pytermor

Release 2.103.0

Alexandr Shavykin

Sep 24, 2023

INTRODUCTION

(yet another) Python library initially designed for formatting terminal output using ANSI escape codes.

Provides high-level <guide-hi> methods for working with text sections, colors, formats, alignment and wrapping, as well as low-level <guide-lo> modules which allow to operate with *ANSI* sequences directly and also implement automatic format termination. Depending on the context and technical requirements either approach can be used. Also includes a set of additional number/string/time formatters for pretty output, filters, templating engine, escape sequence parser and provides support for several color spaces, which is also used for fluent color approximation if terminal capabilities do not allow to work in True Color mode. See features page for the details.

The library is extendable and supports a variety of formatters (called renderers<guide.renderers>), which determine the output syntax:

- SgrRenderer, global default; formats the text with ANSI escape sequences for ttys;
- TmuxRenderer, suitable for integration with tmux (terminal multiplexer);
- HtmlRenderer, which makes a HTML page with all the formatting composed by CSS styles;
- SgrDebugger, same as `SgrRenderer`, but ESC (0x1B) bytes are replaced with a regular letter, therefore all the sequences are no longer sequences and can be seen as a text, for SGR debugging;
- etc.

No dependencies required, only Python Standard Library (although there are some for testing and docs building).

Contents

1

INSTALLATION

Python 3.8 or later should be installed and available in `$PATH`; that's basically it if intended usage of the package is as a library.

Listing 1: Installing into a project

```
$ python -m pip install pytermor
```

Listing 2: Standalone installation (for developing or experimenting)

```
$ git clone git@github.com:delameter/pytermor.git .
$ python -m venv venv
$ PYTHONPATH=. venv/bin/python -m pytermor
v2.41.1-dev1:Feb-23
```

2

FEATURES

2.1 Flexible input formats

guide.fargs allows to compose formatted text parts much faster and keeps the code compact:

```
import pytermor as pt

ex_st = pt.Style(bg='#ffff00', fg='black')
text = pt.FrozenText(
    'This is red ', pt.cv.RED,
    "This is white ",
    "This is black on yellow", ex_st,
)
pt.echo(text)
```

2.2 Content-aware format nesting

r Template tags and non-closing Fragments <Fragment> allow to build complex formats.

```
import pytermor as pt

s = """:[fg=red]nested for:[bold]mat:[bg=blue]ting a:[fg=yellow]nd :[-]over:[-]laps"""
pt.echo(pt.TemplateEngine().substitute(s))
```

2.3 256 colors / True Color support

The library supports extended color modes:

- XTerm 256 colors indexed mode
- True Color RGB mode (16M colors)

```
import pytermor as pt

for outm in ['xterm_16', 'xterm_256', 'true_color']:
    print(' '+outm.ljust(12), end="")
    for c in range((W := 80) + 1):
        b = pt.RGB.from_ratios(1 - (p := c / W), 2 * min(p, 1 - p), p).int
        f = pt.Fragment(" ·"[c & 1], pt.Style(fg=(1 << 24) - b, bg=b, bold=True))
        print(f.render(pt.SgrRenderer(outm)), end=["", 2*"\n"][c >= W], flush=True)
```

2.4 Different color spaces

Currently supported spaces: RGB, HSV, XYZ, LAB. Any of these can be transparently translated to any other.

```
import pytermor as pt

col = pt.RGB(0xDA9AC4)
for v in [col.rgb, col.hsv, col.xyz, col.lab]:
    print(repr(v))
```

2.5 Named colors collection

Registry containing more than 2400 named colors, in addition to default 256 from `xterm` palette.

2.6 Extendable renderers

Renderers <guide.renderers> is a family of classes responsible for creating formatted strings from IRenderable instances, which, in general, consist of a text piece and a *Style* – a set of formatting rules. Concrete implementation of the renderer determines the target format and/or platform.

This is how SgrRenderer, HtmlRenderer, TmuxRenderer, SgrDebugger (from top to bottom) output can be seen in a terminal emulator:

2.7 Number formatters

Set of highly customizable helpers, see numfmt.

format_si() output sample:

format_time_ns() output samples:

format_time_delta() output sample:

2.8 Data dumps

Special formatters for raw binary/string data.

These examples were composed for a terminal 80-chars wide; tracers dynamically change the amount of elements per line at each dump() call.

Input data for all examples below was the same.

Listing 1: Decomposition into separate bytes by BytesTracer. Note the hexadecimal offset format.

```
0x00 | 3D 90 39 05  B9 54 BA 89  90 A8 86 4C  A3 99 75 DD  BC 02 0D 0A
0x14 | 7A E8 E6 40  76 4B 36 1C  00 AD 02 E2  61 45 FD 92  CD B6 71 02
0x28 | 4F 52 EC 39  64 22 68 6A  2E 4E 80 1E  67 07 31 0D  83 55 4D F2
0x3C | D0 D5 D9 41  72 54 6D 2B  03 80 FE 95  B3 28 C4 3E  FC BC 4E 30
0x50 | 5C 6B 5C C3  99 B3 A4 93  24 E9 43 E9  30 B8 6A BC  74 F9 EA 4A
0x64 | 30 4F 9A 38  71 DF B2 39  19 30 56 7C  73 91 56 6E  B8 38 48 F5
0x78 | B7 5B 08 BD  96 B5 4F 6E
-------------------------------------------------------------------(0x80)
```

Listing 2: Decomposition into UTF-8 sequences by StringTracer

```
  0 |     3d efbfbd     39     05 efbfbd     54 efbfbd efbfbd |=9·T
  8 | efbfbd efbfbd efbfbd     4c efbfbd efbfbd     75   ddbc |Lu
 16 |     02     0d     0a     7a efbfbd efbfbd     40     76 |·z@v
 24 |     4b     36     1c     00 efbfbd     02 efbfbd     61 |K6···a
 32 |     45 efbfbd efbfbd   cdb6     71     02     4f     52 |Eq·OR
 40 | efbfbd     39     64     22     68     6a     2e     4e |9d"hj.N
 48 | efbfbd     1e     67     07     31     0d efbfbd     55 |·g·1U
 56 |     4d efbfbd efbfbd efbfbd efbfbd     41     72     54 |MArT
 64 |     6d     2b     03 efbfbd efbfbd efbfbd efbfbd     28 |m+·(
 72 | efbfbd     3e efbfbd efbfbd     4e     30     5c     6b |>N0\k
 80 |     5c   c399 efbfbd efbfbd efbfbd     24 efbfbd     43 |\Ù$C
 88 | efbfbd     30 efbfbd     6a efbfbd     74 efbfbd efbfbd |0jt
 96 |     4a     30     4f efbfbd     38     71   dfb2     39 |J0O8q9
104 |     19     30     56     7c     73 efbfbd     56     6e |·0V|sVn
112 | efbfbd     38     48 efbfbd efbfbd     5b     08 efbfbd |8H[·
120 | efbfbd efbfbd     4f     6e                             |On
-------------------------------------------------------------------(124)
```

Listing 3: Decomposition into Unicode codepoints by StringUcpTracer

```
  0 |U+   3D FFFD   39   05 FFFD   54 FFFD FFFD FFFD FFFD FFFD |=9·T
 11 |U+   4C FFFD FFFD   75  77C   02   0D   0A   7A FFFD FFFD |Lu·z
 22 |U+   40   76   4B   36   1C   00 FFFD   02 FFFD   61   45 |@vK6···aE
 33 |U+ FFFD FFFD  376   71   02   4F   52 FFFD   39   64   22 |q·OR9d"
 44 |U+   68   6A   2E   4E FFFD   1E   67   07   31   0D FFFD |hj.N·g·1
 55 |U+   55   4D FFFD FFFD FFFD FFFD   41   72   54   6D   2B |UMArTm+
 66 |U+   03 FFFD FFFD FFFD FFFD   28 FFFD   3E FFFD FFFD   4E |·(>N
 77 |U+   30   5C   6B   5C   D9 FFFD FFFD FFFD   24 FFFD   43 |0\k\Ù$C
 88 |U+ FFFD   30 FFFD   6A FFFD   74 FFFD FFFD   4A   30   4F |0jtJ0O
 99 |U+ FFFD   38   71  7F2   39   19   30   56   7C   73 FFFD |8q9·0V|s
110 |U+   56   6E FFFD   38   48 FFFD FFFD   5B   08 FFFD FFFD |Vn8H[·
121 |U+ FFFD   4F   6E                                         |On
-------------------------------------------------------------------------(124)
```

3

LIBRARY STRUCTURE

ansi	Classes for working with ANSI escape sequences on a lower level.
color	Abstractions for color definitions in three primary modes: 4-bit, 8-bit and 24-bit (`xterm-16`, `xterm-256` and `True Color/RGB`, respectively).
common	
config	Library fine tuning module.
cval	Color preset list:
exception	
filter	Formatters for prettier output and utility classes to avoid writing boilerplate code when dealing with escape sequences.
log	
numfmt	utilnum
renderer	Renderers transform *Style* instances into lower-level abstractions like *SGR sequences*, tmux-compatible directives, HTML markup etc., depending on renderer type.
style	Reusable data classes that control the appearance of the output -- colors (text/background/underline) and attributes (*bold*, *underlined*, *italic*, etc.).
template	
term	A
text	"Front-end" module of the library.

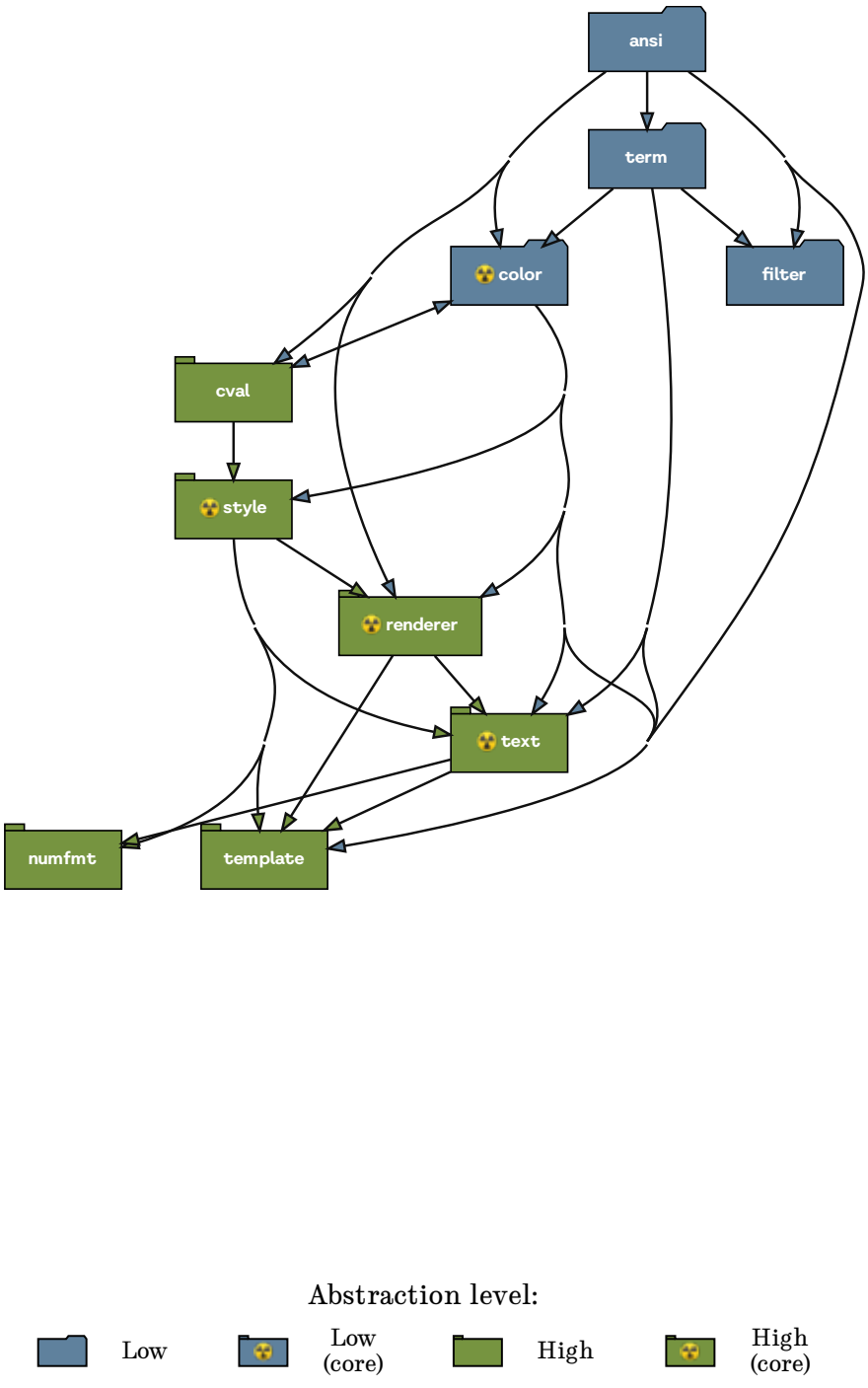


Fig. 1: Module dependency graph[Page 9, 2]

[2] Overly common modules (`exception`, `log`, `config` and `common` itself) are not shown, as they turn the graph into a mess. Same applies to internal modules which name starts with _.

4

EXAMPLES

The library can be split into two domains, the first one being "**high**-level" domain, which includes templating, style abstractions, text implementations which support aligning, wrapping, padding, etc., as well as number formatting helpers and a registry of preset colors.

The second one is "**low**-level", containing colors and color spaces definitions, helpers for composing various terminal escape sequences, the escape sequence abstractions themselves, as well as a large set of filters for chain-like application.

4.1 Rendering · High-level

Imagine we want to colorize `git --help` output *manually*, i.e., we will not pipe an output of `git` and apply filters to do the job (yet), instead we copy-paste the output to python source code files as string literals and will try to add a formatting using all primary approaches.

Listing 1: Part of the input

```
These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone             Clone a repository into a new directory
   init              Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add               Add file contents to the index
   mv                Move or rename a file, a directory, or a symlink
   restore           Restore working tree files
   rm                Remove files from the working tree and from the index
   sparse-checkout   Initialize and modify the sparse-checkout
   ...
```

Part of the output:

The examples in this part are sorted from simple ones at the beginning to complicated ones at the end.

4.1.1 Separated pre-rendering

Use render() method to apply a *style* to a string part individually for each of them.

```
import pytermor as pt

subtitle = pt.render("start a working area", pt.Style(fg=pt.cv.YELLOW, bold=True))
subtitle += " (see also: "
subtitle += pt.render("git help tutorial", pt.cv.GREEN)
subtitle += ")"

pt.echo(subtitle)
```

start a working area (see also: git help tutorial)

render() method uses SgrRenderer by default, which is set up automatically depending on output device characteristics and environment setup.

Note that `render()` accepts FT as format argument, which can be Style or Color or *str* or *int* (there is a few methods to define a color).

4.1.2 Fragments

Fragment is a basic class implementing IRenderable interface and contains a text string along with a Style instance and that's it.

`Fragment` instances can be safely concatenated with a regular *str* (but not with another Fragment) from the left side as well as from the right side (line #15). If you attempt to add one `Fragment` to another `Fragment`, you'll end up with a Text instance (see the example after next).

```
from collections.abc import Iterable
import pytermor as pt

data = [
    ("clone", "Clone a repository into a new directory"),
    ("init", "Create an empty Git repository or reinitialize an existing one"),
]

st = pt.Style(fg=pt.cv.GREEN)
for name, desc in data:
    frag = pt.Fragment(name.ljust(16), st)
    pt.echo('  ' + frag + desc)
```

4.1.3 Fragments in f-strings

Another approach to align a formatted text is to combine Python's *f-strings* with Fragment instances.

```
import pytermor as pt

data = [
    ("bisect", "Use binary search to find the commit that introduced a bug"),
    ("diff", "Show changes between commits, commit and working tree, etc"),
    ("grep", "Print lines matching a pattern"),
]

st = pt.Style(fg=pt.cv.GREEN)
for name, desc in data:
    frag = pt.Fragment(name, st)
    pt.echo(f"  {frag:<16s}    {desc}")
```

4.1.4 Texts & FrozenTexts

Text is a general-purpose composite IRenderable implementation, which can contain any amount of strings linked with styles (i.e. Fragment instances).

`Text` also supports aligning, padding with specified chars to specified width, but most importantly it supports fargs syntax (for the details see guide.fargs), which allows to compose formatted text parts much faster and keeps the code compact. Generally speaking, the basic input parameter is either a tuple of string and Style or Color, which then will be applied to preceeding string, or a standalone string. Usually explicit definition of a tuple is not neccessary, but there are cases, when it is.

```
import pytermor as pt

subtitle_st = pt.Style(fg=pt.cv.YELLOW, bold=True)
command_st = pt.Style(fg=pt.cv.GREEN)
text = pt.FrozenText(
    ("work on the current change ", subtitle_st),
    "(see also: ",
    "git help everyday", command_st,
    ")"
)
pt.echo(text)
```

FrozenText is immutable version of Text (to be precise, its quite the opposite: `Text` is a child of `FrozenText`).

We will utilize aligning capabilities of `FrozenText` class in a following code fragment:

```
import pytermor as pt

data = [
    ("add", "Add file contents to the index"),
    ("mv", "Move or rename a file, a directory, or a symlink"),
    ("restore", "Restore working tree files"),
]
st = pt.Style(fg=pt.cv.GREEN)

```

(continues on next page)

(continued from previous page)

```
10 for name, desc in data:
11     pt.echo([pt.FrozenText("  ", name, st, width=18, pad=4), desc])
```

At line #13 we compose a FrozenText instance with command name and set up desired width (18=16+2 for left padding), and explicitly set up right padding with `pad` argument. Padding chars will be applied to the left, right or both sides depending on `align` argument.

Note that although echo() accepts a single RT as a first argument, it also accepts a sequence of them, which allows us to call `echo` just once. RT is a type var including *str* type and all IRenderable implementations.

