chromalog Documentation

Release 1.0.5

Julien Kauffmann

May 25, 2016

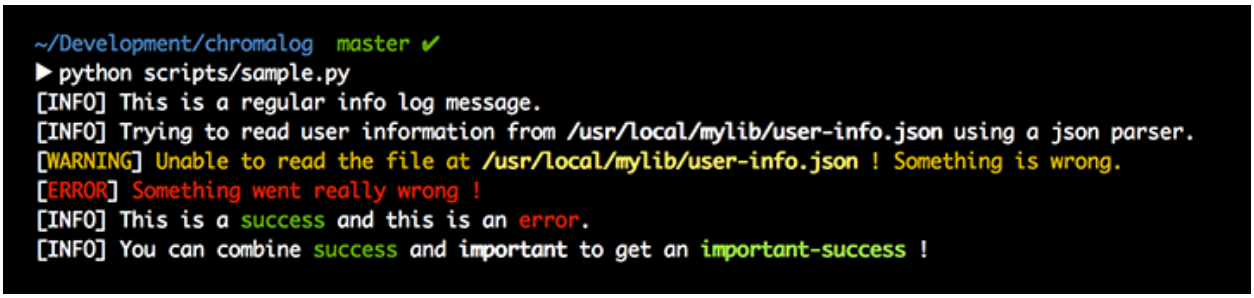
Contents

Chromalog is a Python library that eases the use of colors in Python logging.

It integrates seamlessly into any Python 2 or Python 3 project. Based on colorama, it works on both Windows and *NIX platforms and is highly configurable.

Chromalog can detect whether the associated output stream is color-capable and even has a fallback mechanism: if color is not supported, your log will look no worse than it was before you colorized it.

Using **Chromalog**, getting a logging-system that looks like this is a breeze:



Its use is simple and straightforward:

```
from chromalog.mark.helpers.simple import important

logger.info("Connected as %s for 2 hours.", important(username))
```

Ready to add some colors in your life ? *Get started* or check out *Chromalog's API* !

CHAPTER 1

Table of contents

1.1 Installation

1.1.1 Using pip

The simplest way to install **Chromalog** is to use pip.

Just type the following command in your command prompt:

```
pip install chromalog
```

That's it ! No configuration is needed. **Chromalog** is now installed on your system.

1.1.2 From source

If you feel in hacky mood, you can also install **Chromalog** from source.

Clone the Git repository:

```
git clone git@github.com:freelan-developers/chromalog.git
```

Then, inside the cloned repository folder:

```
python setup.py install
```

And that's it ! **Chromalog** should now be installed in your Python packages.

You can easily test it by typing in a command prompt:

```
python -c "import chromalog"
```

This should not raise any error (especially not an `ImportError`).

1.1.3 What's next ?

Get started or explore *Chromalog's API*.

1.2 Quickstart

If you haven't installed **Chromalog** yet, it is highly recommended that *you do so* before reading any further.

1.2.1 How it works

Chromalog provides colored logging through the use of custom *StreamHandler* and *Formatter*.

The *ColorizingStreamHandler* is responsible for writing the log entries to the output stream. It can detect whether the associated stream has color capabilities and eventually fallback to a non-colored output mechanism. In this case it behaves exactly like a standard `logging.StreamHandler`. It is associated to a *color map* that is passed to every formatter that requests it.

The *ColorizingFormatter* is responsible for adding the color-specific markup in the formatted string. If used with a non colorizing stream handler, the *ColorizingFormatter* will transparently fallback to a non-colorizing behavior.

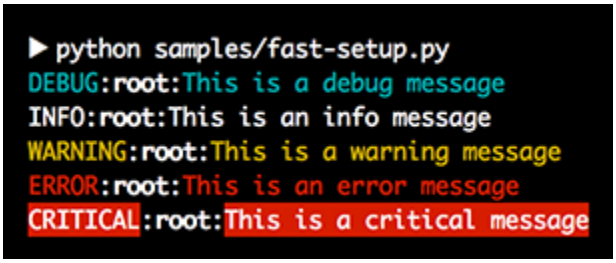
1.2.2 Fast setup

Chromalog provides a *basicConfig* function, very similar to `logging.basicConfig()` that quickly sets up the root logger, but using a *ColorizingStreamHandler* and a *ColorizingFormatter* instead.

