using *Chain* classes should be pretty similar to how you use *Chalk* instances. You can compose them with other chains or chalks and apply them to strings just like chalk instances. They ultimately just provide a different interface for constructing the chalk instance and quickly consuming it.

1.2.5 Chalk Shortcuts

Since it can be pretty darn tedious to create instances of *Chalk* all the time, I threw in some pre-initialized chalk in the *shortcuts* module.

From this module we export `fg` (foreground), `bg` (background), and `sty` (style) namespaces to make it easy and quick to compose custom chalk instances:

```
from chalky import fg, bg, sty

debug = sty.dim & fg.white
success = fg.green & sty.bold
error = fg.red & sty.bold
critical = bg.red & fg.white


print(debug | "This is a DEBUG message")
print(success | "This is a SUCCESS message")
print(error | "This is a ERROR message")
print(critical | "This is a CRITICAL message")
```

You can quickly produce truecolor's as well (if your terminal supports them) by using the *hex()* or *rgb()* functions to quickly produce *TrueColor* instances:

```
from chalky import hex, rgb

custom_rgb = rgb(102, 102, 255) & sty.underline
custom_hex = hex("#90ff9c", background=True) & fg.black & sty.bold

print(custom_rgb | "Potential link text")
print(custom_hex | "Black on green text")
```

CHAPTER

TWO

CONTRIBUTING

Important: When contributing to this repository, please adhere to our code-of-conduct and first discuss the change you wish to make via an issue **before** submitting a pull request.

2.1 Local Development

The following sections will guide you through setting up a local development environment for working on this project package. **At the very least**, make sure that you have the necessary pre-commit hooks installed to make sure that all commits are pristine before they make it into the change history.

2.1.1 Installing Python

Note: If you already have Python 3.7+ installed on your local system, you can skip this step completely.

Installing Python should be done through pyenv. To first install `pyenv` please follow the guide they provided at https://github.com/pyenv/pyenv#installation. When you finally have `pyenv` you should be good to continue on.

```
$ pyenv --version
pyenv x.x.x
```

Now that you have `pyenv` we can install the necessary Python version. This project's package depends on Python 3.7+, so we can request that through `pyenv`.

```
$ pyenv install 3.7  # to install Python 3.7+
...

$ pwd
/PATH/TO/CLONED/REPOSITORY/project-name
$ pyenv local 3.7  # to mark the project directory as needing Python 3.7+
...

$ pyenv global 3.7  # if you wish Python 3.7 to be aliased to `python` everywhere
...
```

After installing and marking the repository as requiring Python 3.7+ you should be good to continue on installing the project's dependencies.

2.1.2 Virtual Environment

We use Poetry to manage both our dependencies and virtual environments. Setting up `poetry` just involves installing it through `pip` as a user-level dependency.

```
$ pip install --user poetry
Collecting poetry
Downloading poetry-x.x.x-py2.py3-non-any.whl
...
```

You can quickly setup your entire development environment by running the installation process from poetry.

```
$ poetry install
Installing dependencies from lock file
...
```

This with create a virtual environment for you and install the necesary development dependencies. From there you can jump into a subshell using the newly created virtual environment using the `shell` subcommand.

```
$ poetry shell
pawning shell within ~/.local/share/virtualenvs/my-project-py3.7
...

$ exit # when you wish to exit the subshell
```

From this shell you have access to all the necessary development dependencies installed in the virutal environment and can start actually writing and running code within the client package.

2.1.3 Style Enforcement

This project's preferred styles are fully enforced through pre-commit hooks. In order to take advantage of these hooks please make sure that you have `pre-commit` and the configured hooks installed in your local environment.