4.1.5 Templates

There is a support of library's internal tag format, which allows to inline formatting into the original string, and get the final result by calling just one method:

```
1 import pytermor as pt
2
3 s = """@st:[fg=yellow bold] @cmd:[fg=green]
4 :[st]grow, mark and tweak your common history:[-]
5    :[cmd]branch:[-]            List, create, or delete branches
6    :[cmd]commit:[-]            Record changes to the repository
7    :[cmd]merge:[-]             Join two or more development histories together
8 """
9 pt.echo(pt.TemplateEngine().substitute(s))
```

Here `@st:[fg=yellow bold]` is a definition of a custom user style named `st`, `:[st]` is a opening tag for that style, and `:[-]` is a closing tag matching the most recently opened one. See guide.templates for the details.

4.1.6 Regexp group substitution

A little bit artificial example, but this method can be applied to solve real tasks nevertheless. The trick is to apply the desired style to a string containing special characters like `r"\1"`, which will represent regexp group 1 after passing it into `re.sub()`. The actual string being passed as 2nd argument will be `ESC [ 32m \1 ESC [ m`. Regexp substitution function will replace all `\1` with a matching group in every line of the input, therefore the match will end up being surrounded with (already rendered) SGRs responsible for green text color, ???, PROFIT:

```
1  import re
2  import pytermor as pt
3
4  s = """
5     fetch             Download objects and refs from another repository
6     pull              Fetch from and integrate with another repository or a local␣
   ↪branch
7     push              Update remote refs along with associated objects
8  """
9
10 regex = re.compile(r"^(\s+)(\S+)(.+)$")
11 for line in s.splitlines():
```

(continues on next page)

(continued from previous page)

```
    pt.echo(
        regex.sub(
            pt.render(r"\1" + pt.Fragment(r"\2", pt.cv.GREEN) + r"\3"),
            line,
        )
    )

```

For more complex logic it's usually better to extract it into separate function:

```
def replace_expand(m: re.Match) -> str:
    tpl = pt.render(r"\1" + pt.Fragment(r"\2", pt.cv.GREEN) + r"\3")
    return m.expand(tpl)
regex.sub(replace_expand, "...")
```

Another approach:

```
def replace_manual(m: re.Match) -> str:
    return pt.render(m.group(1) + pt.Fragment(m.group(2), pt.cv.GREEN) + m.group(3))
regex.sub(replace_manual, "...")
```

4.1.7 Refilters

Refilters (**Re**ndering **filter**s) are usually applied in sequences, where each of those matches one or two named regexp groups and applies the specified styles accordingly.

In the example below we first (#10-12) implement `_render()` method in a new class inherited from AbstractNamedGroupsRefilter, then (#14-16) the refilter is created (note regexp group name `'cmd'` and matching dictionary key, which value is a FT), then (#19) the refilter is applied and result is printed.

Note: Although filters in general are classified as **low**-level, this example is placed into **high**-level group, because no manipulation at byte level or at color channel level is performed.

```
import re
import pytermor as pt

s = """
   reset             Reset current HEAD to the specified state
   switch            Switch branches
   tag               Create, list, delete or verify a tag object signed with GPG
"""

class SgrNamedGroupsRefilter(pt.AbstractNamedGroupsRefilter):
    def _render(self, v: pt.IT, st: pt.FT) -> str:
        return pt.render(v, st, pt.SgrRenderer)

f = SgrNamedGroupsRefilter(
    re.compile(r"(\s+)(?P<cmd>\S+)(.+)"),
    {"cmd": pt.cv.GREEN},
)

pt.echo(pt.apply_filters(s, f))
```

4.2 Rendering · Low-level

The examples in this part are sorted from simple (for the developer) ones at the beginning to complicated (for the developer) ones at the end. But after you change the point of view, the results are reversed: first ones are most complicated for the interpreter to run, while the ones at the end are simplest (roughly one robust method per instance is invoked). Therefore, the answer to the question "which method is most suitable" should always be evaluated on the individual basis.

4.2.1 Preset compositions

Preset composition methods produce sequence instances or already rendered sequence bytes as if they were rendered by SgrRenderer. Methods with names starting with `make_` return seq. instances, and methods named `compose_*` return *str*, which means that more than one sequence were involved.

In the next example we create an SGR which colors text to black, and bg to 0xffaf00 (line #3), then compose a sequence chain which includes:

- CUP (Cursor Position) instruction: `ESC [1;1H`;
- SGR instruction with our prev. defined colors: `ESC [30;48;5;214m`;
- EL (Erase in Line) instruction: `ESC [0K`.

Effectively this results in a whole terminal line colored with colors specified, and note that we did not fill the line with spaces or something like that – this method is (in theory) faster, because the tty needs to process only ~10-20 characters of input instead of 120+ (average terminal width).

```
1 import pytermor as pt
2
3 col_sgr = pt.make_color_256(214, pt.ColorTarget.BG) + pt.ansi.SeqIndex.BLACK
4 seq = pt.compose_clear_line_fill_bg(col_sgr)
5 pt.echo(seq + 'AAAA    BBBB')
```

Note: `compose_*` methods do not belong to any renderer, so the decision of using or not using these depending on a terminal settings should be made by the developer on a higher level. The suggested implementation of conditional composite sequences would be to request current renderer setup and ensure is_format_allowed returns *True*, in which case it's ok to write composite sequences (as the default renderer already uses them):

```
seq = ""
if pt.RendererManager.get_default().is_format_allowed:
  seq = pt.compose_clear_line_fill_bg(pt.cv.NAVY_BLUE)
pt.echo(seq + 'AAAA    BBBB')
```

Todo: More consistent way of working with composite sequences would be to merge classes from ansi module with classes from text module, i.e. make ISequence children also inherit IRenderable interface and therefore be rendered using the same mechanism as for Text or Fragment, but that would require quite a bit of refactoring and, considering relatively rare usage of pre-rendered composites, was deferred for a time.

4.2.2 Assisted wrapping

Similar to the next one, but here we call helper method enclose(), which automatically builds the closing sequence complement to specified opening one, while there we pick and insert a closing sequence manually.

```
import pytermor as pt

pt.echo(pt.enclose(pt.SeqIndex.CYAN, "imported") + " rich.inspect")
```

4.2.3 Manual wrapping

Pretty straightforward wrapping of target string into a format which, for example, colors the text with a specified color, can be performed with f-stings. All inheritors of ISequence class implement `__str__()` method, which ensures that they can be safely evaluated in f-strings even without format specifying.

Resetter, of closing sequence, in this case can vary; for example, it can be "hard-reset" sequence, which resets the terminal format stack completely (`ESC [m`), or it can be text color reset sequence (`ESC [39m`), or even more exotic ones.

SeqIndex class contains prepared sequences which can be inserted into f-string directly without any modifications.

```
import pytermor as pt

print(f"{pt.SeqIndex.CYAN}imported{pt.SeqIndex.RESET} rich.inspect", end="")
```

4.2.4 Manual instantiating

In case of necessity of some non-standard sequence types or "illegal" parameter values there is also a possibility to build the sequence from the scratch, instantiating one of the base sequence classes and providing required parameters values.

If your case is covered with an existing helper method in term package, use it instead of making new instance directly. This approach will make it easier to maintain the code, if something in internal logic of sequence base classes changes in the future.

```
print(pt.SequenceCSI("J", 2).assemble(), end="")
# equivalent to
print(pt.make_erase_in_display(2).assemble(), end="")
```

4.2.5 Manual assembling (don't do this)

The last resort method which works in 100% is to assemble the sequence char by char manually, contain it as a string in source code and just print it when there is a necessity to do that. The only problem with this approach is an empirical rule, which says:

> Each raw ANSI escape sequence in the source code reduces the readability of the whole file by 50%.

This means that even 2 SGRs would give 25% readability of the original, while 4 SGRs give 6% (this rule is a joke I made up just now, but the key idea should be true).

In short:

- they are hard to modify,
- they are hard to maintain,

- they are hard to debug.

Even if it seems OK for a while:

```
print('\x1b[41m', end="(¬¬)")
print('\x1b[41m\x1b[2J\x1b[1;1H', end="(OO)")
```

...things get worse pretty fast:

```
print('\x1b[38;2;232;232;22m\x1b[1;41m\x1b[2J\x1b[1;1H', end="(°°)")
```

Compare with the next fragment, which does literally the same as the *highlighted line* from the example above, but is much easier to read thanks to low-level abstractions:

```
print(pt.make_color_rgb(232, 232, 22), end="")
print(pt.ansi.SeqIndex.BOLD + pt.ansi.SeqIndex.BG_RED, end="")
print(pt.make_erase_in_display(2).assemble(), end="")
print(pt.make_reset_cursor().assemble(), end="(°~°)")
```

Or after adding some high-level abstractions as well:

```
st = pt.Style(fg=0xe8e816, bg='red', bold=True)
fill = pt.compose_clear_line_fill_bg(st.fg.to_sgr())
pt.echo(fill + "(°v°)", st)
```

5

GUIDE · HIGH-LEVEL

5.1 Core API

5.1.1 Glossary

rendering
: A process of transforming text-describing instances into specified output format, e.g. instance of Fragment class with content and *Style* class containing colors and other text formatting can be rendered into terminal-compatible string with SgrRenderer, or into HTML markup with HtmlRenderer, etc.

style
: Class describing text format options: text color, background color, boldness, underlining, etc. Styles can be inherited and merged with each other. See *Style* constructor description for the details.

color
: Three different classes describing the color options: Color16, Color256 and ColorRGB. The first one corresponds to 16-color terminal mode, the second – to 256-color mode, and the last one represents full RGB color space rather than color index palette. The first two also contain terminal *SGR* bindings.

5.1.2 Core methods

text.render([string, fmt, renderer])	.
text.echo([string, fmt, renderer, nl, file, ...])	.
color.resolve_color(subject[, color_type, ...])	Suggested usage is to transform the user input in a free form in an attempt to find any matching color.
style.make_style([fmt])	General *Style* constructor.
style.merge_styles([origin, fallbacks, ...])	Bulk style merging method.

5.2 Text fragments

5.2.1 Renderable class hierarchy

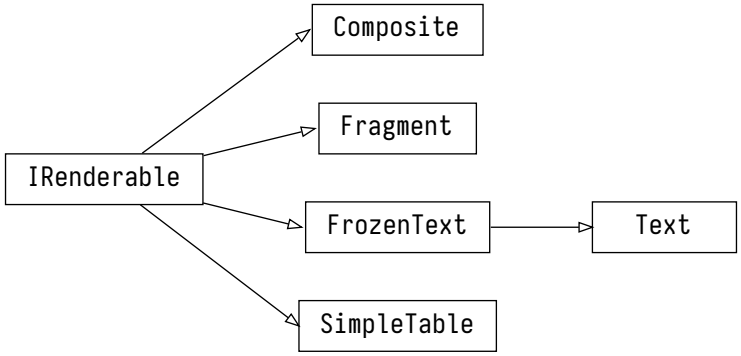


Fig. 1: IRenderable inheritance diagram

5.3 Styles

5.4 Colors

5.4.1 Color mode fallbacks

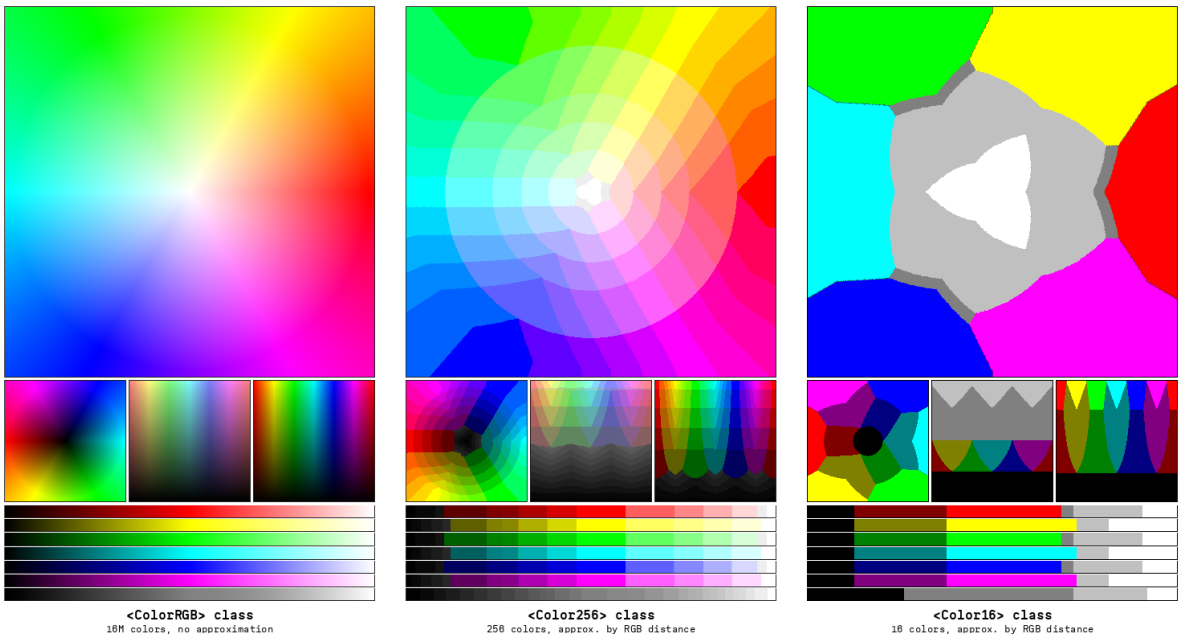


Fig. 2: Color approximations for indexed modes

5.4.2 Color class hierarchy

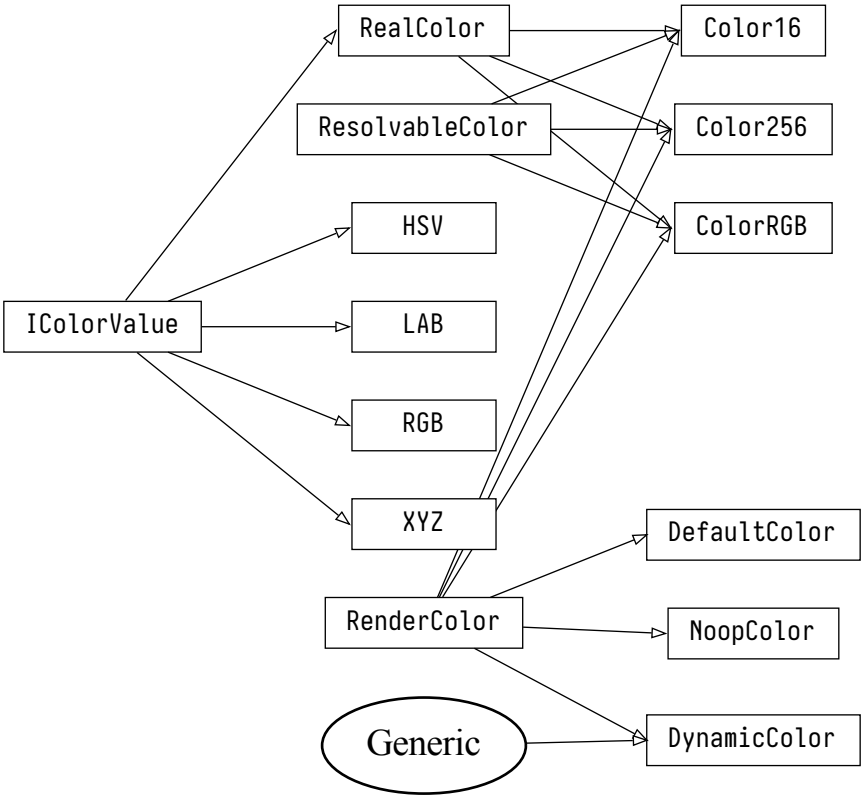


Fig. 3: `Color` inheritance diagram

5.5 fargs syntax

Todo: @TODO

5.6 Renderers

5.6.1 Renderer setup

The library provides options to select the output format, and that option comes in the form of *renderers* .

Selecting the renderer can be accomplished in several ways:

a. By using general-purpose functions render()<text.render> and echo()<text.echo()> – both have an argument `renderer` (preferrable; *introduced in v2.x*).

b. Method RendererManager.set_default() sets the default renderer globally. After that calling render()<text.render> will automatically invoke a said renderer and apply the required formatting (but only

if `renderer` argument of `render()` method is left empty).

c. Set up the config variable Config.renderer_class directly or via environment variable.

d. Use renderer's instance method IRenderer.render() directly, but that's not recommended and possibly will be deprecated in the future.

Generally speaking, if you need to invoke a custom renderer just once, it's convenient to use the first method for this matter, and use the second one in all the other cases.

On the contrary, if there is a necessity to use more than one renderer alternatingly, it's better to avoid using the global one at all, and just instantiate and invoke both renderers independently.

TL;DR

To unconditionally print formatted message to standard output, call force_ansi_rendering() and then render()<text.render>.

5.6.2 Default renderers priority

When it comes to the rendering, RendererManager will use the first non-empty renderer from the list below, skipping the undefined elements:

1. Explicitly specified as argument `renderer` in methods render()<text.render>, echo()<text.echo>, echoi()<text.echoi>.
2. Default renderer in global RendererManager class (see RendererManager.set_default())
3. Renderer class in the current loaded library config: Config.renderer_class.
4. Value from environment variable PYTERMOR_RENDERER_CLASS.
5. Default library renderer SgrRenderer.

Argument > RendererManager > Config > Environment > Library's default

5.6.3 Output mode auto-selection

SgrRenderer can be set up with automatic output mode OutputMode.AUTO. In that case the renderer will return OutputMode.NO_ANSI for any output device other than terminal emulator, or try to find a matching rule from this list:

Table 1: Auto output mode parameters and results

Is a tty?	`TERM` env. var	`COLORTERM` env. var[1]	Result output mode
<any>			*Config.force_output_mode*[2]
No	<any>		NO_ANSI
Yes	`xterm-256color`	`24bit`, `truecolor`	TRUE_COLOR
	`*-256color`[3]	<any>	XTERM_256
	`xterm-color`	<any>	XTERM_16
	`xterm`	<any>	NO_ANSI
	<any other>	<any>	*Config.default_output_mode*[4]

[1] should both env. var requirements be present, they both must be true as well (i.e. logical AND is applied).
[2] empty by default and thus ignored
[3] * represents any string; that's how e.g. *bash 5* determines the color support.
[4] XTERM_256 by default, but can be customized.