It can be used like so to setup logging in a Python project:

```
import logging
import chromalog

chromalog.basicConfig(level=logging.DEBUG)
logger = logging.getLogger()

logger.debug("This is a debug message")
logger.info("This is an info message")
logger.warning("This is a warning message")
logger.error("This is an error message")
logger.critical("This is a critical message")
```

Which produces the following output:



It's as simple as it gets !

1.2.3 Marking log objects

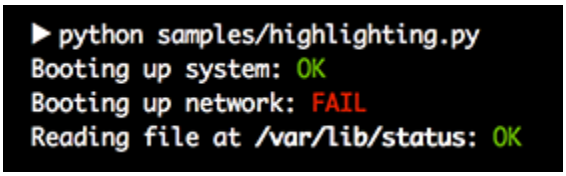
While **Chromalog** has the ability to color entire log lines, it can also mark some specific log elements to highlight them in the output.

A good example of that could be:

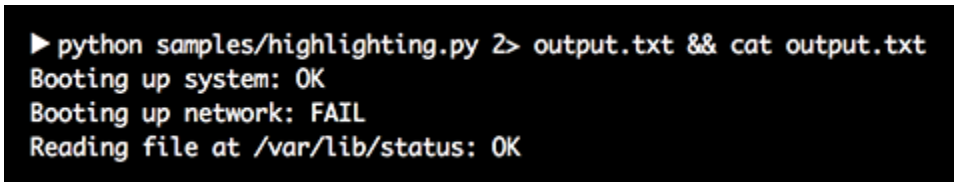
```
import logging
import chromalog

from chromalog.mark.helpers.simple import success, error, important

```

```
chromalog.basicConfig(format="%(message)s", level=logging.INFO)
logger = logging.getLogger()

filename = r'/var/lib/status'

logger.info("Booting up system: %s", success("OK"))
logger.info("Booting up network: %s", error("FAIL"))
logger.info("Reading file at %s: %s", important(filename), success("OK"))
```

Which produces the following output:



Note what happens when we redirect the output to a file:



As you can see, **Chromalog** automatically detected that the output stream wasn't color-capable and disabled automatically the colorizing. Awesome !

Checkout *Marking functions* for the complete list of available marking functions.

1.2.4 What's next ?

Want to learn more about **Chromalog** ? Go read *Advanced usage* !

1.3 Advanced usage

We've seen in *Quickstart* how to quickly colorize your logging output. But **Chromalog** has much more to offer than just that !

1.3.1 Marking functions

The *chromalog.mark* module contains all **Chromalog**'s marking logic.

Its main component is the `Mark` class which wraps any Python object and associates it with one or several *color tags*.

Those color tags are evaluated during the formatting phase by the *ColorizingFormatter* and transformed into color sequences, as defined in the *ColorizingStreamHandler*'s *color map*.

To decorate a Python object, one can just do:

```
from chromalog.mark import Mark

marked_value = Mark('value', 'my_color_tag')
```

You may define several color tags at once, by specifying a list:

```
from chromalog.mark import Mark

marked_value = Mark('value', ['my_color_tag', 'some_other_tag'])
```

Nested `Mark` instances are actually flattened automatically and their color tags appended.

```
from chromalog.mark import Mark

marked_value = Mark(Mark('value', 'some_other_tag'), 'my_color_tag')
```

> **Warning:** Be careful when specifying several color tags: their order **matters** !
> Depending on the color sequences of your color map, the formatted result might differ.
> See *Color maps* for an example.

Helpers

Chromalog also comes with several built-in helpers which make marking objects even more readable. Those helpers are generated automatically by several *magic* modules.

Simple helpers

Simple helpers are a quick way of marking an object and an explicit way of highlighting a value.

You can generate simple helpers by importing them from the *chromalog.mark.helpers.simple* magic module, like so:

```
from chromalog.mark.helpers.simple import important

print(important(42).color_tag)
```

Which gives the following output:

```
['important']
```

An helper function with a color tag similar to its name will be generated and made accessible transparently.

Like `Mark` instances, you can obviously combine several helpers to cumulate the effects.