Installing `pre-commit` is done through `pip` and should be installed as a user-level dependency as it adds some console scripts that all projects using `pre-commit` will need.

```
$ pip install --user pre-commit
Collecting pre-commit
Downloading pre_commit-x.x.x-py2.py3-none-any.whl
...

$ pre-commit --version
pre-commit 2.4.0
```

Once `pre-commit` is installed you should also install the hooks into the cloned repository.

```
$ pwd
/PATH/TO/CLONED/REPOSITORY/project-name

$ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

After this you should be good to continue on. These installed hooks will do a first-time setup when you attempt your next commit to build hook environments. Changes that violate the defined style specifications in `setup.cfg` and `pyproject.toml` will cause the commit to fail and will likely make the necessary changes to added / changed files to be written to the failing files.

This will give you the opportunity to view the changes the hooks made to the failing files and add the new changes to the commit in order to make the commit pass. It also gives you the opportunity to make tweaks to the autogenerated changes to make them more human accessible (only if necessary).

2.1.4 Editor Configuration

We also have some specific settings for editor configuration via editorconfig. We recommend you install the appropriate plugin for your editor of choice if your editor doesn't already natively support `.editorconfig` configuration files.

2.1.5 Project Tasking

All of our tasks are built and run through invoke which is basically just a more advanced (a little too advanced) Python alternative to make. The only reason we are using this utility is because I know how it works and I already had most of the necessary tasks defined from other projects.

From within the Poetry subshell, you can access and run these commands through the provided `invoke` development dependency.

```
$ invoke --list
Available tasks:

   build                  Build the project.
   clean                  Clean the project.
   lint                   Lint the project.
   profile                Run and profile a given Python script.
   test                   Test the project.
   docs.build             Build docs.
   docs.build-news        Build towncrier newsfragments.
   docs.clean             Clean built docs.
   docs.view              Build and view docs.
   linter.black           Run Black tool check against source.
   linter.flake8          Run Flake8 tool against source.
   linter.isort           Run ISort tool check against source.
   linter.mypy            Run MyPy tool check against source.
   package.build          Build package source files.
   package.check          Check built package is valid.
   package.clean          Clean previously built package artifacts.
   package.coverage       Build coverage report for test run.
   package.format         Auto format package source files.
   package.requirements   Generate requirements.txt from Poetry's lock.
   package.stub           Generate typing stubs for the package.
   package.test           Run package tests.
   package.typecheck      Run type checking with generated package stubs.
```

You can run these tasks to do many miscellaneous project tasks such as building documentation.

```
$ invoke docs.build
[docs.build] ... building 'html' documentation
Running Sphinx v3.0.3
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
```

(continues on next page)

(continued from previous page)

```
no targets are out of date.
build succeeded.

The HTML pages are in build/html.
```

All of these tasks should just work right out of the box, but something might break eventually after required tooling gets enough major updates.

2.2 Opening Issues

Issues should follow the included `ISSUE_TEMPLATE` found in `.github/ISSUE_TEMPLATE.md`.

- **Issues should contain the following sections:**
  - Expected Behavior
  - Current Behavior
  - Possible Solution
  - Steps to Reproduce (for bugs)
  - Context
  - Your Environment

These sections help the developers greatly by providing a large understanding of the context of the bug or requested feature without having to launch a full fledged discussion inside of the issue.

2.3 Creating Pull Requests

Pull requests should follow the included `PULL_REQUEST_TEMPLATE` found in `.github/PULL_REQUEST_TEMPLATE.md`.

- Pull requests should always be from a `topic` / `feature` / `bugfix` (left side) branch.
  Pull requests from master branches will not be merged.
- Pull requests should not fail our requested style guidelines or linting checks.

CHAPTER

THREE

CHANGELOG

All notable changes to this project will be documented in this file.
The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

3.1 1.0.0 (*2021-01-19*)

No significant changes.

3.2 0.5.0 (*2021-01-07*)

3.2.1 Features

- Automatically casting values applied to *Chalk* to strings. This will fix issues where the user wants to easily use an instance of some class in a templated string without having to cast it to a string themselves.

```
from chalky import fg

class MyObject:

    def __str__(self) -> str:
        return f"{self.__class__.__qualname__!s}()"

print(fg.green | MyObject())
```

3.3 0.4.0 (*2020-12-28*)

3.3.1 Features

- Consuming the current chain's styles and colors if *chalk()* is consumed. This helps with constructing reusable styles with the chaining syntax:

```
from chalky import chain

# previously not possible
error = chain.bold.white.bg.red
```

(continues on next page)

(continued from previous page)

```
success = chain.bold.bright_green

# now possible
error = chain.bold.white.bg.red.chalk
success = chain.bold.bright_green.chalk
```

3.4 0.3.0 (*2020-12-25*)

3.4.1 Features

- Including a global *`configure()`* callable to handle configuring the entire module. At the moment, the only feature we are adding is the ability to completely disable the actual application of styles and colors through the `disable` flag.

3.4.2 Documentation

- Adding some simple chalky chaining documentation.

3.5 0.2.0 (*2020-12-24*)

3.5.1 Features

- Adding an interface for producing styles and colors using chained properties.
  Usage looks like this:

```
from chalky import chain
print(chain.bold.blue | "I'm blue bold text!")
```

3.5.2 Documentation

- Adding some chaining documentation.
- Adding basic usage documentation for the initial release.

3.5.3 Miscellaneous

- Adding a basic Chalky logo to make the documentation a bit more friendly.

3.6 0.1.0 (*2020-12-23*)

3.6.1 Miscellaneous

- Adding the contents of an initial alpha release.

CHAPTER

FOUR

LICENSE

```
ISC License

Copyright (c) 2020, Stephen Bunn <stephen@bunn.io>

Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

CHAPTER

FIVE

PROJECT REFERENCE

5.1 Chalky Package

Simple ANSI terminal text coloring.

Compose multiple of the included chalk instances to produce a style that can be applied directly to strings. Compose multiple chalk instances together with `&` and apply it to a string using `|`:

```
from chalky import sty, fg
print(sty.bold & fg.green | "Hello, World!")
```

5.1.1 Chalk

Contains the base chalk class that is used to group some style and colors.

class chalky.chalk.**Chalk**(*style=<factory>*, *foreground=None*, *background=None*)

Describes the style and color to use for styling some printable text.

Chalk can be composed using `&` and can be applied to strings with `|`. You can create your own instance of chalk by setting your desired `Style` and `Color` when creating a new instance.

Examples

Creating custom instances of chalk looks like the following:

```
>>> my_chalk = Chalk(
...     style={Style.BOLD, Style.UNDERLINE},
...     foreground=Color.RED,
... )
```

Composing two chalk instances can be done through either using `&` or `+`:

```
>>> bold_chalk = Chalk(style={Style.BOLD})
>>> error_chalk = bold_chalk & Chalk(foreground=Color.RED)
```

Using chalk instances to style strings can be done using either `|` or `+`:

```
>>> error_chalk | "Hello, World!"
Hello, World!
```

Important: When composing two chalk instances together, the chalk *being* applied to the base chalk instance will override the foreground and background colors. This means that if you apply a new chalk with a different foreground color it will override the starting foreground color:

```
>>> red = Chalk(foreground=Color.RED)
>>> blue = Chalk(foreground=Color.BLUE)
>>> assert blue == (red & blue)
```

Parameters

- **style** (Set[*Style*]) –
- **foreground** (Union[*Color*, *TrueColor*, None]) –
- **background** (Union[*Color*, *TrueColor*, None]) –

__add__(*other:* chalky.chalk.Chalk) → *chalky.chalk.Chalk*
__add__(*other: str*) → str
Handle applying chalk instances to things.

Parameters **other** (*Union[Chalk, str]*) – Either another chalk instance or a string.

Returns The combined chalk instances or a styled string.

Return type Union[*Chalk*, str]

__and__(*other*)
Create a new chalk instance from the composition of two chalk.

Parameters **other** (*Chalk*) – Another chalk to combine with the current chalk.

Returns The newly created chalk instance.

Return type *Chalk*

__call__(*value*)
Handle applying chalk instances to strings.

Parameters **value** (*str*) – The string to apply the current styles to.

Returns The newly styled string.

Return type str

__or__(*value*)
Style some given string with the current chalk instance.

Tip: If a non-string value is provided, we will attempt to get the most appropriate string from the value by simply calling `str(value)`. So if you are passing in an object, make sure to use an appropriate `__str__` or `__repr__`.

Parameters **value** (*Any*) – The value to apply the current chalk styles to.

Returns The newly styled string.

Return type str

property reverse
Color reverse of the current chalk instance.

Returns The revsered chalk instance

Return type *Chalk*

5.1.2 Style

Contains the available styles we can use for chalk.

class chalky.style.**Style**(*value*)

Enum of the available styles that can be applied to some printable text.

RESET
Resets all styles and colors to the original terminal style.