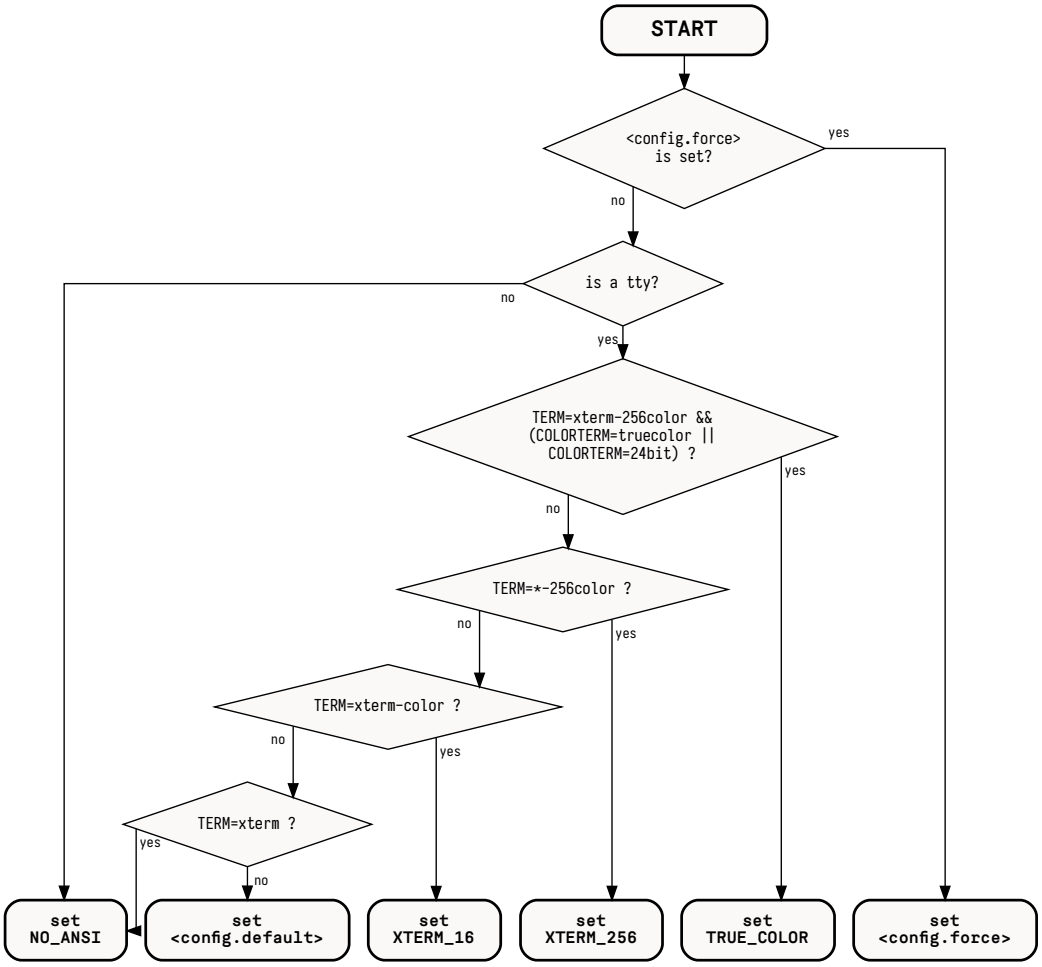


Fig. 4: Auto output mode algorithm

5.6.4 Renderer class hierarchy

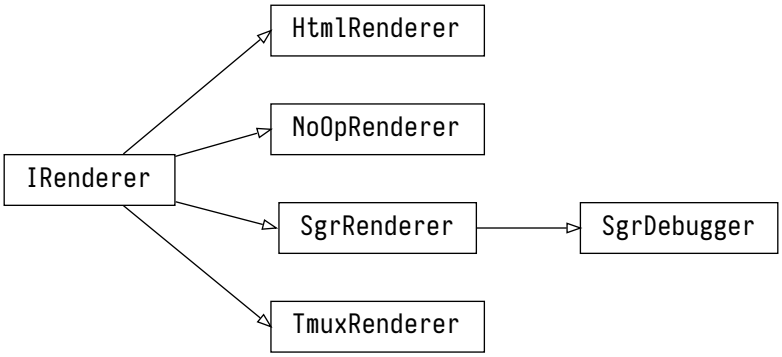


Fig. 5: IRenderer inheritance tree

5.7 Templates

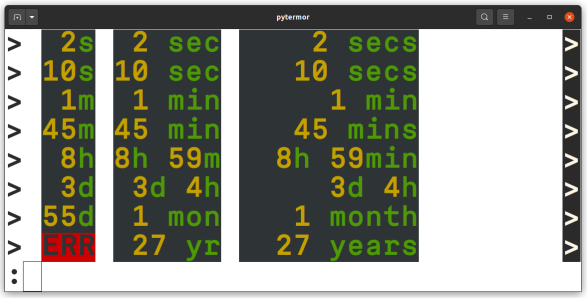
5.8 Number formatters

Todo: The library contains @TODO

5.8.1 Auto-float formatter

5.8.2 Prefixed-unit formatter

5.8.3 Time delta formatter



5.8.4 NumFormatter class hierarchy

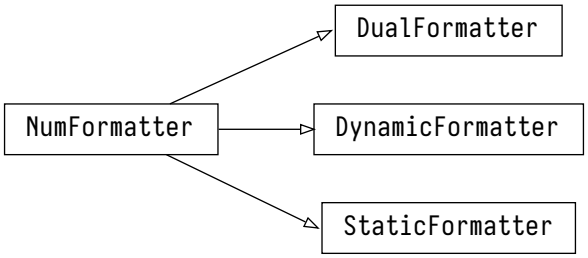


Fig. 6: NumFormatter inheritance tree

5.9 Named colors collection

Todo: @TODO

5.10 Dynamic/deferred colors

Todo: @TODO

6

GUIDE · LOW-LEVEL

6.1 Core API II

So, what's happening under the hood?

6.1.1 Glossary

ASCII
: Basic charset developed back in 1960s, consisting of 128 code points. Nevertheless it is still used nowadays as a part of Unicode character set.

ANSI
: ..escape sequence is a standard for in-band signaling to control cursor location, color, font styling, and other options on video text terminals and terminal emulators. Certain sequences of bytes, most starting with an *ASCII* escape character (ESC 0x1B) and a bracket character ([0x5B), are embedded into text. The terminal interprets these sequences as commands, rather than text to display verbatim.[1]

SGR
: ..sequence is a subtype of *ANSI* escape sequences with a varying amount of parameters. SGR sequences used for changing the color of text or/and terminal background (in 3 different color modes), as well as for decorating text with italic font, underline, overline, cross-line, making it bold or blinking etc. Represented by SequenceSGR class.

[1] https://en.wikipedia.org/wiki/ANSI_escape_code

6.1.2 Core methods

ansi.SequenceSGR(*params)	Class representing SGR (Select Graphic Rendition)-type escape sequence with varying amount of parameters.
term.make_color_256(code[, target])	Wrapper for creation of SequenceSGR that sets foreground (or background) to one of 256-color palette value.:
term.make_color_rgb(r, g, b[, target])	Wrapper for creation of SequenceSGR operating in True Color mode (16M). Valid values for `r`, `g` and `b` are in range of [0; 255]. This range linearly translates into [0x00; 0xFF] for each channel. The result value is composed as "#RRGGBB". For example, a sequence with color of #ff3300 can be created with::.
color.Color256.to_sgr([target, upper_bound])	Make an SGR sequence<SequenceSGR> out of `Color`.

Sources

1. XTerm Control Sequences
2. ECMA-48 specification

6.2 SGR sequences

6.2.1 Format soft reset

Todo: This is how you **should** format examples:

Fig. 1: https://chrisyeh96.github.io/2020/03/28/terminal-colors.html#color-schemes

There are two ways to manage color and attribute termination:

- hard reset (SGR-0 or `ESC [0m`)
- soft reset (SGR-22, 23, 24 etc.)

The main difference between them is that *hard* reset disables all formatting after itself, while *soft* reset disables only actually necessary attributes (i.e. used as opening sequence in Span instance's context) and keeps the other.

That's what `Span` class is designed for: to simplify creation of soft-resetting text spans, so that developer doesn't have to restore all previously applied formats after every closing sequence.

Example

We are given a text span which is initially *bold* and *underlined*. We want to recolor a few words inside of this span. By default this will result in losing all the formatting to the right of updated text span (because RESET <SeqIndex.RESET>, or `ESC [0m`, clears all text attributes).

However, there is an option to specify what attributes should be disabled or let the library do that for you:

As you can see, the update went well – we kept all the previously applied formatting. Of course, this method cannot be 100% applicable; for example, imagine that original text was colored blue. After the update "string" word won't be blue anymore, as we used `SeqIndex.COLOR_OFF` escape sequence to neutralize our own yellow color. But it still can be helpful for a majority of cases (especially when text is generated and formatted by the same program and in one go).

6.2.2 Working with Spans

Use Span constructor to create new instance with specified control sequence(s) as a opening/starter sequence and **automatically composed** closing sequence that will terminate attributes defined in opening sequence while keeping the others (soft reset).

Resulting sequence params' order is the same as argument's order.

Each sequence param can be specified as:

- string key (see ansi-presets);
- integer param value;
- existing SequenceSGR instance (params will be extracted).

It's also possible to avoid auto-composing mechanism and create Span with explicitly set parameters using Span.init_explicit().

6.2.3 Creating and applying SGRs

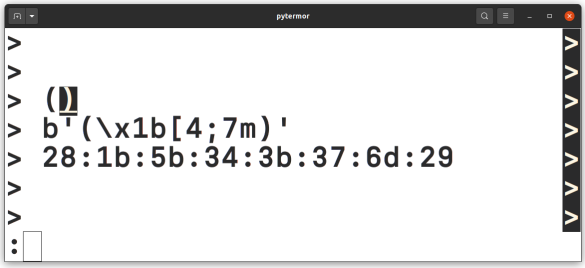
You can use any of predefined sequences from SeqIndex registry or create your own via standard constructor. Valid argument values as well as preset constants are described in ansi-presets page.

Important: SequenceSGR with zero params `ESC [m` is interpreted by terminal emulators as `ESC [0m`, which is *hard* reset sequence.

There is also a set of methods for dynamic SequenceSGR creation:

- make_color_256() will produce sequence operating in 256-colors mode (for a complete list see ansi-presets);
- make_color_rgb() will create a sequence capable of setting the colors in True Color 16M mode (however, some terminal emulators doesn't support it).

To get the resulting sequence chars use assemble() <SequenceSGR.assemble()> method or cast instance to *str*.



- First line is the string with encoded escape sequence;
- Second line shows up the string in raw mode, as if sequences were ignored by the terminal;
- Third line is hexadecimal string representation.

6.2.4 SGR sequence structure

1. `ESC` is escape *control character*, which opens a control sequence (can also be written as `\x1b`, `\033` or `\e`).
2. `[` is sequence *classifier*; it determines the type of control sequence (in this case it's CSI (Control Sequence Introducer)).
3. `4` and `7` are *parameters* of the escape sequence; they mean "underlined" and "inversed" attributes respectively. Those parameters must be separated by `;`.
4. `m` is sequence *terminator*; it also determines the sub-type of sequence, in our case SGR. Sequences of this kind are most commonly encountered.

6.2.5 Combining SGRs

One instance of SequenceSGR can be added to another. This will result in a new `SequenceSGR` with combined params.

6.2.6 Sequence class hierarchy

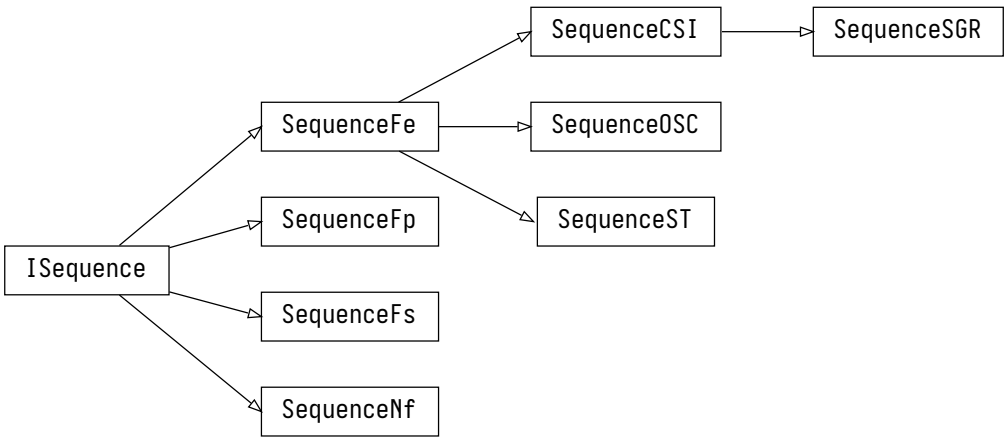


Fig. 2: ISequence inheritance tree

6.3 Sequence presets

Preset lists are omitted from API docs to avoid unnesessary duplication; summary list of all presets defined in the library (excluding util* <util>) is displayed here.

Todo: USAGE - list all memthods that accept string keys of those prsets.

There are two types of color palettes used in modern terminals – first one containing 16 colors (Color16), and second one consisting of 256 colors (Color256). There is also True Color mode (referenced as *RGB* mode), but it is not palette-based.

Legend

- INT (intcode module -- 1st or 3rd SGR param value)
- STY (style module)

6.3.1 Meta, attributes, resetters

	Name	INT	STY	Description
Meta				
	NOOP		V	No-operation; always assembled as empty string
	RESET	0		Reset all attributes and colors
Attributes				
	BOLD	1	V[1]	Bold or increased intensity
	DIM	2	V	Faint, decreased intensity
	ITALIC	3	V	Italic; *not widely supported*
	UNDERLINED	4	V	Underline
	BLINK_SLOW	5	V[2]	Set blinking to < 150 cpm
	BLINK_FAST	6		Set blinking to 150+ cpm; *not widely supported*
	INVERSED	7	V	Swap foreground and background colors
	HIDDEN	8		Conceal characters; *not widely supported*
	CROSSLINED	9	V	Strikethrough
	DOUBLE_UNDERLINED	21		Double-underline; *on several terminals disables* BOLD *instead*
	COLOR_EXTENDED	38		Set foreground color [*indexed*/*RGB* mode]; *use* make_color_256 *and* make_color_rgb *instead*
	BG_COLOR_EXTENDED	48		Set background color [*indexed*/*RGB* mode]; *use* make_color_256 *and* make_color_rgb *instead*
	OVERLINED	53	V	Overline; *not widely supported*
Resetters				
	BOLD_DIM_OFF	22		Disable BOLD and DIM attributes. *Special aspects... It's impossible to reliably disable them on a separate basis.*
	ITALIC_OFF	23		Disable italic
	UNDERLINED_OFF	24		Disable underlining
	BLINK_OFF	25		Disable blinking
	INVERSED_OFF	27		Disable inversing
	HIDDEN_OFF	28		Disable conecaling
	CROSSLINED_OFF	29		Disable strikethrough
	COLOR_OFF	39		Reset foreground color
	BG_COLOR_OFF	49		Reset background color
	OVERLINED_OFF	55		Disable overlining

6.3.2 Color16 presets

	Name	INT	STY	RGB code	XTerm name
Foreground *default* colors					
	BLACK	30		#000000	Black
	RED	31		#800000	Maroon
	GREEN	32		#008000	Green
	YELLOW	33		#808000	Olive

continues on next page

[1] for this and subsequent items in "Attributes" section: as boolean flags.
[2] as blink.

Table 1 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	BLUE	34		#000080	Navy
	MAGENTA	35		#800080	Purple
	CYAN	36		#008080	Teal
	WHITE	37		#c0c0c0	Silver
Background *default* colors					
	BG_BLACK	40		#000000	Black
	BG_RED	41		#800000	Maroon
	BG_GREEN	42		#008000	Green
	BG_YELLOW	43		#808000	Olive
	BG_BLUE	44		#000080	Navy
	BG_MAGENTA	45		#800080	Purple
	BG_CYAN	46		#008080	Teal
	BG_WHITE	47		#c0c0c0	Silver
High-intensity foreground *default* colors					
	GRAY	90		#808080	Grey
	HI_RED	91		#ff0000	Red
	HI_GREEN	92		#00ff00	Lime
	HI_YELLOW	93		#ffff00	Yellow
	HI_BLUE	94		#0000ff	Blue
	HI_MAGENTA	95		#ff00ff	Fuchsia
	HI_CYAN	96		#00ffff	Aqua
	HI_WHITE	97		#ffffff	White
High-intensity background *default* colors					
	BG_GRAY	100		#808080	Grey
	BG_HI_RED	101		#ff0000	Red
	BG_HI_GREEN	102		#00ff00	Lime
	BG_HI_YELLOW	103		#ffff00	Yellow
	BG_HI_BLUE	104		#0000ff	Blue
	BG_HI_MAGENTA	105		#ff00ff	Fuchsia
	BG_HI_CYAN	106		#00ffff	Aqua
	BG_HI_WHITE	107		#ffffff	White