For instance:

```
from chromalog.mark.helpers.simple import important, success

print(important(success(42)).color_tag)
```

Gives:

```
['important', 'success']
```

If the name of the helper you want to generate is not a suitable python identifier, you can use the `chromalog.mark.helpers.simple.make_helper()` function instead.

Note that, should you need it, documentation is generated for each helper. For instance, here is the generated documentation for the *chromalog.mark.helpers.simple.success()* function:

chromalog.mark.helpers.simple.**success**(*obj*)

Mark an object for coloration.

The color tag is set to 'success'.

Parameters **obj** – The object to mark for coloration.

Returns A *Mark* instance.

```
>>> from chromalog.mark.helpers.simple import success
```

```
>>> success(42).color_tag
['success']
```

Conditional helpers

Conditional helpers are a quick way of associating a color tag to an object depending on a boolean condition.

You can generate conditional helpers by importing them from the *chromalog.mark.helpers.conditional* magic module:

```
from chromalog.mark.helpers.conditional import success_or_error

print(success_or_error(42, True).color_tag)
print(success_or_error(42, False).color_tag)
print(success_or_error(42).color_tag)
print(success_or_error(0).color_tag)
```

Which gives:

```
['success']
['error']
['success']
['error']
```

Warning: Automatically generated conditional helpers must have a name of the form `a_or_b` where `a` and `b` are color tags.

If the name of the helper you want to generate is not a suitable python identifier, you can use the `chromalog.mark.helpers.conditional.make_helper()` function instead.

Note: If no `condition` is specified, then the value itself is evaluated as a boolean value.

This is useful for outputing exit codes for instance.

1.3.2 Colorizers

The *GenericColorizer* class is responsible for turning color tags into colors (or decoration sequences).

Color maps

To do so, each *GenericColorizer* instance has a `color_map` `dictionary` which has the following structure:

```
color_map = {
    'alpha': ('[', ']'),
    'beta': ('{', '}'),
}
```

That is, each *key* is the color tag, and each *value* is a pair `(start_sequence, stop_sequence)` of start and stop sequences that will surround the decorated value when it is output.

Values are decorated in order with the seqauences that match their associated color tags. For instance:

```
from chromalog.mark.helpers.simple import alpha, beta
from chromalog.colorizer import GenericColorizer

colorizer = GenericColorizer(color_map={
    'alpha': ('[', ']'),
    'beta': ('{', '}'),
})

print(colorizer.colorize(alpha(beta(42))))
print(colorizer.colorize(beta(alpha(42))))
```

Which gives:

```
[{42}]
{[42]}
```

Context colorizing

Note that the *colorize* method takes an optional parameter `context_color_tag` which is mainly used by the *ColorizingFormatter* to colorize subparts of a colorized message.

`context_color_tag` should match the color tag used to colorize the contextual message as a whole. Unless you write your own formatter, you shouldn't have to care much about it.

Here is an example on how `context_color_tag` modifies the output:

```
from chromalog.mark.helpers.simple import alpha
from chromalog.colorizer import GenericColorizer

colorizer = GenericColorizer(color_map={
    'alpha': ('[', ']'),
    'beta': ('{', '}'),
})

print(colorizer.colorize(alpha(42), context_color_tag='beta'))
```

Which gives:

```
}{[42]}{
```

As you can see, the context color tag is first closed then reopened, then the usual color tags are used. This behavior is required as it prevents some color escaping sequences to persist after the tags get closed on some terminals.

Built-in colorizers

Chromalog ships with two default colorizers:

- *Colorizer* which is associated to a color map constitued of color escaping sequences.

- *MonochromaticColorizer* which may be used on non color-capable output streams and that only decorates objects marked with the `'important'` color tag.

See *Default color maps and sequences* for a comprehensive list of default color tags and their resulting sequences.

Custom colorizers

One can create its own colorizer by simply deriving from the *GenericColorizer* class and defining the `default_color_map` class attribute, like so:

```
from chromalog.colorizer import GenericColorizer

from colorama import (
   Fore,
   Back,
   Style,
)

class MyColorizer(GenericColorizer):
   default_color_map = {
      'success': (Fore.GREEN, Style.RESET_ALL),
   }
```

Decorating messages

Colorizers also provide a method to directly colorize a message, regardless of any output stream and its color capabilities:

GenericColorizer.**colorize_message**(*message*, **args*, ***kwargs*)

Colorize a message.