BOLD
Emphasizes the text by increasing the font weight.

DIM
Dims the text color and sometimes decreases font weight.

ITALIC
Italicize the text.

UNDERLINE
Underlines the text (works in most modern terminals).

SLOW_BLINK
Flash the text slowly (doesn't work in most modern terminals).

RAPID_BLINK
Flash the text very quickly (doesn't work in most modern terminals).

REVERSED
Reversed the current style of the terminal for the text.

CONCEAL
Hide the text.

STRIKETHROUGH
Draw a line through the text (works in most modern terminals).

NORMAL
Normalizes the text for the current terminal.

5.1.3 Color

Contains the available tools to define the colors that can be used for chalk.

class chalky.color.**Color**(*value*)

Enum of the available colors that can be used to color some printable text.

BLACK

RED

GREEN

YELLOW

BLUE

MAGENTA

CYAN

WHITE

BRIGHT_BLACK
Otherwise known as gray.

BRIGHT_RED

BRIGHT_GREEN

BRIGHT_YELLOW

BRIGHT_BLUE

BRIGHT_MAGENTA

BRIGHT_CYAN

BRIGHT_WHITE
Otherwise known as actual white.

class chalky.color.**TrueColor**(*red*, *green*, *blue*)
Describes a true color that can be displayed on compatible terminals.

Parameters

- **red** (*int*) – The value of red (0-255).
- **green** (*int*) – The value of green (0-255).
- **blue** (*int*) – The value of blue (0-255).

__hash__()
Generate a comparable hash for the current instance.

Returns The appropriate hash of the current instance.

Return type int

classmethod from_hex(*color*)
Create an instance from a given hex color string.

Parameters **color** (*str*) – The hex color string of the color to create.

Raises **ValueError** – If the given hex color string is not a length of 3 or 6

Returns The created instance.

Return type *TrueColor*

to_bytes()
Convert the current color to a tuple of RGB bytes.

Returns The corresponding bytes of the current instance (red, green, blue).

Return type Tuple[bytes, bytes, bytes]

5.1.4 Chain

Contains the chain class that can be used to quickly produce styles and colors.

class chalky.chain.**Chain**(*_chalk=<factory>*, *_background=False*)
Quickly produce a chain of styles and colors that can be applied to a string.

Parameters **chalk** (Chalk) – The container chalk instance that contains the current chain style.

Examples

Chaining styles together should be fairly straightforward:

```
>>> from chalk import chain
>>> print(chain.bold.blue | "Bold blue text")
>>> print(chain.black.bg.green.italic | "Italic black text on green background")
```

Once a *Chain* instance is applied to a string, the styles are consumed and the chain instance is reset.

Parameters

- **_chalk** (*Chalk*) –
- **_background** (bool) –

__add__(*other: Union[*chalky.chalk.Chalk, chalky.chain.Chain*]*) → *chalky.chain.Chain*
__add__(*other: str*) → str
Handle applying chain instances to things.

Parameters **other** (*Union[Chalk, Chain, str]*) – Either a chalk instance, a chain instance, or a string.

Returns The updated chain or the applied string.

Return type Union[*Chain*, str]

__and__(*other*)
Compose the chain with another chain or a chalk instance.

Parameters **other** (*Union[Chalk, Chain]*) – Another Chain or *Chalk* instance to compose with the current chain.

Returns The newly updated chain.

Return type Chain

__call__(*value*)
Handle applying a chain to a strings.

Parameters **value** (*str*) – The string to apply the current chain to.

Returns The newly styled string.

Return type str

__or__(*value*)
Style some given string with the current chain.

Parameters **value** (*str*) – THe string to apply the current chain to.

Returns The newly styled string.

Return type str

property bg
Following colors will be applied as the background color.

Return type *Chain*

property chalk
Extract the currently built chalk instance.

Important: Consuming this property will reset the current chain's styles. We are assuming that if you need the *Chalk*, you have finished constructing it through the chaining syntax.

Returns The current chalk instance from the chained styles and colors.

Return type Chalk

property fg
Following colors will be applied as the foreground color.

Return type *Chain*

hex(*color*)
Add a truecolor chalk from a hex string.