6.3.3 Color256 presets

	Name	INT	STY	RGB code	XTerm name
	XTERM_BLACK[3]	0		#000000	
	XTERM_MAROON	1		#800000	
	XTERM_GREEN	2		#008000	
	XTERM_OLIVE	3		#808000	
	XTERM_NAVY	4		#000080	
	XTERM_PURPLE_5	5		#800080	**Purple**[4]
	XTERM_TEAL	6		#008080	
	XTERM_SILVER	7		#c0c0c0	
	XTERM_GREY	8		#808080	
	XTERM_RED	9		#ff0000	
	XTERM_LIME	10		#00ff00	

continues on next page

Table 2 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	XTERM_YELLOW	11		#ffff00	
	XTERM_BLUE	12		#0000ff	
	XTERM_FUCHSIA	13		#ff00ff	
	XTERM_AQUA	14		#00ffff	
	XTERM_WHITE	15		#ffffff	
	XTERM_GREY_0	16		#000000	
	XTERM_NAVY_BLUE	17		#00005f	
	XTERM_DARK_BLUE	18		#000087	
	XTERM_BLUE_3	19		#0000af	
	XTERM_BLUE_2	20		#0000d7	**Blue3**
	XTERM_BLUE_1	21		#0000ff	
	XTERM_DARK_GREEN	22		#005f00	
	XTERM_DEEP_SKY_BLUE_7	23		#005f5f	**DeepSkyBlue4**
	XTERM_DEEP_SKY_BLUE_6	24		#005f87	**DeepSkyBlue4**
	XTERM_DEEP_SKY_BLUE_5	25		#005faf	**DeepSkyBlue4**
	XTERM_DODGER_BLUE_3	26		#005fd7	
	XTERM_DODGER_BLUE_2	27		#005fff	
	XTERM_GREEN_5	28		#008700	**Green4**
	XTERM_SPRING_GREEN_4	29		#00875f	
	XTERM_TURQUOISE_4	30		#008787	
	XTERM_DEEP_SKY_BLUE_4	31		#0087af	**DeepSkyBlue3**
	XTERM_DEEP_SKY_BLUE_3	32		#0087d7	
	XTERM_DODGER_BLUE_1	33		#0087ff	
	XTERM_GREEN_4	34		#00af00	**Green3**
	XTERM_SPRING_GREEN_5	35		#00af5f	**SpringGreen3**
	XTERM_DARK_CYAN	36		#00af87	
	XTERM_LIGHT_SEA_GREEN	37		#00afaf	
	XTERM_DEEP_SKY_BLUE_2	38		#00afd7	
	XTERM_DEEP_SKY_BLUE_1	39		#00afff	
	XTERM_GREEN_3	40		#00d700	
	XTERM_SPRING_GREEN_3	41		#00d75f	
	XTERM_SPRING_GREEN_6	42		#00d787	**SpringGreen2**
	XTERM_CYAN_3	43		#00d7af	
	XTERM_DARK_TURQUOISE	44		#00d7d7	
	XTERM_TURQUOISE_2	45		#00d7ff	
	XTERM_GREEN_2	46		#00ff00	**Green1**
	XTERM_SPRING_GREEN_2	47		#00ff5f	
	XTERM_SPRING_GREEN_1	48		#00ff87	
	XTERM_MEDIUM_SPRING_GREEN	49		#00ffaf	
	XTERM_CYAN_2	50		#00ffd7	
	XTERM_CYAN_1	51		#00ffff	
	XTERM_DARK_RED_2	52		#5f0000	**DarkRed**
	XTERM_DEEP_PINK_8	53		#5f005f	**DeepPink4**
	XTERM_PURPLE_6	54		#5f0087	**Purple4**
	XTERM_PURPLE_4	55		#5f00af	
	XTERM_PURPLE_3	56		#5f00d7	
	XTERM_BLUE_VIOLET	57		#5f00ff	
	XTERM_ORANGE_4	58		#5f5f00	
	XTERM_GREY_37	59		#5f5f5f	
	XTERM_MEDIUM_PURPLE_7	60		#5f5f87	**MediumPurple4**
	XTERM_SLATE_BLUE_3	61		#5f5faf	
	XTERM_SLATE_BLUE_2	62		#5f5fd7	**SlateBlue3**
	XTERM_ROYAL_BLUE_1	63		#5f5fff	

continues on next page

Table 2 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	XTERM_CHARTREUSE_6	64		#5f8700	**Chartreuse4**
	XTERM_DARK_SEA_GREEN_9	65		#5f875f	**DarkSeaGreen4**
	XTERM_PALE_TURQUOISE_4	66		#5f8787	
	XTERM_STEEL_BLUE	67		#5f87af	
	XTERM_STEEL_BLUE_3	68		#5f87d7	
	XTERM_CORNFLOWER_BLUE	69		#5f87ff	
	XTERM_CHARTREUSE_5	70		#5faf00	**Chartreuse3**
	XTERM_DARK_SEA_GREEN_8	71		#5faf5f	**DarkSeaGreen4**
	XTERM_CADET_BLUE_2	72		#5faf87	**CadetBlue**
	XTERM_CADET_BLUE	73		#5fafaf	
	XTERM_SKY_BLUE_3	74		#5fafd7	
	XTERM_STEEL_BLUE_2	75		#5fafff	**SteelBlue1**
	XTERM_CHARTREUSE_4	76		#5fd700	**Chartreuse3**
	XTERM_PALE_GREEN_4	77		#5fd75f	**PaleGreen3**
	XTERM_SEA_GREEN_3	78		#5fd787	
	XTERM_AQUAMARINE_3	79		#5fd7af	
	XTERM_MEDIUM_TURQUOISE	80		#5fd7d7	
	XTERM_STEEL_BLUE_1	81		#5fd7ff	
	XTERM_CHARTREUSE_2	82		#5fff00	
	XTERM_SEA_GREEN_4	83		#5fff5f	**SeaGreen2**
	XTERM_SEA_GREEN_2	84		#5fff87	**SeaGreen1**
	XTERM_SEA_GREEN_1	85		#5fffaf	
	XTERM_AQUAMARINE_2	86		#5fffd7	**Aquamarine1**
	XTERM_DARK_SLATE_GRAY_2	87		#5fffff	
	XTERM_DARK_RED	88		#870000	
	XTERM_DEEP_PINK_7	89		#87005f	**DeepPink4**
	XTERM_DARK_MAGENTA_2	90		#870087	**DarkMagenta**
	XTERM_DARK_MAGENTA	91		#8700af	
	XTERM_DARK_VIOLET_2	92		#8700d7	**DarkViolet**
	XTERM_PURPLE_2	93		#8700ff	**Purple**
	XTERM_ORANGE_3	94		#875f00	**Orange4**
	XTERM_LIGHT_PINK_3	95		#875f5f	**LightPink4**
	XTERM_PLUM_4	96		#875f87	
	XTERM_MEDIUM_PURPLE_6	97		#875faf	**MediumPurple3**
	XTERM_MEDIUM_PURPLE_5	98		#875fd7	**MediumPurple3**
	XTERM_SLATE_BLUE_1	99		#875fff	
	XTERM_YELLOW_6	100		#878700	**Yellow4**
	XTERM_WHEAT_4	101		#87875f	
	XTERM_GREY_53	102		#878787	
	XTERM_LIGHT_SLATE_GREY	103		#8787af	
	XTERM_MEDIUM_PURPLE_4	104		#8787d7	**MediumPurple**
	XTERM_LIGHT_SLATE_BLUE	105		#8787ff	
	XTERM_YELLOW_4	106		#87af00	
	XTERM_DARK_OLIVE_GREEN_6	107		#87af5f	**DarkOliveGreen3**
	XTERM_DARK_SEA_GREEN_7	108		#87af87	**DarkSeaGreen**
	XTERM_LIGHT_SKY_BLUE_3	109		#87afaf	
	XTERM_LIGHT_SKY_BLUE_2	110		#87afd7	**LightSkyBlue3**
	XTERM_SKY_BLUE_2	111		#87afff	
	XTERM_CHARTREUSE_3	112		#87d700	**Chartreuse2**
	XTERM_DARK_OLIVE_GREEN_4	113		#87d75f	**DarkOliveGreen3**
	XTERM_PALE_GREEN_3	114		#87d787	
	XTERM_DARK_SEA_GREEN_5	115		#87d7af	**DarkSeaGreen3**
	XTERM_DARK_SLATE_GRAY_3	116		#87d7d7	

continues on next page

Table 2 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	XTERM_SKY_BLUE_1	117		#87d7ff	
	XTERM_CHARTREUSE_1	118		#87ff00	
	XTERM_LIGHT_GREEN_2	119		#87ff5f	**LightGreen**
	XTERM_LIGHT_GREEN	120		#87ff87	
	XTERM_PALE_GREEN_1	121		#87ffaf	
	XTERM_AQUAMARINE_1	122		#87ffd7	
	XTERM_DARK_SLATE_GRAY_1	123		#87ffff	
	XTERM_RED_4	124		#af0000	**Red3**
	XTERM_DEEP_PINK_6	125		#af005f	**DeepPink4**
	XTERM_MEDIUM_VIOLET_RED	126		#af0087	
	XTERM_MAGENTA_6	127		#af00af	**Magenta3**
	XTERM_DARK_VIOLET	128		#af00d7	
	XTERM_PURPLE	129		#af00ff	
	XTERM_DARK_ORANGE_3	130		#af5f00	
	XTERM_INDIAN_RED_4	131		#af5f5f	**IndianRed**
	XTERM_HOT_PINK_5	132		#af5f87	**HotPink3**
	XTERM_MEDIUM_ORCHID_4	133		#af5faf	**MediumOrchid3**
	XTERM_MEDIUM_ORCHID_3	134		#af5fd7	**MediumOrchid**
	XTERM_MEDIUM_PURPLE_2	135		#af5fff	
	XTERM_DARK_GOLDENROD	136		#af8700	
	XTERM_LIGHT_SALMON_3	137		#af875f	
	XTERM_ROSY_BROWN	138		#af8787	
	XTERM_GREY_63	139		#af87af	
	XTERM_MEDIUM_PURPLE_3	140		#af87d7	**MediumPurple2**
	XTERM_MEDIUM_PURPLE_1	141		#af87ff	
	XTERM_GOLD_3	142		#afaf00	
	XTERM_DARK_KHAKI	143		#afaf5f	
	XTERM_NAVAJO_WHITE_3	144		#afaf87	
	XTERM_GREY_69	145		#afafaf	
	XTERM_LIGHT_STEEL_BLUE_3	146		#afafd7	
	XTERM_LIGHT_STEEL_BLUE_2	147		#afafff	**LightSteelBlue**
	XTERM_YELLOW_5	148		#afd700	**Yellow3**
	XTERM_DARK_OLIVE_GREEN_5	149		#afd75f	**DarkOliveGreen3**
	XTERM_DARK_SEA_GREEN_6	150		#afd787	**DarkSeaGreen3**
	XTERM_DARK_SEA_GREEN_4	151		#afd7af	**DarkSeaGreen2**
	XTERM_LIGHT_CYAN_3	152		#afd7d7	
	XTERM_LIGHT_SKY_BLUE_1	153		#afd7ff	
	XTERM_GREEN_YELLOW	154		#afff00	
	XTERM_DARK_OLIVE_GREEN_3	155		#afff5f	**DarkOliveGreen2**
	XTERM_PALE_GREEN_2	156		#afff87	**PaleGreen1**
	XTERM_DARK_SEA_GREEN_3	157		#afffaf	**DarkSeaGreen2**
	XTERM_DARK_SEA_GREEN_1	158		#afffd7	
	XTERM_PALE_TURQUOISE_1	159		#afffff	
	XTERM_RED_3	160		#d70000	
	XTERM_DEEP_PINK_5	161		#d7005f	**DeepPink3**
	XTERM_DEEP_PINK_3	162		#d70087	
	XTERM_MAGENTA_3	163		#d700af	
	XTERM_MAGENTA_5	164		#d700d7	**Magenta3**
	XTERM_MAGENTA_4	165		#d700ff	**Magenta2**
	XTERM_DARK_ORANGE_2	166		#d75f00	**DarkOrange3**
	XTERM_INDIAN_RED_3	167		#d75f5f	**IndianRed**
	XTERM_HOT_PINK_4	168		#d75f87	**HotPink3**
	XTERM_HOT_PINK_3	169		#d75faf	**HotPink2**

continues on next page

Table 2 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	XTERM_ORCHID_3	170		#d75fd7	**Orchid**
	XTERM_MEDIUM_ORCHID_2	171		#d75fff	**MediumOrchid1**
	XTERM_ORANGE_2	172		#d78700	**Orange3**
	XTERM_LIGHT_SALMON_2	173		#d7875f	**LightSalmon3**
	XTERM_LIGHT_PINK_2	174		#d78787	**LightPink3**
	XTERM_PINK_3	175		#d787af	
	XTERM_PLUM_3	176		#d787d7	
	XTERM_VIOLET	177		#d787ff	
	XTERM_GOLD_2	178		#d7af00	**Gold3**
	XTERM_LIGHT_GOLDENROD_5	179		#d7af5f	**LightGoldenrod3**
	XTERM_TAN	180		#d7af87	
	XTERM_MISTY_ROSE_3	181		#d7afaf	
	XTERM_THISTLE_3	182		#d7afd7	
	XTERM_PLUM_2	183		#d7afff	
	XTERM_YELLOW_3	184		#d7d700	
	XTERM_KHAKI_3	185		#d7d75f	
	XTERM_LIGHT_GOLDENROD_3	186		#d7d787	**LightGoldenrod2**
	XTERM_LIGHT_YELLOW_3	187		#d7d7af	
	XTERM_GREY_84	188		#d7d7d7	
	XTERM_LIGHT_STEEL_BLUE_1	189		#d7d7ff	
	XTERM_YELLOW_2	190		#d7ff00	
	XTERM_DARK_OLIVE_GREEN_2	191		#d7ff5f	**DarkOliveGreen1**
	XTERM_DARK_OLIVE_GREEN_1	192		#d7ff87	
	XTERM_DARK_SEA_GREEN_2	193		#d7ffaf	**DarkSeaGreen1**
	XTERM_HONEYDEW_2	194		#d7ffd7	
	XTERM_LIGHT_CYAN_1	195		#d7ffff	
	XTERM_RED_1	196		#ff0000	
	XTERM_DEEP_PINK_4	197		#ff005f	**DeepPink2**
	XTERM_DEEP_PINK_2	198		#ff0087	**DeepPink1**
	XTERM_DEEP_PINK_1	199		#ff00af	
	XTERM_MAGENTA_2	200		#ff00d7	
	XTERM_MAGENTA_1	201		#ff00ff	
	XTERM_ORANGE_RED_1	202		#ff5f00	
	XTERM_INDIAN_RED_1	203		#ff5f5f	
	XTERM_INDIAN_RED_2	204		#ff5f87	**IndianRed1**
	XTERM_HOT_PINK_2	205		#ff5faf	**HotPink**
	XTERM_HOT_PINK	206		#ff5fd7	
	XTERM_MEDIUM_ORCHID_1	207		#ff5fff	
	XTERM_DARK_ORANGE	208		#ff8700	
	XTERM_SALMON_1	209		#ff875f	
	XTERM_LIGHT_CORAL	210		#ff8787	
	XTERM_PALE_VIOLET_RED_1	211		#ff87af	
	XTERM_ORCHID_2	212		#ff87d7	
	XTERM_ORCHID_1	213		#ff87ff	
	XTERM_ORANGE_1	214		#ffaf00	
	XTERM_SANDY_BROWN	215		#ffaf5f	
	XTERM_LIGHT_SALMON_1	216		#ffaf87	
	XTERM_LIGHT_PINK_1	217		#ffafaf	
	XTERM_PINK_1	218		#ffafd7	
	XTERM_PLUM_1	219		#ffafff	
	XTERM_GOLD_1	220		#ffd700	
	XTERM_LIGHT_GOLDENROD_4	221		#ffd75f	**LightGoldenrod2**
	XTERM_LIGHT_GOLDENROD_2	222		#ffd787	

continues on next page

Table 2 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	XTERM_NAVAJO_WHITE_1	223		#ffd7af	
	XTERM_MISTY_ROSE_1	224		#ffd7d7	
	XTERM_THISTLE_1	225		#ffd7ff	
	XTERM_YELLOW_1	226		#ffff00	
	XTERM_LIGHT_GOLDENROD_1	227		#ffff5f	
	XTERM_KHAKI_1	228		#ffff87	
	XTERM_WHEAT_1	229		#ffffaf	
	XTERM_CORNSILK_1	230		#ffffd7	
	XTERM_GREY_100	231		#ffffff	
	XTERM_GREY_3	232		#080808	
	XTERM_GREY_7	233		#121212	
	XTERM_GREY_11	234		#1c1c1c	
	XTERM_GREY_15	235		#262626	
	XTERM_GREY_19	236		#303030	
	XTERM_GREY_23	237		#3a3a3a	
	XTERM_GREY_27	238		#444444	
	XTERM_GREY_30	239		#4e4e4e	
	XTERM_GREY_35	240		#585858	
	XTERM_GREY_39	241		#626262	
	XTERM_GREY_42	242		#6c6c6c	
	XTERM_GREY_46	243		#767676	
	XTERM_GREY_50	244		#808080	
	XTERM_GREY_54	245		#8a8a8a	
	XTERM_GREY_58	246		#949494	
	XTERM_GREY_62	247		#9e9e9e	
	XTERM_GREY_66	248		#a8a8a8	
	XTERM_GREY_70	249		#b2b2b2	
	XTERM_GREY_74	250		#bcbcbc	
	XTERM_GREY_78	251		#c6c6c6	
	XTERM_GREY_82	252		#d0d0d0	
	XTERM_GREY_85	253		#dadada	
	XTERM_GREY_89	254		#e4e4e4	
	XTERM_GREY_93	255		#eeeeee	

Sources

1. https://en.wikipedia.org/wiki/ANSI_escape_code
2. https://www.ditig.com/256-colors-cheat-sheet

[3] First 16 colors are effectively the same as colors in *default* 16-color mode and share with them the same color values (and depend on terminal color scheme as well).

[4] XTerm name list contains duplicates; variable names for these were slightly modified (different numbers at the end) to avoid namespace conflicts. Every changed name is displayed with **bold** font.

6.4 xterm-256 palette

Actual colors of *default* palette depend on user's terminal settings, i.e. the result color of Color16 is not guaranteed to exactly match the corresponding color listed below. What's more, note that *default* palette is actually a part of *indexed* one (first 16 colors of 256-color table).

Todo: **(Verify)** The approximation algomanrithm was explicitly made to ignore these colors because otherwise the results of transforming *RGB* values into *indexed* ones would be unpredictable, in addition to different results for different users, depending on their terminal emulator setup.

However, it doesn't mean that Color16 is useless. Just the opposite – it's ideal for situtations when you don't actually **have to** set exact values and it's easier to specify estimation of desired color. I.e. setting color to `'red'` is usually more than enough for displaying an error message – we don't really care of precise hue or brightness values for it.