Parameters **message** – The message to colorize. If message is a marked object, its color tag will be used as a `context_color_tag`. `message` may contain formatting placeholders as described in `str.format()`.

Returns The colorized message.

> **Warning:** This function has no way of check the color-capability of any stream that the resulting string might be printed to.

Here is an example of usage:

```
from chromalog.colorizer import GenericColorizer
from chromalog.mark.helpers.simple import alpha

colorizer = GenericColorizer(color_map={
    'alpha': ('[', ']'),
})

print(colorizer.colorize_message(
    'hello {0} ! How {are} you ?',
    alpha('world'),
    are=alpha('are'),
))
```

This gives the following output:

```
hello [world] ! How [are] you ?
```

Default color maps and sequences

Here is a list of the default color tags and their associated sequences:

Colorizer	Color tag	Effect
Colorizer	*debug*	Light blue color.
	info	Default terminal style.
	important	Brighter output.
	success	Green color.
	warning	Yellow color.
	error	Red color.
	critical	Red background.
MonochromaticColorizer	*important*	Value surrounded by `**`.

1.4 Chromalog's API

Here is a comprehensive list of all modules, classes and function provided by **Chromalog**.

1.4.1 chromalog

Enhance Python logging with colors.

chromalog.**basicConfig**(*format=None*, *datefmt=None*, *level=None*, *stream=None*, *colorizer=None*)

Does basic configuration for the logging system by creating a *chromalog.log.ColorizingStreamHandler* with a default *chromalog.log.ColorizingFormatter* and adding it to the root logger.

This function does nothing if the root logger already has handlers configured for it.

Parameters

- **format** – The format to be passed to the formatter.
- **datefmt** – The date format to be passed to the formatter.
- **level** – Set the root logger to the specified level.
- **stream** – Use the specified stream to initialize the stream handler.
- **colorizer** – Set the colorizer to be passed to the stream handler.

1.4.2 chromalog.log

Log-related functions and structures.

class chromalog.log.**ColorizingFormatter**(*fmt=None*, *datefmt=None*)

A formatter that colorize its output.

Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional datefmt argument (if omitted, you get the ISO8601 format).

format(*record*)
Colorize the arguments of a record.

Record A *LogRecord* instance.

Returns The colorized formatted string.

Note: The *record* object must have a *colorizer* attribute to be use for colorizing the formatted string. If no such attribute is found, the default non-colorized behaviour is used instead.

---

class chromalog.log.**ColorizingStreamHandler**(*stream=None*, *colorizer=None*, *highlighter=None*, *attributes_map=None*)
A stream handler that colorize its output.

Initializes a colorizing stream handler.

Parameters

- **stream** – The stream to use for output.
- **colorizer** – The colorizer to use for colorizing the output. If not specified, a *chromalog.colorizer.Colorizer* is instantiated.
- **highlighter** – The colorizer to use for highlighting the output when color is not supported.
- **attributes_map** – A map of LogRecord attributes/color tags.

active_colorizer
The active colorizer or highlighter depending on whether color is supported.

format(*record*)
Format a *LogRecord* and prints it to the associated stream.

1.4.3 chromalog.colorizer

Colorizing functions and structures.

class chromalog.colorizer.**ColorizableMixin**(*color_tag=None*)
Make an object colorizable by a colorizer.

Initialize a colorizable instance.

Parameters **color_tag** – The color tag to associate to this instance.

color_tag can be either a string or a list of strings.

class chromalog.colorizer.**ColorizedObject**(*obj*, *color_pair=None*)
Wraps any object to colorize it.

Initialize the colorized object.

Parameters

- **obj** – The object to colorize.
- **color_pair** – The (start, stop) pair of color sequences to wrap that object in during string rendering.

class chromalog.colorizer.**Colorizer**(*color_map=None*, *default_color_tag=None*)
Colorize log entries.

Initialize a new colorizer with a specified *color_map*.