Parameters **color** (*str*) – The hex color string (#ffffff)

Returns The newly updated chain.

Return type *Chain*

rgb(*red*, *green*, *blue*)
Add a truecolor chalk from an RGB tuple.

Parameters

- **red** (*int*) – The intensity of red (0-255).
- **green** (*int*) – The intensity of green (0-255).
- **blue** (*int*) – The intensity of blue (0-255).

Returns The newly updated chain.

Return type *Chain*

5.1.5 Shortcuts

Contains shortcuts for quickly utilizing chalk.

Simply combine colors and styles using `&` or `+` to produce a style that can used to format a string with `|` or `+`.

Examples

```
>>> from chalky import bg, fg, sty
>>> my_style = bg.red & fg.black & sty.bold
>>> print(my_style | "I'm bold black on red text")
I'm bold black on red text
```

Many chalk styles can be combined together:

Examples

```
>>> from chalky import sty
>>> my_style = sty.bold & sty.underline
>>> print(my_style | "I'm bold and underlined")
I'm bold and underlined
```

The last applied foreground or background color will be used when applied to a string:

Examples

```
>>> from chalky import bg
>>> my_style = bg.red & bg.blue  # BLUE will override RED when styling the string
>>> print(my_style | "My background is BLUE")
My background is BLUE
```

chalky.shortcuts.**hex**(*hexcolor*, *background=False*)

Generate a new truecolor chalk from a HEX color (`#ffffff`) string.

Parameters

- **hexcolor** (*str*) – The hex color string.
- **background** (*bool, optional*) – If `True` will generate the new chalk to be applied as background color. Defaults to False.

Returns The new chalk instance.

Return type *Chalk*

chalky.shortcuts.**rgb**(*red*, *green*, *blue*, *background=False*)

Generate a new truecolor chalk from an RGB tuple.

Parameters

- **red** (*int*) – The intensity of red (0-255).
- **green** (*int*) – The intensity of green (0-255).
- **blue** (*int*) – The intensity of blue (0-255).
- **background** (*bool, optional*) – If `True` will generate the new chalk to be applied as a background color. Defaults to False.

Returns The new chalk instance.

Return type *Chalk*

5.1.6 Interface

Contains the actual implementation of interacting with a type of terminal buffer.

chalky.interface.**get_interface**(*io=None*)
Get the appropriate interface to interact with some terminal buffer.

Examples

Get the default interface for interacting with the terminal buffer. This defaults to using `sys.stdout`

```
>>> from chalky.interface import get_interface
>>> interface = get_interface()
```

To get an interface using a different text io buffer, pass it in:

```
>>> import sys
>>> stderr_interface = get_interface(sys.stderr)
```

Parameters **io** (Optional[`TextIO`], optional) – The io to build an interface for. Defaults to `sys.stdout`.

Returns The created interface for the given io buffer.

Return type `BaseInterface`

5.1.7 Constants

Contains package constants that modify the global functions of the package.

Utilize the *`configure()`* method to quickly and easily disable all future application of *`Chalk`* to strings.

```
>>> from chalky import configure, fg
>>> configure(disable=True)
>>> print(fg.green | "I'm NOT green text")
```

chalky.constants.**configure**(*disable=False*)
Configure the global state of the chalky module.

Parameters **disable** (*`bool, optional`*) – If True, will disable all future application of colors and styles. Defaults to False.

chalky.constants.**is_disabled**()
Callable to evaluate the disabled conditional.

Returns True if chalky is disabled, otherwise False.

Return type bool

5.1.8 Helpers

Contains some miscellaneous helpers for the rest of the package.

chalky.helpers.**int_to_bytes**(*value*)
Convert a given number to its representation in bytes.

Parameters **value** (*int*) – The value to convert.

Returns The representation of the given number in bytes.

Return type bytes

chalky.helpers.**supports_posix**()
Check if the current machine supports basic posix.

Returns True if the current machine is MacOSX or Linux.

Return type bool

chalky.helpers.**supports_truecolor**()
Attempt to check if the current terminal supports truecolor.

Returns True if the current terminal supports truecolor, otherwise False.

Return type bool

PYTHON MODULE INDEX

C

INDEX

Symbols

B

C

D

F

G

H

I

M

N

R

S

T

U

W

Y