Todo: Approximation algorithm is as simple as iterating through all colors in the *lookup table* (which contains all possible …

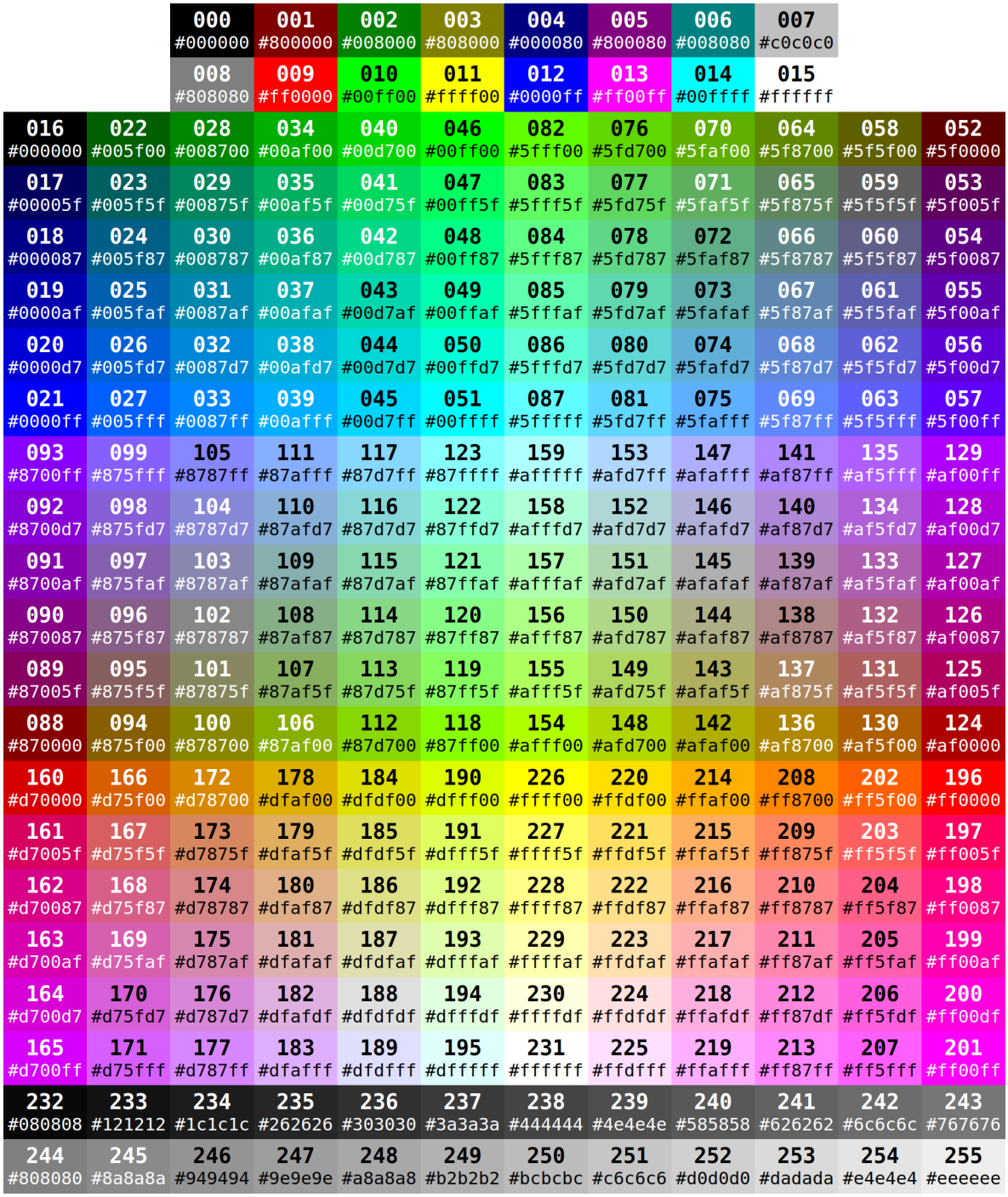


Fig. 3: *Indexed* mode palette

Sources

1. https://www.tweaking4all.com/software/linux-software/xterm-color-cheat-sheet/

6.5 ANSI sequences review

6.5.1 Sequence classes

Sequences can be divided to 4 different classes depending on their classifier byte(s); a class indicates the application domain the purpose of the sequence in general. According to ECMA-48 specification the classes are: **nF**, **Fp**, **Fe**, **Fs**.

- **nF** escape sequences are mostly used for ANSI/ISO code-switching mechanisms. All **nF**-class sequences start with ESC plus ASCII byte from the range 0x20-0x2F: (!"#$%&'()*+\-./ and space).

  They are represented by *SequenceNf* class without any specific implementations.

- **fP**-class sequences can be used for invoking private control functions. The characteristic property is that the first byte after ESC is always in range 0x30-0x3F (0123456789:;<=>?).

  They are represented by *SequenceFp* class, which, for example, assembles DECSC (Save Cursor) and DECRC (Restore Cursor) sequence types.

- **Fe**-class sequences are the most common ones and 99% of the sequences you will ever encounter will be of **Fe** class. ECMA-48 names them "C1 set sequences", and their *classifier* byte (the one right after escape byte) is from 0x40 to 0x5F range (@[\\]_^ABCDEFGHIJKLMNOPQRSTUVWXYZ).

  These sequences are implemented in SequenceFe parent class, which is then subclassed by even more specific classes SequenceST, SequenceOSC, SequenceCSI and *(drums)* SequenceSGR – the one responsible for setting the terminal colors and formats (or at least the majority of them), and also the one that's going to be encountered most of the time. The examples include CUP, ED (Erase in Display), aforementioned SGR and much more.

- **Fs**-class sequences ...

Todo: This

6.5.2 Sequence types

ECMA-48 introduces a list of terminal control functions and contains the implementation details and formats. Each of these usually has a 3+ letters abbreviation (SGR, CSI, EL, etc.) which determines the action that will be performed after the terminal receives control sequence of this function. Let's identify these abbreviations as sequence types.

At the time of writing (v2.75) ansi module contains the implementations of about 25 control sequence types (that should be read as "has seperated classes and/or factory methods and is also documented). However, ECMA-48 standard mentions about 160 sequence types.

The main principle of pytermor development was the rule *"if I don't see it, it doesn't exist"*, which should be read as "Don't waste days and nights on specs comprehension and implementation of the features no one ever will use".

That's why the only types of sequences implemented are the ones that I personally encountered in the modern environment (and having a practical application, of course).

However, the library was designed to provide an easy way to extend the control sequences class hierarchy; what's more, this includes not only the extendability of the library itself (i.e., improvements in the context of library source code), but also the extra logic in the client code referencing the library classes. In case something important is missed – there is an Issues page on the GitHub, you are welcome to make a feature request.

6.6 Parser

6.7 Filters

6.7.1 Filter class hierarchy

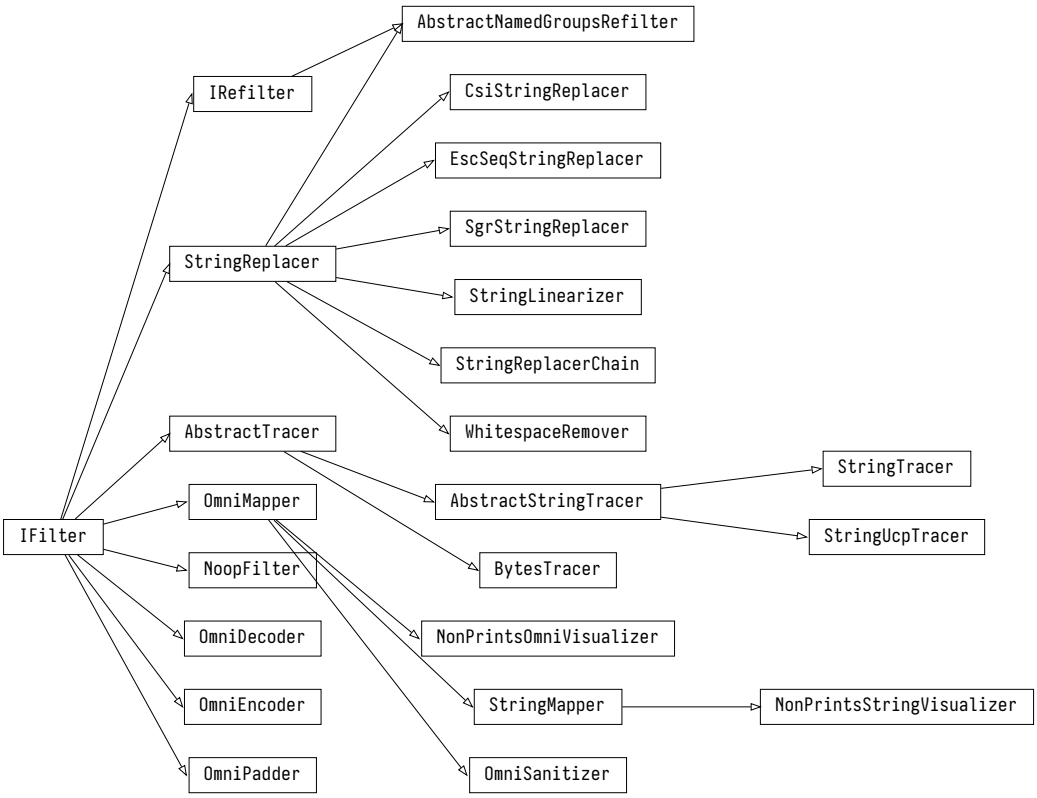


Fig. 4: IFilter inheritance tree

6.8 Color spaces and transformations

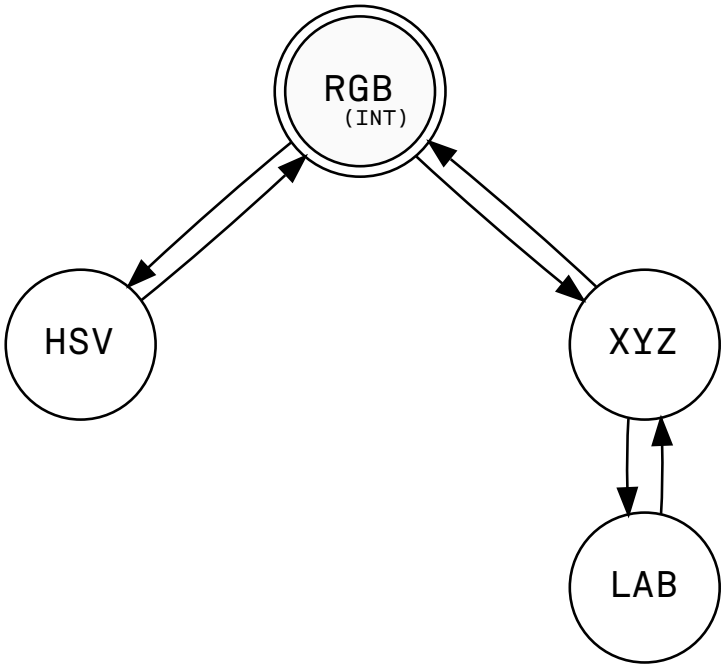


Fig. 5: Supported color spaces and transformations

7

API REFERENCE

Note: Almost all public classes are imported into the first package level on its initialization, which makes kind of a contract on library's API. The exceptions include some abstract superclasses or metaclasses, which generally should not be used outside of the library, but still can be imported directly using a full module path.

ansi	Classes for working with ANSI escape sequences on a lower level.
color	Abstractions for color definitions in three primary modes: 4-bit, 8-bit and 24-bit (`xterm-16`, `xterm-256` and `True Color/RGB`, respectively).
common	
config	Library fine tuning module.
cval	Color preset list:
exception	
filter	Formatters for prettier output and utility classes to avoid writing boilerplate code when dealing with escape sequences.
log	
numfmt	utilnum
renderer	Renderers transform *Style* instances into lower-level abstractions like *SGR sequences*, tmux-compatible directives, HTML markup etc., depending on renderer type.
style	Reusable data classes that control the appearance of the output -- colors (text/background/underline) and attributes (*bold*, *underlined*, *italic*, etc.).
template	
term	A
text	"Front-end" module of the library.

7.1 pytermor.ansi

Classes for working with ANSI escape sequences on a lower level. Can be used for creating a variety of sequences including:

- SGR sequences (text and background coloring, other text formatting and effects);
- CSI sequences (cursor management, selective screen clearing);
- OSC (Operating System Command) sequences (various system commands).

Provides a bunch of ready-to-use sequence makers, as well as core method get_closing_seq() that queries SGR pairs registry and composes "counterpart" sequence for a specified one: every attribute that the latter modifies, will be changed back by the one that's being created, while keeping the other attributes untouched. This method is used by SgrRenderer and is essential for nested style processing, as regular RESET sequence cancels all the formatting applied to the output at the moment it's getting introduced to a terminal emulator, and is near to impossible to use because of that (at least when there is a need to perform partial attribute termination, e.g. for overlapping styles rendering).

Module Attributes

NOOP_SEQ	Special sequence in case one *has to* provide one or another SGR, but does not want any control sequences to be actually included in the output.
ESCAPE_SEQ_REGEX	Regular expression that matches all classes of escape sequences.

Functions

contains_sgr(string, *codes)	Return the first match of *SGR* sequence in `string` with specified `codes` as params, strictly inside a single sequence in specified order, or *None* if nothing was found.
enclose(opening_seq, string)	**param opening_seq**
get_closing_seq(opening_seq)	**param opening_seq**
get_resetter_codes()	
parse(string)	**param string**
seq_from_dict(groupdict)	

Classes

ColorTarget(value)	An enumeration.
ISequence(classifier[, interm, final, abbr])	Abstract ancestor of all escape sequences.
IntCode(value)	Complete or almost complete list of reliably working SGR param integer codes.
SeqIndex()	Registry of static sequences that can be utilized without implementing an extra logic.
SequenceCSI([final, interm, abbr])	Class representing CSI-type ANSI escape sequence.
SequenceFe(classifier, *params[, interm, ...])	C1 set sequences -- a wide range of sequences that includes CSI <SequenceCSI>, OSC <SequenceOSC> and more.
SequenceFp(classifier[, abbr])	Sequence class representing private control functions.
SequenceFs(classifier[, abbr])	Sequences referred by ECMA-48 as "independent control functions".
SequenceNf(classifier, final[, interm, abbr])	Escape sequences mostly used for ANSI/ISO code-switching mechanisms.
SequenceOSC(*params)	OSC-type sequence.
SequenceSGR(*params)	Class representing SGR-type escape sequence with varying amount of parameters.
SequenceST()	String Terminator sequence (ST).
SubtypedParam(value, subtype)	

class pytermor.ansi.**ISequence**(*classifier*, *interm=None*, *final=None*, *abbr='ESC*'*)

Bases: Sized

Abstract ancestor of all escape sequences.

Parameters

- **classifier** (*str*) – Classifier char, see guide.advanced-seq-types.
- **interm** (*str*) – Intermediate chars.
- **final** (*str*) – Final char.
- **abbr** (*str*) – Abbreviation for debug purposes.

class pytermor.ansi.**SequenceNf**(*classifier*, *final*, *interm=None*, *abbr='nF'*)

Bases: *ISequence*

Escape sequences mostly used for ANSI/ISO code-switching mechanisms.

All **nF**-class sequences start with ESC plus ASCII byte from the range 0x20-0x2F (space, !, ", #, $, %, &, ', (,), *, +, ,, -, ., /).

Parameters

- **classifier** (*str*) – Classifier char (0x20-0x2F)
- **final** (*str*) – Final char (0x30-0x7E)
- **interm** (*str*) – intermediate chars (0x20-0x2F)
- **abbr** – Abbreviation for debug purposes.

assemble()

Build up actual byte sequence and return as an ASCII-encoded string.

Return type
str

class pytermor.ansi.**SequenceFp**(*classifier*, *abbr='Fp'*)

Bases: *ISequence*

Sequence class representing private control functions.

All **Fp**-class sequences start with ESC plus ASCII byte in the range 0x30-0x3F (0-9, :, ;, <, =, >, ?).

Parameters

- **classifier** (*str*) – Classifier char (0x30-0x3F)
- **abbr** – Abbreviation for debug purposes.

class pytermor.ansi.**SequenceFs**(*classifier*, *abbr='Fs'*)

Bases: *ISequence*

Sequences referred by ECMA-48 as "independent control functions".

All **Fs**-class sequences start with ESC plus a byte in the range 0x60-0x7E (`, a-z, {, |, }).

Parameters

- **classifier** (*str*) – Classifier char (0x60-0x7E)
- **abbr** – Abbreviation for debug purposes.

class pytermor.ansi.**SequenceFe**(*classifier*, **params*, *interm=None*, *final=None*, *abbr='Fe'*)

Bases: *ISequence*

C1 set sequences – a wide range of sequences that includes CSI <SequenceCSI>, OSC <SequenceOSC> and more.

All **Fe**-class sequences start with ESC plus ASCII byte from 0x40 to 0x5F (@, [, \,], _, ^ and capital letters A-Z).

Parameters

- **classifier** (*str*) – Classifier char (0x40-0x5F)
- **params** (*int | str*) – Parameter chars (0x30-0x3F)
- **interm** (*str*) – Intermediate chars (0x20-0x2F)
- **final** (*str*) – Final char (0x40-0x7E)
- **abbr** – Abbreviation for debug purposes.

class pytermor.ansi.**SequenceST**

Bases: *SequenceFe*

String Terminator sequence (ST). Terminates strings in other control sequences. Encoded as ESC \ (0x1B 0x5C).

class pytermor.ansi.**SequenceOSC**(**params*)

Bases: *SequenceFe*

OSC-type sequence. Starts a control string for the operating system to use. Encoded as ESC], plus params separated by ;. The control string can contain bytes from ranges 0x08-0x0D, 0x20-0x7E and is usually terminated by ST <SequenceST>.

Parameters

params (*int | str*) – Parameter chars (0x30-0x3F)

class pytermor.ansi.**SequenceCSI**(*final=None*, **params*, *interm=None*, *abbr='CSI'*)

Bases: *SequenceFe*

Class representing CSI-type ANSI escape sequence. All subtypes of this sequence start with ESC [.

Sequences of this type are used to control text formatting, change cursor position, erase screen and more.

```
>>> from pytermor import *
>>> make_clear_line().assemble()
'[2K'
```

Parameters

- **final** (*str*) – Final char (0x40-0x7E)
- **params** (*int*) – Parameter chars (0x30-0x3F)
- **interm** (*str*) – Intermediate chars. (0x21/0x3F)
- **abbr** (*str*) – Abbreviation for debug purposes.

class pytermor.ansi.**SequenceSGR**(**params*)

Bases: *SequenceCSI*

Class representing SGR-type escape sequence with varying amount of parameters. SGR sequences allow to change the color of text or/and terminal background (in 3 different color spaces) as well as set decorate text with italic style, underlining, overlining, cross-lining, making it bold or blinking etc.

```
>>> SequenceSGR(IntCode.HI_CYAN, 'underlined', 1)
<SGR[96;4;1m]>
```

To encode into control sequence byte-string invoke assemble() method or cast the instance to *str*, which internally does the same (this actually applies to all children of ISequence):

```
>>> SequenceSGR('blue', 'italic').assemble()
'[34;3m'
>>> str(SequenceSGR('blue', 'italic'))
'[34;3m'
```

The latter also allows fluent usage in f-strings:

```
>>> f'{SeqIndex.RED}should be red{SeqIndex.RESET}'
'[31mshould be red[0m'
```

Note: SequenceSGR with zero params `ESC [m` is interpreted by terminal emulators as `ESC [0m`, which is *hard* reset sequence. The empty-string-sequence is predefined at module level as NOOP_SEQ.