Parameters

- **color_map** – A dictionary where the keys are color tags and the value are couples of color sequences (start, stop).
- **default_color_tag** – The color tag to default to in case an unknown color tag is encountered. If set to a falsy value no default is used.

class chromalog.colorizer.**GenericColorizer**(*color_map=None*, *default_color_tag=None*)

A class reponsible for colorizing log entries and `chromalog.important.Important` objects.

Initialize a new colorizer with a specified *color_map*.

Parameters

- **color_map** – A dictionary where the keys are color tags and the value are couples of color sequences (start, stop).
- **default_color_tag** – The color tag to default to in case an unknown color tag is encountered. If set to a falsy value no default is used.

colorize(*obj*, *color_tag=None*, *context_color_tag=None*)

Colorize an object.

Parameters

- **obj** – The object to colorize.
- **color_tag** – The color tag to use as a default if `obj` is not marked.
- **context_color_tag** – The color tag to use as context.

Returns `obj` if `obj` is not a colorizable object. A colorized string otherwise.

colorize_message(*message*, **args*, ***kwargs*)

Colorize a message.

Parameters **message** – The message to colorize. If message is a marked object, its color tag will be used as a `context_color_tag`. `message` may contain formatting placeholders as described in `str.format()`.

Returns The colorized message.

> **Warning:** This function has no way of check the color-capability of any stream that the resulting string might be printed to.

get_color_pair(*color_tag*, *context_color_tag=None*, *use_default=True*)

Get the color pairs for the specified *color_tag* and *context_color_tag*.

Parameters

- **color_tag** – A list of color tags.
- **context_color_tag** – A list of color tags to use as a context.
- **use_default** – If `False` then the default value won't be used in case the `color_tag` is not found in the associated color map.

Returns A pair of color sequences.

class chromalog.colorizer.**MonochromaticColorizer**(*color_map=None*, *default_color_tag=None*)

Monochromatic colorizer for non-color-capable streams that only highlights `chromalog.mark.Mark` objects with an `important` color tag.

Initialize a new colorizer with a specified *color_map*.

Parameters

- **color_map** – A dictionary where the keys are color tags and the value are couples of color sequences (start, stop).
- **default_color_tag** – The color tag to default to in case an unknown color tag is encountered. If set to a falsy value no default is used.

1.4.4 chromalog.mark

Marking classes and methods.

1.4.5 chromalog.mark.objects

Mark log entries.

class chromalog.mark.objects.**Mark**(*obj*, *color_tag*)
Wraps any object and mark it for colored output.

Mark `obj` for coloration.

Parameters

- **obj** – The object to mark for colored output.
- **color_tag** – The color tag to use for coloring. Can be either a list of a string. If `color_tag` is a string it will be converted into a single-element list automatically.

Note: Nested `chromalog.mark.Mark` objects are flattened automatically and their `color_tag` are appended.

```
>>> from chromalog.mark.objects import Mark
```

```
>>> Mark(42, 'a').color_tag
['a']
```

```
>>> Mark(42, ['a']).color_tag
['a']
```

```
>>> Mark(42, ['a', 'b']).color_tag
['a', 'b']
```

```
>>> Mark(Mark(42, 'c'), ['a', 'b']) == Mark(42, ['a', 'b', 'c'])
True
```

1.4.6 chromalog.mark.helpers

Automatically generate marking helpers functions.

class chromalog.mark.helpers.**ConditionalHelpers**
A class that is designed to act as a module and implement magic helper generation.

make_helper(*color_tag_true*, *color_tag_false*)
Make a conditional helper.

Parameters

- **color_tag_true** – The color tag if the condition is met.
- **color_tag_false** – The color tag if the condition is not met.

Returns The helper function.

class chromalog.mark.helpers.**SimpleHelpers**
A class that is designed to act as a module and implement magic helper generation.

make_helper(*color_tag*)
Make a simple helper.

Parameters **color_tag** – The color tag to make a helper for.

Returns The helper function.

1.4.7 chromalog.mark.helpers.simple

Pseudo-module that generates simple helpers.

See *SimpleHelpers*.

1.4.8 chromalog.mark.helpers.conditional

Pseudo-module that generates conditional helpers.

See *ConditionalHelpers*.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

C

A

B

C

F

G

M

S