Note: The module doesn't distinguish "single-instruction" sequences from several ones merged together, e.g. `Style(fg='red', bold=True)` produces only one opening SequenceSGR instance:

```
>>> SequenceSGR(IntCode.BOLD, IntCode.RED).assemble()
'[1;31m'
```

...although generally speaking it is two of them (`ESC [1m` and `ESC [31m`). However, the module can automatically match terminating sequences for any form of input SGRs and translate it to specified format.

It is possible to add of one SGR sequence to another, resulting in a new one with merged params:

```
>>> SequenceSGR('blue') + SequenceSGR('italic')
<SGR[34;3m]>
```

Parameters

params (*str | int | SubtypedParam | SequenceSGR*) – Sequence params. Resulting param order is the same as an argument order. Each argument can be specified as:

- *str* – any of IntCode names, case-insensitive;
- *int* – IntCode instance or plain integer;
- *SubtypeParam*
- another SequenceSGR instance (params will be extracted).

property params: List[int | pytermor.ansi.SubtypedParam]

Returns

Sequence params as integers.

class pytermor.ansi.**IntCode**(*value*)

Bases: `IntEnum`

Complete or almost complete list of reliably working SGR param integer codes. Fully interchangeable with plain *int*. Suitable for SequenceSGR default constructor.

Note: IntCode predefined constants are omitted from documentation to avoid useless repeats and save space, as most of the time "higher-level" class SeqIndex will be more appropriate, and on top of that, the constant names are literally the same for SeqIndex and IntCode.

---

classmethod resolve(*name*)

Parameters
name (*str*) –

Return type
IntCode

class pytermor.ansi.**SeqIndex**

Registry of static sequences that can be utilized without implementing an extra logic.

RESET = <SGR[0m]>

Hard reset sequence.

BOLD = <SGR[1m]>

Bold or increased intensity.

DIM = <SGR[2m]>

Faint, decreased intensity.

ITALIC = <SGR[3m]>

Italic *(not widely supported)*.

UNDERLINED = <SGR[4m]>

Underline.

CURLY_UNDERLINED = <SGR[4:3m]>

Curly underline.

BLINK_SLOW = <SGR[5m]>

Set blinking to < 150 cpm.

BLINK_FAST = <SGR[6m]>

Set blinking to 150+ cpm *(not widely supported)*.

INVERSED = <SGR[7m]>

Swap foreground and background colors.

HIDDEN = <SGR[8m]>

Conceal characters *(not widely supported)*.

CROSSLINED = <SGR[9m]>

Strikethrough.

DOUBLE_UNDERLINED = <SGR[21m]>

Double-underline. *On several terminals disables* BOLD *instead*.

FRAMED = <SGR[51m]>

Rectangular border *(not widely supported, to say the least)*.

OVERLINED = <SGR[53m]>

Overline *(not widely supported)*.

BOLD_DIM_OFF = <SGR[22m]>

Disable BOLD and DIM attributes.

Special aspects... It's impossible to reliably disable them on a separate basis.

ITALIC_OFF = <SGR[23m]>

Disable italic.

UNDERLINED_OFF = <SGR[24m]>

Disable underlining.

BLINK_OFF = <SGR[25m]>

Disable blinking.

INVERSED_OFF = <SGR[27m>
Disable inversing.

HIDDEN_OFF = <SGR[28m>
Disable conecaling.

CROSSLINED_OFF = <SGR[29m>
Disable strikethrough.

FRAMED_OFF = <SGR[54m>
Disable border.

OVERLINED_OFF = <SGR[55m>
Disable overlining.

UNDERLINE_COLOR_OFF = <SGR[59m>
Reset underline color.

BLACK = <SGR[30m>
Set text color to 0x000000.

RED = <SGR[31m>
Set text color to 0x800000.

GREEN = <SGR[32m>
Set text color to 0x008000.

YELLOW = <SGR[33m>
Set text color to 0x808000.

BLUE = <SGR[34m>
Set text color to 0x000080.

MAGENTA = <SGR[35m>
Set text color to 0x800080.

CYAN = <SGR[36m>
Set text color to 0x008080.

WHITE = <SGR[37m>
Set text color to 0xc0c0c0.

COLOR_OFF = <SGR[39m>
Reset foreground color.

BG_BLACK = <SGR[40m>
Set background color to 0x000000.

BG_RED = <SGR[41m>
Set background color to 0x800000.

BG_GREEN = <SGR[42m>
Set background color to 0x008000.

BG_YELLOW = <SGR[43m>
Set background color to 0x808000.

BG_BLUE = <SGR[44m>
Set background color to 0x000080.

BG_MAGENTA = <SGR[45m>
Set background color to 0x800080.

BG_CYAN = <SGR[46m]>
Set background color to 0x008080.

BG_WHITE = <SGR[47m]>
Set background color to 0xc0c0c0.

BG_COLOR_OFF = <SGR[49m]>
Reset background color.

GRAY = <SGR[90m]>
Set text color to 0x808080.

HI_RED = <SGR[91m]>
Set text color to 0xff0000.

HI_GREEN = <SGR[92m]>
Set text color to 0x00ff00.

HI_YELLOW = <SGR[93m]>
Set text color to 0xffff00.

HI_BLUE = <SGR[94m]>
Set text color to 0x0000ff.

HI_MAGENTA = <SGR[95m]>
Set text color to 0xff00ff.

HI_CYAN = <SGR[96m]>
Set text color to 0x00ffff.

HI_WHITE = <SGR[97m]>
Set text color to 0xffffff.

BG_GRAY = <SGR[100m]>
Set background color to 0x808080.

BG_HI_RED = <SGR[101m]>
Set background color to 0xff0000.

BG_HI_GREEN = <SGR[102m]>
Set background color to 0x00ff00.

BG_HI_YELLOW = <SGR[103m]>
Set background color to 0xffff00.

BG_HI_BLUE = <SGR[104m]>
Set background color to 0x0000ff.

BG_HI_MAGENTA = <SGR[105m]>
Set background color to 0xff00ff.

BG_HI_CYAN = <SGR[106m]>
Set background color to 0x00ffff.

BG_HI_WHITE = <SGR[107m]>
Set background color to 0xffffff.

class pytermor.ansi.**ColorTarget**(*value*)

Bases: Enum

An enumeration.

pytermor.ansi.**get_closing_seq**(*opening_seq*)

Parameters
opening_seq (SequenceSGR) –

Returns

Return type
SequenceSGR

pytermor.ansi.**enclose**(*opening_seq*, *string*)

Parameters

- **opening_seq** (SequenceSGR) –
- **string** (*str*) –

Returns

Return type
str

pytermor.ansi.**NOOP_SEQ = <SGR/NOP>**

Special sequence in case one *has to* provide one or another SGR, but does not want any control sequences to be actually included in the output.

`NOOP_SEQ.assemble()` returns empty string, `NOOP_SEQ.params` returns empty list:

```
>>> NOOP_SEQ.assemble()
''
>>> NOOP_SEQ.params
[]
```

Important: Casting to *bool* results in **False** for all `NOOP` instances in the library (NOOP_SEQ, NOOP_COLOR and NOOP_STYLE). This is intended.

Can be safely added to regular SequenceSGR from any side, as internally SequenceSGR always makes a new instance with concatenated params from both items, rather than modifies state of either of them:

```
>>> NOOP_SEQ + SequenceSGR(1)
<SGR[1m]>
>>> SequenceSGR(3) + NOOP_SEQ
<SGR[3m]>
```

pytermor.ansi.**ESCAPE_SEQ_REGEX**

Regular expression that matches all classes of escape sequences.

More specifically, it recognizes **nF**, **Fp**, **Fe** and **Fs**[1] classes. Useful for removing the sequences as well as for granular search thanks to named match groups, which include:

escape_byte
first byte of every sequence – `ESC`, or 0x1B.

data
remaining bytes of the sequence (without escape byte) represented as one of the following groups: `nf_class_seq`, `fp_class_seq`, `fe_class_seq` or `fs_class_seq`; each of these splits further to even more specific subgroups:

- `nf_classifier`, `nf_interm` and `nf_final` as parts of **nF**-class sequences,
- `fp_classifier` for **Fp**-class sequences,

[1] ECMA-35 specification

- `st_classifier`, `osc_classifier`, `osc_param`, `csi_classifier`, `csi_interm`, `csi_param`, `csi_final`, `fe_classifier`, `fe_param`, `fe_interm` and `fe_final` for **Fe**-class generic sequences and subtypes (including *SGRs*),
- `fs_classifier` for **Fs**-class sequences.

pytermor.ansi.**contains_sgr**(*string*, **codes*)

Return the first match of *SGR* sequence in `string` with specified `codes` as params, strictly inside a single sequence in specified order, or *None* if nothing was found.

The match object has one group (or, technically, two):

- Group #0: the whole matched SGR sequence;
- Group #1: the requested params bytes only.

Example regex used for searching: `x1b[(?:|[d;]*;)(48;5)(?:|;[d;]*)m`.

```
>>> contains_sgr(make_color_256(128).assemble(), 38)
<re.Match object; span=(0, 11), match='[38;5;128m'>
>>> contains_sgr(make_color_256(84, ColorTarget.BG).assemble(), 48, 5)
<re.Match object; span=(0, 10), match='[48;5;84m'>
```

Parameters

- **string** (*str*) – String to search the SGR in.
- **codes** (*int*) – Integer SGR codes to find.

Return type
re.Match | None

pytermor.ansi.**parse**(*string*)

Parameters
string (*str*) –

Return type
Iterable[*pytermor.ansi.ISequence* | str]

7.2 pytermor.color

Abstractions for color definitions in three primary modes: 4-bit, 8-bit and 24-bit (`xterm-16`, `xterm-256` and `True Color/RGB`, respectively). Provides a global registry for color searching by names and codes, as well as approximation algorithms, which are used for output devices with limited advanced color modes support. Renderers do that automatically and transparently for the developer, but the manual control over this process is also an option.

Supports 4 different color spaces: RGB, HSV, XYZ and LAB, and also provides methods to covert colors from any space to any other.

Functions

approximate(value[, color_type, max_results])	Search for nearest to `value` colors of specified `color_type` and return the first `max_results` of them.
find_closest(value[, color_type])	Search and return nearest to `value` instance of specified `color_type`.
resolve_color(subject[, color_type, ...])	Suggested usage is to transform the user input in a free form in an attempt to find any matching color.

Classes

ApxResult(color, distance)	Approximation result.
Color16(*args, **kwargs)	Variant of a `Color` operating within the most basic color set -- **xterm-16**.
Color256(*args, **kwargs)	Variant of a `Color` operating within relatively modern **xterm-256** indexed color table.
ColorRGB(*args, **kwargs)	Variant of a `Color` operating within RGB color space.
DefaultColor()	Special `Color` instance rendering to SGR sequence telling the terminal to reset fg or bg color; same for TmuxRenderer. Useful when you inherit some *Style* with fg or bg color that you don't need, but at the same time you don't actually want to set up any color whatsoever::.
DynamicColor(*args, **kwargs)	Color that returns different values depending on internal class-level state that can be altered globally for all instances of a concrete implementation.
HSV(hue, saturation, value)	Initially HSV is a transformation of RGB color space; color is stored as 3 floats representing Hue channel, Saturation channel and Value channel correspondingly.
`IColorValue`()	
LAB(lum, a, b)	Color value in a *uniform* color space, CIELAB, which expresses color as three values: L* for perceptual lightness and a* and b* for the four unique colors of human vision: red, green, blue and yellow.
NoopColor()	Special `Color` class always rendering into empty string.
RGB(value)	Color value stored internally as an 24-bit integer.
`RealColor`(value)	
RenderColor()	Abstract superclass for other `Colors`.
ResolvableColor(*args, **kwargs)	Mixin for other `Colors`.
XYZ(x, y, z)	Color in XYZ space is represented by three floats: Y is the luminance, Z is quasi-equal to blue (of CIE RGB), and X is a mix of the three CIE RGB curves chosen to be nonnegative.

class pytermor.color.**RGB**(*value*)

Bases: `IColorValue`

Color value stored internally as an 24-bit integer. Base for more complex color classes.

classmethod diff(*c1*, *c2*)

RGB euclidean distance.

Return type
float

classmethod from_channels(*red*, *green*, *blue*)

Parameters

- **red** –
- **green** –
- **blue** –

Returns

Return type

classmethod from_ratios(*rr*, *gr*, *br*)

d :param rr: :param gr: :param br:

Return type
RGB

property red: int

Red channel value [0;255]

property green: int

Green channel value [0;255]

property blue: int

Blue channel value [0;255]

property int: int

Color value in RGB space (24-bit integer within [0; 0xFFFFFF] range)

property rgb: *RGB*

Color value in RGB space (3 × 8-bit ints)

property hsv: *HSV*

Color value in HSV space (three floats)

property xyz: *XYZ*

Color value in XYZ space (three floats)

property lab: *LAB*

Color value in LAB space (three floats)

class pytermor.color.**HSV**(*hue*, *saturation*, *value*)

Bases: IColorValue

Initially HSV is a transformation of RGB color space; color is stored as 3 floats representing Hue channel, Saturation channel and Value channel correspondingly. Supports direct (fast) transformation to RGB and indirect (=slow) to all other spaces through using more than one conversion with HSV → RGB being the first one.

classmethod diff(*c1*, *c2*)

HSV euclidean distance.

Return type
float

property hue: float

Hue channel value [0;360]

property saturation: float

Saturation channel value [0;1]

property value: float

Value channel value [0;1]

property int: int

Color value in RGB space (24-bit integer within [0; 0xFFFFFF] range)

property rgb: *RGB*

Color value in RGB space (3 × 8-bit ints)

property hsv: *HSV*

Color value in HSV space (three floats)

property xyz: *XYZ*

Color value in XYZ space (three floats)

property lab: *LAB*

Color value in LAB space (three floats)

class pytermor.color.**XYZ**(*x*, *y*, *z*)

Bases: IColorValue

Color in XYZ space is represented by three floats: Y is the luminance, Z is quasi-equal to blue (of CIE RGB), and X is a mix of the three CIE RGB curves chosen to be nonnegative. CIE 1931 XYZ color space was one of the first attempts to produce a color space based on measurements of human color perception. Setting Y as luminance has the useful result that for any given Y value, the XZ plane will contain all possible chromaticities at that luminance.

Note: x and z values can be above 100.

---

classmethod diff(*c1*, *c2*)

Note: This one is written on the analogy of other diffs, therefore it can be actually a little bit incorrect or outright wrong.

Return type
float

property x: float

X channel value [0;100)

property y: float

Luminance [0;100]

property z: float

Quasi-equal to blue [0;100)

property int: int

Color value in RGB space (24-bit integer within [0; 0xFFFFFF] range)

property rgb: *RGB*

Color value in RGB space (3 × 8-bit ints)

property hsv: *HSV*

Color value in HSV space (three floats)

property xyz: *XYZ*

Color value in XYZ space (three floats)

property lab: *LAB*

Color value in LAB space (three floats)

class pytermor.color.**LAB**(*lum*, *a*, *b*)

Bases: IColorValue

Color value in a *uniform* color space, CIELAB, which expresses color as three values: L* for perceptual lightness and a* and b* for the four unique colors of human vision: red, green, blue and yellow. CIELAB was intended as a perceptually uniform space, where a given numerical change corresponds to a similar perceived change in color. Like the CIEXYZ space it derives from, CIELAB color space is a device-independent, "standard observer" model.

classmethod diff(*c1*, *c2*)

CIE76 E* color difference.

Return type
float

property lum: float

Luminance [0;100]

property a: float

Green–magenta axis, [-100;100] in general, but can be less/more

property b: float

Blue–yellow axis, [-100;100] in general, but can be less/more

property int: int

Color value in RGB space (24-bit integer within [0; 0xFFFFFF] range)

property rgb: *RGB*

Color value in RGB space (3 × 8-bit ints)

property hsv: *HSV*

Color value in HSV space (three floats)

property xyz: *XYZ*

Color value in XYZ space (three floats)

property lab: *LAB*

Color value in LAB space (three floats)

class pytermor.color.**RenderColor**

Abstract superclass for other Colors. Provides interfaces for transforming RGB values to SGRs for different terminal modes.

abstract to_sgr(*target=ColorTarget.FG*, *upper_bound=None*)

Make an SGR sequence<SequenceSGR> out of Color. Used by SgrRenderer.

Parameters

- **target** (ColorTarget) – Sequence context (FG, BG, UNDERLINE).

- **upper_bound** (*Optional[Type[Color]]*) – Required result `Color` type upper boundary, i.e., the maximum acceptable color class, which will be the basis for SGR being made. See Color256.to_sgr() for the details.

Return type
SequenceSGR

abstract to_tmux(*target=ColorTarget.FG*)

Make a tmux markup directive, which will change the output color to this color's value (after tmux processes and prints it). Used by TmuxRenderer.

Parameters
target (ColorTarget) – Sequence context (FG, BG, UNDERLINE).

Return type
str

class pytermor.color.**ResolvableColor**(**args*, ***kwargs*)

Bases: `Generic[_RCT]`

Mixin for other `Colors`. Implements color search by name.

Return type
_RCT

classmethod names()

All registried colors' names of this type.

Return type
Iterable[*Tuple*[str]]

classmethod find_by_name(*name*)

Case-insensitive search through registry contents.

See also:

resolve_color() for the details

Parameters
name (*str*) – Name to search for.

Return type
_RCT

classmethod find_closest(*value*)

Search and return color instance nearest to `value`.

See also:

color.find_closest() for the details

Parameters
value (*pytermor.color.IColorValue | int*) – Target color/color value.

Return type
_RCT

classmethod approximate(*value*, *max_results=1*)

Search for the colors nearest to `value` and return the first `max_results`.

See also:

color.approximate() for the details

Parameters

- **value** (*pytermor.color.IColorValue | int*) – Target color/color value.
- **max_results** (*int*) – Result limit.

Return type
List[ApxResult[*_RCT*]]

property name: str | None

Color name, e.g. "navy-blue".

class pytermor.color.**ApxResult**(*color*, *distance*)

Bases: Generic[_RCT]

Approximation result.

color: _RCT

Found Color instance.

distance: float

Color difference between this instance and the approximation target.

class pytermor.color.**Color16**(**args*, ***kwargs*)

Bases: RealColor, *RenderColor*, *ResolvableColor*[*Color16*]

Variant of a Color operating within the most basic color set – **xterm-16**. Represents basic color-setting SGRs with primary codes 30-37, 40-47, 90-97 and 100-107 (see guide.ansi-presets.color16).

Parameters

- **value** (*int | IColorValue*) – Color value as 24-bit integer in RGB space, or any instance implementing color value interface (e.g. HSV).
- **code_fg** (*int*) – Int code for a foreground color setup, e.g. 30.
- **code_bg** (*int*) – Int code for a background color setup. e.g. 40.
- **name** (*str*) – Name of the color, e.g. "red".
- **register** (*bool*) – If *True*, add color to registry for resolving by name and approximation.
- **aliases** (*list[str]*) – Alternative color names (used in resolve_color()).

property code_fg: int

Int code for a foreground color setup, e.g. 30.

property code_bg: int

Int code for a background color setup. e.g. 40.

property color256_equiv: *pytermor.color.Color256* | None

...

classmethod get_by_code(*code*)

Get a Color16 instance with specified code. Only *foreground* (=text) colors are indexed, therefore it is not possible to look up for a Color16 with given background color (on second thought, it *is* actually possible using find_closest()).

Parameters
code (*int*) – Foreground integer code to look up for (see guide.ansi-presets.color16).

Raises
LookupError – If no color with specified code is found.

Return type
Color16

`to_sgr`(*target=ColorTarget.FG*, *upper_bound=None*)

Make an SGR sequence<SequenceSGR> out of `Color`. Used by SgrRenderer.

Parameters

- **target** (ColorTarget) – Sequence context (FG, BG, UNDERLINE).
- **upper_bound** (*Optional[Type[Color]]*) – Required result `Color` type upper boundary, i.e., the maximum acceptable color class, which will be the basis for SGR being made. See Color256.to_sgr() for the details.

Return type
SequenceSGR

`to_tmux`(*target=ColorTarget.FG*)

Make a tmux markup directive, which will change the output color to this color's value (after tmux processes and prints it). Used by TmuxRenderer.

Parameters
target (ColorTarget) – Sequence context (FG, BG, UNDERLINE).

Return type
str

`classmethod approximate`(*value*, *max_results=1*)

Search for the colors nearest to `value` and return the first `max_results`.

See also:

color.approximate() for the details

Parameters

- **value** (*pytermor.color.IColorValue | int*) – Target color/color value.
- **max_results** (*int*) – Result limit.

Return type
List[ApxResult[*_RCT*]]

`classmethod find_by_name`(*name*)

Case-insensitive search through registry contents.

See also:

resolve_color() for the details

Parameters
name (*str*) – Name to search for.

Return type
_RCT

`classmethod find_closest`(*value*)

Search and return color instance nearest to `value`.

See also:

color.find_closest() for the details

Parameters
value (*pytermor.color.IColorValue | int*) – Target color/color value.

Return type
_RCT

format_value(*prefix='0x'*)

Format color value as "0xRRGGBB".

Return type
str

property hsv: *HSV*

Color value in HSV space (three floats)

property int: **int**

Color value in RGB space (24-bit integer within [0; 0xFFFFFF] range)

property lab: *LAB*

Color value in LAB space (three floats)

property name: **str | None**

Color name, e.g. "navy-blue".

classmethod names()

All registried colors' names of this type.

Return type
Iterable[*Tuple*[str]]

property rgb: *RGB*

Color value in RGB space (3 × 8-bit ints)

property xyz: *XYZ*

Color value in XYZ space (three floats)

class pytermor.color.**Color256**(**args*, ***kwargs*)

Bases: RealColor, *RenderColor*, *ResolvableColor*[*Color256*]

Variant of a Color operating within relatively modern **xterm-256** indexed color table. Represents SGR complex codes 38;5;* and 48;5;* (see guide.ansi-presets.color256).

Parameters

- **value** (*int | IColorValue*) – Color value as 24-bit integer in RGB space, or any instance implementing color value interface (e.g. HSV).
- **code** (*int*) – Int code for a color setup, e.g. 52.
- **name** (*str*) – Name of the color, e.g. "dark-red".
- **register** (*bool*) – If *True*, add color to registry for resolving by name.
- **aliases** (*t.List[str]*) – Alternative color names (used in resolve_color()).
- **color16_equiv** (Color16) – Color16 counterpart (applies only to codes 0-15).

to_sgr(*target=ColorTarget.FG*, *upper_bound=None*)

Make an SGR sequence<SequenceSGR> out of Color. Used by SgrRenderer.

Each Color type represents one SGR type in the context of colors. For example, if upper_bound is set to Color16, the resulting SGR will always be one of 16-color index table, even if the original color was of different type – it will be approximated just before the SGR assembling.

The reason for this is the necessity to provide a similar look for all users with different terminal settings/capabilities. When the library sees that user's output device supports 256 colors only, it cannot assemble True Color SGRs, because they will be ignored (if we are lucky), or displayed in a glitchy way, or mess up the output completely. The good news is that the process is automatic and in most cases the library will manage the transformations by itself. If it's not the case, the developer can correct the behaviour by overriding the renderers' output mode. See SgrRenderer and OutputMode docs.

Parameters

- **target** (ColorTarget) –
- **upper_bound** (*Optional[Type[Color]]*) – Required result Color type upper boundary, i.e., the maximum acceptable color class, which will be the basis for SGR being made.

Return type
SequenceSGR

to_tmux(*target=ColorTarget.FG*)

Make a tmux markup directive, which will change the output color to this color's value (after tmux processes and prints it). Used by TmuxRenderer.

Parameters
target (ColorTarget) – Sequence context (FG, BG, UNDERLINE).

Return type
str

property code: int

Int code for a color setup, e.g. 52.

classmethod get_by_code(*code*)

Get a Color256 instance with specified code (=position in the index).

Parameters
code (*int*) – Color code to look up for (see guide.ansi-presets.color256).

Raises
LookupError – If no color with specified code is found.

Return type
Color256

classmethod approximate(*value*, *max_results=1*)

Search for the colors nearest to value and return the first max_results.

See also:

color.approximate() for the details

Parameters
- **value** (*pytermor.color.IColorValue | int*) – Target color/color value.
- **max_results** (*int*) – Result limit.

Return type
List[ApxResult[*_RCT*]]

classmethod find_by_name(*name*)

Case-insensitive search through registry contents.

See also:

resolve_color() for the details

Parameters
name (*str*) – Name to search for.

Return type
_RCT

classmethod find_closest(*value*)

Search and return color instance nearest to `value`.

See also:

color.find_closest() for the details

Parameters
value (*pytermor.color.IColorValue | int*) – Target color/color value.

Return type
_RCT

format_value(*prefix='0x'*)

Format color value as "0xRRGGBB".

Return type
str

property hsv: *HSV*

Color value in HSV space (three floats)

property int: **int**

Color value in RGB space (24-bit integer within [0; 0xFFFFFF] range)

property lab: *LAB*

Color value in LAB space (three floats)

property name: **str | None**

Color name, e.g. "navy-blue".

classmethod names()

All registried colors' names of this type.

Return type
Iterable[*Tuple*[str]]

property rgb: *RGB*

Color value in RGB space (3 × 8-bit ints)

property xyz: *XYZ*

Color value in XYZ space (three floats)

class pytermor.color.**ColorRGB**(**args*, ***kwargs*)

Bases: `RealColor`, *RenderColor*, *ResolvableColor*[*ColorRGB*]

Variant of a `Color` operating within RGB color space. Presets include es7s named colors <guide.es7s-colors>, a unique collection of colors compiled from several known sources after careful selection. However, it's not limited to aforementioned color list and can be easily extended.

Parameters

- **value** (*int|IColorValue*) – Color value as 24-bit integer in RGB space (e.g. 0x73a9c2), or any instance implementing color value interface (e.g. HSV).
- **name** (*str*) – Name of the color, e.g. "moonstone-blue".
- **register** (*bool*) – If *True*, add color to registry for resolving by name.
- **aliases** (*t.List[str]*) – Alternative color names (used in resolve_color()).
- **variation_map** (*t.Dict[int, str]*) – Mapping {*int*: *str*}, where keys are hex values, and values are variation names.

to_sgr(*target=ColorTarget.FG*, *upper_bound=None*)

Make an SGR sequence<SequenceSGR> out of `Color`. Used by SgrRenderer.

Parameters

- **target** (ColorTarget) – Sequence context (FG, BG, UNDERLINE).
- **upper_bound** (*Optional[Type[Color]]*) – Required result `Color` type upper boundary, i.e., the maximum acceptable color class, which will be the basis for SGR being made. See Color256.to_sgr() for the details.

Return type
SequenceSGR

to_tmux(*target=ColorTarget.FG*)

Make a tmux markup directive, which will change the output color to this color's value (after tmux processes and prints it). Used by TmuxRenderer.

Parameters
target (ColorTarget) – Sequence context (FG, BG, UNDERLINE).

Return type
str

property base: Optional[_RCT]

Parent color for color variations. Empty for regular colors.

property variations: Dict[str, _RCT]

List of color variations. *Variation* of a color is a similar color with almost the same name, but with differing suffix. The main idea of variations is to provide a basis for fuzzy searching, which will return several results for one query; i.e., when the query matches a color with variations, the whole color family can be considered a match, which should increase searching speed.

classmethod approximate(*value*, *max_results=1*)

Search for the colors nearest to `value` and return the first `max_results`.

See also:

color.approximate() for the details

Parameters

- **value** (*pytermor.color.IColorValue | int*) – Target color/color value.
- **max_results** (*int*) – Result limit.

Return type
List[ApxResult[*_RCT*]]

classmethod find_by_name(*name*)

Case-insensitive search through registry contents.

See also:

resolve_color() for the details

Parameters
name (*str*) – Name to search for.

Return type
_RCT

classmethod find_closest(*value*)

Search and return color instance nearest to `value`.

See also:

color.find_closest() for the details

Parameters
value (*pytermor.color.IColorValue | int*) – Target color/color value.

Return type
_RCT

format_value(*prefix='0x'*)

Format color value as "0xRRGGBB".

Return type
str

property hsv: *HSV*

Color value in HSV space (three floats)

property int: **int**

Color value in RGB space (24-bit integer within [0; 0xFFFFFF] range)

property lab: *LAB*

Color value in LAB space (three floats)

property name: **str | None**

Color name, e.g. "navy-blue".

classmethod names()

All registried colors' names of this type.

Return type
Iterable[*Tuple*[str]]

property rgb: *RGB*

Color value in RGB space (3 × 8-bit ints)

property xyz: *XYZ*

Color value in XYZ space (three floats)

class pytermor.color.**NoopColor**

Bases: *RenderColor*

Special `Color` class always rendering into empty string.

Important: Casting to *bool* results in **False** for all `NOOP` instances in the library (NOOP_SEQ, NOOP_COLOR and NOOP_STYLE). This is intended.

---

to_sgr(*target=ColorTarget.FG*, *upper_bound=None*)

Make an SGR sequence<SequenceSGR> out of `Color`. Used by SgrRenderer.

Parameters

- **target** (ColorTarget) – Sequence context (FG, BG, UNDERLINE).
- **upper_bound** (*Optional[Type[Color]]*) – Required result `Color` type upper boundary, i.e., the maximum acceptable color class, which will be the basis for SGR being made. See Color256.to_sgr() for the details.

Return type
SequenceSGR

to_tmux(*target=ColorTarget.FG*)

Make a tmux markup directive, which will change the output color to this color's value (after tmux processes and prints it). Used by TmuxRenderer.

Parameters
target (ColorTarget) – Sequence context (FG, BG, UNDERLINE).

Return type
str

class pytermor.color.**DefaultColor**

Bases: *RenderColor*

Special Color instance rendering to SGR sequence telling the terminal to reset fg or bg color; same for TmuxRenderer. Useful when you inherit some *Style* with fg or bg color that you don't need, but at the same time you don't actually want to set up any color whatsoever:

```
>>> from pytermor import *
>>> DEFAULT_COLOR.to_sgr(target=ColorTarget.BG)
<SGR[49m]>
```

NOOP_COLOR is treated like a placeholder for parent's attribute value and doesn't change the result:

```
>>> from pytermor import SgrRenderer, render
>>> sgr_renderer = SgrRenderer(OutputMode.XTERM_16)
>>> render("MISMATCH", Style(Styles.INCONSISTENCY, fg=NOOP_COLOR), sgr_renderer)
'[93;101mMISMATCH[39;49m'
```

While DEFAULT_COLOR is actually resetting the color to default (terminal) value:

```
>>> render("MISMATCH", Style(Styles.INCONSISTENCY, fg=DEFAULT_COLOR), sgr_
↪renderer)
'[39;101mMISMATCH[49m'
```

to_sgr(*target=ColorTarget.FG*, *upper_bound=None*)

Make an SGR sequence<SequenceSGR> out of Color. Used by SgrRenderer.

Parameters

- **target** (ColorTarget) – Sequence context (FG, BG, UNDERLINE).
- **upper_bound** (*Optional[Type[Color]]*) – Required result Color type upper boundary, i.e., the maximum acceptable color class, which will be the basis for SGR being made. See Color256.to_sgr() for the details.

Return type
SequenceSGR

to_tmux(*target=ColorTarget.FG*)

Make a tmux markup directive, which will change the output color to this color's value (after tmux processes and prints it). Used by TmuxRenderer.

Parameters
target (ColorTarget) – Sequence context (FG, BG, UNDERLINE).

Return type
str

class pytermor.color.**DynamicColor**(**args*, ***kwargs*)

Bases: *RenderColor*, Generic[_T]

Color that returns different values depending on internal class-level state that can be altered globally for all instances of a concrete implementation. Supposed usage is to make a subclass of DynamicColor and define state type, which will be shared between all instances of a new class. Also concrete implementation

of update() method is required, which should contain logic for transforming some external parameters into the state. State can be of any type, from plain RGB value to complex dictionaries or custom classes.

There is also an extractor parameter, which is not shared between instances of same subclass, rather being an instance attribute. This parameter represents the logic of transforming one shared state into several different colors, which therefore can be used as is, or be included as a fg/bg attributes of *Style* instances.

Full usage example can be found at guide.dynamic-deferred-colors docs page.

Parameters

extractor – Concrete implementation of "state" -> "color" transformation logic. Can be a callable, which will be invoked with a state variable as a first argument, or can be a string, in which case it will be used to extract the color value from the instance itself, with this string as an attribute name, or it can be *None*, in which case it implies that state variable is instance of Color or it descendant and it can be returned on extraction without transformation, as is.

_DEFERRED: ClassVar[bool] = False

Class variable responsible for enabling deferred mode. In this mode there is a possibility to delay an initialization of the state of a concrete class and to create all dependant entities regardless. When state is still uninitialized, the return color will be NOOP_COLOR, which automatically updates to an actual color after state creation. See guide.dynamic-deferred-colors for the details.

classmethod update(***kwargs*)

Set new internal state for all instances of this class.

to_sgr(*target=ColorTarget.FG*, *upper_bound=None*)

Make an SGR sequence<SequenceSGR> out of `Color`. Used by SgrRenderer.

Parameters

- **target** (ColorTarget) – Sequence context (FG, BG, UNDERLINE).
- **upper_bound** (*Optional[Type[Color]]*) – Required result `Color` type upper boundary, i.e., the maximum acceptable color class, which will be the basis for SGR being made. See Color256.to_sgr() for the details.

Return type

SequenceSGR

to_tmux(*target=ColorTarget.FG*)

Make a tmux markup directive, which will change the output color to this color's value (after tmux processes and prints it). Used by TmuxRenderer.

Parameters

target (ColorTarget) – Sequence context (FG, BG, UNDERLINE).

Return type

str

pytermor.color.**resolve_color**(*subject*, *color_type=None*, *approx_cache=True*)

Suggested usage is to transform the user input in a free form in an attempt to find any matching color. The method operates in three different modes depending on arguments: resolving by name, resolving by value and instantiating.

Resolving by name: If `subject` is a *str* starting with any character except #, case-insensitive search through the registry of `color_type` colors is performed. In this mode the algorithm looks for the instance which has all the words from `subject` as parts of its name (the order must be the same). Color names are stored in registries as sets of tokens, which allows to use any form of input and get the correct result regardless. The only requirement is to separate the words in any matter (see the example below), so that they could be split to tokens which will be matched with the registry keys.

If `color_type` is omitted, all the registries will be requested in this order: [Color16, Color256, ColorRGB]. Should any registry find a full match, the resolving is stopped and the result is returned.

```
>>> resolve_color('deep-sky-blue-7')
<Color256[x23(#005f5f deep-sky-blue-7)]>
>>> resolve_color('DEEP SKY BLUE 7')
<Color256[x23(#005f5f deep-sky-blue-7)]>
>>> resolve_color('DeepSkyBlue7')
<Color256[x23(#005f5f deep-sky-blue-7)]>
```

```
>>> resolve_color('deepskyblue7')
Traceback (most recent call last):
LookupError: Color 'deepskyblue7' was not found in any registry
```

Resolving by value or **instantiating**: if `subject` is specified as:

1) *int* in [0x000000; 0xffffff] range, or
2) *str* in full hexadecimal form: "#RRGGBB", or
3) *str* in short hexadecimal form: "#RGB",

and `color_type` is **present** , the result will be the best `subject` approximation to corresponding color index. Note that this value is expected to differ from the requested one (and sometimes differs a lot). If `color_type` is **missing**, no searching is performed; instead a new nameless ColorRGB is instantiated and returned.

Note: The instance created this way is an "unbound" color, i.e. it does not end up in a registry or an index bound to its type, thus the resolver and approximation algorithms are unaware of its existence. The rationale for this is to keep the registries clean and stateless to ensure that the same input always resolves to the same output.

```
>>> resolve_color("#333")
<ColorRGB[#333333]>
>>> resolve_color(0xfafef0)
<ColorRGB[#fafef0]>
```

Parameters

- **subject** (*str|int*) – `Color` name or hex value to search for. See CDT.
- **color_type** (*Optional[Type[_RCT]]*) – Target color type (Color16, Color256 or ColorRGB).
- **approx_cache** – Use the approximation cache for **resolving by value** mode or ignore it. For the details see find_closest and approximate which are actually invoked by this method under the hood.

Raises

LookupError – If nothing was found in either of registries.

Returns

`Color` instance with specified name or value.

Return type

_RCT

`pytermor.color.`**find_closest**(*value*, *color_type=None*)

Search and return nearest to `value` instance of specified `color_type`. If color_type is omitted, search for the closest Color256 element.

Note: Distance between two colors is calculated using CIE76 E* color difference formula in LAB color space. This method is considered to be an acceptable tradeoff between sRGB euclidean distance, which doesn't account for differences in human color perception, and CIE94/CIEDE2000, which are more complex and in general excessive for this task.

Method is useful for finding applicable color alternatives if user's terminal is incapable of operating in more advanced mode. Usually it is done by the library automatically and transparently for both the developer and the end-user.

Important: This method caches the results, i.e., the same search query will from then onward result in the same return value without the necessity of iterating through the color index. If that's not applicable, use similar method approximate(), which is unaware of caching mechanism altogether.

Parameters

- **value** (*pytermor.color.IColorValue | int*) – Target color/color value.
- **color_type** (*Optional[Type[_RCT]]*) – Target color type (Color16, Color256 or ColorRGB).

Returns
Nearest to `value` color instance of specified type.

Return type
_RCT

pytermor.color.**approximate**(*value*, *color_type=None*, *max_results=1*)

Search for nearest to `value` colors of specified `color_type` and return the first `max_results` of them. If color_type is omitted, search for the closest Color256 instances. This method is similar to the find_closest(), although they differ in some aspects:

- approximate() can return more than one result;
- approximate() returns not just a `Color` instance(s), but also a number equal to squared distance to the target color for each of them;
- find_closest() caches the results, while approximate() ignores the cache completely.

Parameters

- **value** (*pytermor.color.IColorValue | int*) – Target color/color value.
- **color_type** (*Optional[Type[_RCT]]*) – Target color type (Color16, Color256 or ColorRGB).
- **max_results** (*int*) – Return no more than `max_results` items.

Returns
Pairs of closest `Color` instance(s) found with their distances to the target color, sorted by distance descending, i.e., element at index 0 is the closest color found, paired with its distance to the target; element with index 1 is second-closest color (if any) and corresponding distance value, etc.

Return type
List[ApxResult[*_RCT*]]

7.3 pytermor.common

Module Attributes

CDT	CDT (Color descriptor type) represents a RGB color value.
FT	FT (Format type) is a style descriptor.
RT	RT (Renderable type) includes regular *str*s as well as IRenderable implementations.
filterf	Shortcut for filtering out falsy values from sequences
filtern	Shortcut for filtering out Nones from sequences

Functions

but(cls, inp)	Return all elements from inp *except* instances of cls.
char_range(start, stop)	Yields all the characters from range of [c1; c2], inclusive (end character c2 is also present, in contrast with classic range()).
chunk(items, size)	Split item list into chunks of size `size` and return these
cut(s, max_len[, align, overflow])	**param s**
filterfv(mapping)	Shortcut for filtering out falsy values from mappings
filternv(mapping)	Shortcut for filtering out None values from mappings
fit(s, max_len[, align, overflow, fill])	**param s**
flatten(items[, level_limit])	Unpack a list consisting of any amount of nested lists to 1d-array, or flat list, eliminating all the nesting.
flatten1(items)	Take a list of nested lists and unpack all nested elements one level up.
get_qname(obj)	Convenient method for getting a class name for the instances as well as for the classes themselves, in case where a variable in question can be both.
get_subclasses(target)	Traverse the inheritance tree and return a flat list of all descendants of cls (full hierarchy).
`isiterable`(arg)	
only(cls, inp)	Return all elements from inp that *are* instances of cls
others(cls, inp)	Return all elements from inp *except* instances of cls and its children classes.
ours(cls, inp)	Return all elements from inp that *are* instances of cls or its children classes.
pad(n)	Convenient method to use instead of `"".ljust(n)`.
padv(n)	Convenient method to use instead of `"\n" * n`.

Classes

Align(value)	Align type.
ExtendedEnum(value)	Standard Enum with a few additional methods on top.

pytermor.common.**CDT**

CDT represents a RGB color value. Primary handler is resolve_color(). Valid values include:

- *str* with a color name in any form distinguishable by the color resolver; the color lists can be found at: guide.ansi-presets and guide.es7s-colors;
- *str* starting with a "#" and consisting of 6 more hexadecimal characters, case insensitive (RGB regular form), e.g. "#0b0cca";
- *str* starting with a "#" and consisting of 3 more hexadecimal characters, case insensitive (RGB short form), e.g. "#666";
- *int* in a [0; 0xffffff] range.

alias of TypeVar('CDT', int, str)

pytermor.common.**FT**

FT is a style descriptor. Used as a shortcut precursor for actual styles. Primary handler is make_style().

alias of TypeVar('FT', int, str, IColorValue, Style, None)

pytermor.common.**RT**

RT includes regular *str*s as well as IRenderable implementations.

alias of TypeVar('RT', str, IRenderable)

class pytermor.common.**ExtendedEnum**(*value*)

Bases: Enum

Standard Enum with a few additional methods on top.

classmethod list()

Return all enum values as list.

Example
[1, 10]

Return type
List[_T]

classmethod dict()

Return mapping of all enum keys to corresponding enum values.

Example
{<ExampleEnum.VAL1: 1>: 1, <ExampleEnum.VAL2: 10>: 10}

Return type
Dict[str, _T]

class pytermor.common.**Align**(*value*)

Bases: str, *ExtendedEnum*

Align type.

pytermor.common.**pad**(*n*)

Convenient method to use instead of "".ljust(n).

Return type
str

pytermor.common.**padv**(*n*)

Convenient method to use instead of "\n" * n.

Return type
str

pytermor.common.**cut**(*s*, *max_len*, *align=Align.LEFT*, *overflow=''*)

Parameters

- **s** (*str*) –
- **max_len** (*int*) –
- **align** (pytermor.common.Align | *str*) –
- **overflow** –

Return type
str

pytermor.common.**fit**(*s*, *max_len*, *align=Align.LEFT*, *overflow=''*, *fill=' '*)

Parameters

- **s** (*str*) –
- **max_len** (*int*) –
- **align** (pytermor.common.Align | *str*) –
- **overflow** (*str*) –
- **fill** (*str*) –

Return type
str

pytermor.common.**get_qname**(*obj*)

Convenient method for getting a class name for the instances as well as for the classes themselves, in case where a variable in question can be both.

```
>>> get_qname("aaa")
'str'
>>> get_qname(ExtendedEnum)
'<ExtendedEnum>'
```

Return type
str

pytermor.common.**only**(*cls*, *inp*)

Return all elements from inp that *are* instances of cls

Return type
List[_T]

pytermor.common.**but**(*cls*, *inp*)

Return all elements from inp *except* instances of cls.

Return type
List[_T]

pytermor.common.**ours**(*cls*, *inp*)

Return all elements from inp that *are* instances of cls or its children classes.

Return type
List[_T]

pytermor.common.**others**(*cls*, *inp*)

Return all elements from inp *except* instances of cls and its children classes.

Return type
List[*_T*]

pytermor.common.**chunk**(*items*, *size*)

Split item list into chunks of size `size` and return these chunks as *tuples*.

```
>>> print(*chunk(range(10), 3), sep='''
```

Block quote ends without a blank line; unexpected unindent.

‘’’)
(0, 1, 2) (3, 4, 5) (6, 7, 8) (9,)

param items
Input elements.

param size
Chunk size.

Return type
Iterator[*Tuple*[*_T*, ...]]

pytermor.common.**get_subclasses**(*target*)

Traverse the inheritance tree and return a flat list of all descendants of cls (full hierarchy).

```
>>> from pytermor import SequenceCSI, Color16
>>> get_subclasses(SequenceCSI())
[<class 'pytermor.ansi.SequenceSGR'>, <class 'pytermor.ansi._NoOpSequenceSGR'>]
```

```
>>> get_subclasses(Color16)
[]
```

Return type
Iterable[*Type*[*_T*]]

pytermor.common.**flatten1**(*items*)

Take a list of nested lists and unpack all nested elements one level up.

```
>>> flatten1([1, 2, [3, 4], [[5, 6]]])
[1, 2, 3, 4, [5, 6]]
```

Return type
List[*_T*]

pytermor.common.**flatten**(*items*, *level_limit=None*)

Unpack a list consisting of any amount of nested lists to 1d-array, or flat list, eliminating all the nesting. Note that nesting can be irregular, i.e. one part of initial list can have deepest elemenets on 3rd level, while the other – on 5th level.

Attention: Tracking of visited objects is not performed, i.e., circular references and self-references will be unpacked again and again endlessly, until max recursion depth limit exceeds with a RecursionError or until the program eats up all the available RAM (in theory, that is; in practice I personally didn’t enconuter that outcome even once). That was the reason of adding level_limit parameter (see below).

```
>>> flatten([1, 2, [3, [4, [[5]], [6, 7, [8]]]]])
[1, 2, 3, 4, 5, 6, 7, 8]
```

Parameters

- **items** (*Iterable[Union[_T, Iterable[_T]]]*) – N-dimensional iterable to unpack.
- **level_limit** (*Optional[int]*) – Adjust how many levels deep can unpacking proceed, e.g. if set to 1, only 2nd-level elements will be raised up to level 1, but not the deeper ones. If set to 2, the first two levels will be unpacked, while keeping the 3rd and others. 0 disables the limit. *None* is treated like a default value, which is set to 50 empirically.

  Note that altering/disabling this limit doesn't affect max recursion depth limiting mechanism, which will (sooner or later) interrupt the attempt to descent on hierarchy with a self-referencing object or several objects forming a circular reference.

Return type
List[_T]

pytermor.common.**char_range**(*start*, *stop*)

Yields all the characters from range of [c1; c2], inclusive (end character c2 is also present, in contrast with classic range()).

```
>>> ''.join(char_range('1', '9'))
'123456789'
```

Note: In some cases the result will seem to be incorrent, i.e. this: pt.char_range('¹', '⁴') yields 8124 characters total. The reason is that the algoritm works with input characters as Unicode codepoints, and '¹', '⁴' are relatively distant from each other: "¹" U+B9, "⁴" Ux2074, which leads to an unexpected results. Character ranges in regular expessetions, e.g. [A-Z0-9] work the same way.

:param start; Character to start from (inclusive) :param stop; Character to stop at (**inclusive**)

pytermor.common.**filterf = functools.partial(<class 'filter'>, None)**

Shortcut for filtering out falsy values from sequences

pytermor.common.**filtern = functools.partial(<class 'filter'>, <function <lambda>>)**

Shortcut for filtering out Nones from sequences

pytermor.common.**filterfv**(*mapping*)

Shortcut for filtering out falsy values from mappings

Return type
dict

pytermor.common.**filternv**(*mapping*)

Shortcut for filtering out None values from mappings

Return type
dict

7.4 pytermor.config

Library fine tuning module.

Functions

get_config()	Return the current config instance.
init_config()	Reset all config vars to default values.
replace_config(cfg)	Replace the global config instance with provided one.

Classes

Config([renderer_class, force_output_mode, ...])	Configuration variables container.

class pytermor.config.**Config**(*renderer_class=<factory>*, *force_output_mode=<factory>*, *default_output_mode=<factory>*, *trace_renders=<factory>*, *prefer_rgb=<factory>*)

Configuration variables container. Values can be modified in two ways:

1) create new *Config* instance from scratch and activate with replace_config();
2) or preliminarily set the corresponding environment variables to intended values, and the default config instance will catch them up on initialization.

See also:

Environment variable list is located in config guide section.

Parameters

- **renderer_class** (*str*) – Explicitly set renderer class (e.g. TmuxRenderer). See config.renderer_class.
- **force_output_mode** (*str*) – Explicitly set output mode (e.g. `xterm_16`; any *value* from OutputMode enum is valid). See config.force_output_mode.
- **default_output_mode** (*str*) – Output mode to use as a fallback value when renderer is unsure about user's terminal capabilities (e.g. `xterm_16`; any *value* from OutputMode enum is valid). Initial value is `xterm_256`. See Config.default_output_mode.
- **prefer_rgb** (*bool*) – By default SGR renderer uses 8-bit color mode sequences for Color256 instances (as it should), even when the output device supports more advanced 24-bit/True Color mode. With this option set to *True* Color256 will be rendered using True Color sequences instead, provided the terminal emulator supports them. Most of the time the results from different color modes are indistinguishable from each other, however, there *are* rare cases, when it does matter. See Config.prefer_rgb.
- **trace_renders** (*bool*) – Set to *True* to log hex dumps of rendered strings. Note that default handler is `logging.NullHandler` with `WARNING` level, so in order to see the traces attached handler is required. See Config.trace_renders.

pytermor.config.**get_config**()

Return the current config instance.

Return type
Config

pytermor.config.**init_config**()

Reset all config vars to default values.

pytermor.config.**replace_config**(*cfg*)

Replace the global config instance with provided one.

7.5 pytermor.cval

Color preset list:

- 16x Color16 (16 unique)
- 256x Color256 (247 unique)
- 2304x ColorRGB (2297 unique)

7.6 pytermor.exception

Exceptions

ArgCountError(actual, *expected)	
ArgTypeError(arg_value, arg_name, *expected_type)	.
ColorCodeConflictError(code, existing_color, ...)	
ColorNameConflictError(tokens, ...)	
ConflictError	
LogicError	
NotInitializedError	
ParseError(groupdict)	
UserAbort	
UserCancel	

exception pytermor.exception.**LogicError**

Bases: Exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pytermor.exception.**ParseError**(*groupdict*)

Bases: Exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pytermor.exception.**ConflictError**

Bases: Exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pytermor.exception.**NotInitializedError**

Bases: Exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pytermor.exception.**ArgTypeError**(*arg_value*, *arg_name*, **expected_type*, *suggestion=None*)

Bases: Exception

.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pytermor.exception.**ArgCountError**(*actual*, **expected*)

Bases: Exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pytermor.exception.**UserCancel**

Bases: Exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pytermor.exception.**UserAbort**

Bases: Exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pytermor.exception.**ColorNameConflictError**(*tokens*, *existing_color*, *new_color*)

Bases: Exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pytermor.exception.**ColorCodeConflictError**(*code*, *existing_color*, *new_color*)

Bases: Exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

7.7 pytermor.filter

Formatters for prettier output and utility classes to avoid writing boilerplate code when dealing with escape sequences. Also includes several Python Standard Library methods rewritten for correct work with strings containing control sequences.

Module Attributes

SGR_SEQ_REGEX	Regular expression that matches *SGR* sequences.
CSI_SEQ_REGEX	Regular expression that matches CSI sequences (a superset which includes *SGRs*).
CONTROL_CHARS	Set of ASCII control characters: 0x00-0x08, 0x0E-0x1F and 0x7F.
WHITESPACE_CHARS	Set of ASCII whitespace characters: 0x09-0x0D and 0x20.
PRINTABLE_CHARS	Set of ASCII "normal" characters, i.e. non-control and non-space ones: letters, digits and punctuation (0x21-0x7E).
NON_ASCII_CHARS	Set of bytes that are invalid in ASCII-7 context: 0x80-0xFF.
IT	input-type
OT	output-type
PTT	pattern type
RPT	replacer type
MPT	# map

Functions

apply_filters(inp, *args)	Method for applying dynamic filter list to a target string/bytes.
center_sgr(string, width[, fillchar])	SGR-formatting-aware implementation of `str.center`.
dump(data[, tracer_cls, extra, force_width])	.
get_max_ucs_chars_cp_length(string)	.
get_max_utf8_bytes_char_length(string)	cc
ljust_sgr(string, width[, fillchar])	SGR-formatting-aware implementation of `str.ljust`.
rjust_sgr(string, width[, fillchar])	SGR-formatting-aware implementation of `str.rjust`.

Classes

AbstractNamedGroupsRefilter(*args, **kwargs)	Substitute the input by applying following rules:
AbstractStringTracer(*args, **kwargs)	
AbstractTracer(*args, **kwargs)	
BytesTracer(*args, **kwargs)	str/bytes as byte hex codes, grouped by 4
CsiStringReplacer(*args, **kwargs)	Find all CSI <SequenceCSI> seqs (i.e., starting with `ESC [`) and replace with given string.
EscSeqStringReplacer(*args, **kwargs)	,
IFilter(*args, **kwargs)	Main idea is to provide a common interface for string filtering, that can make possible working with filters like with objects rather than with functions/lambdas.
IRefilter(*args, **kwargs)	*Refilters* are rendering filters (output is *str* with SGRs).
NonPrintsOmniVisualizer(*args, **kwargs)	Input type: *str*, *bytes*.
NonPrintsStringVisualizer(*args, **kwargs)	Input type: *str*.
NoopFilter(*args, **kwargs)	
OmniDecoder(*args, **kwargs)	
OmniEncoder(*args, **kwargs)	
OmniMapper(*args, **kwargs)	Input type: *str*, *bytes*.
OmniPadder(*args, **kwargs)	
OmniSanitizer(*args, **kwargs)	Input type: *str*, *bytes*.
SgrStringReplacer(*args, **kwargs)	Find all SGR <SequenceSGR> seqs (e.g., `ESC [1; 4m`) and replace with given string.
StringLinearizer(*args, **kwargs)	Filter transforms all whitespace sequences in the input string into a single space character, or into a specified string.
StringMapper(*args, **kwargs)	a
StringReplacer(*args, **kwargs)	.
StringReplacerChain(*args, **kwargs)	.
StringTracer(*args, **kwargs)	str as byte hex codes (UTF-8), grouped by characters
StringUcpTracer(*args, **kwargs)	str as Unicode codepoints
TracerExtra([label, addr_shift, hash])	
WhitespaceRemover(*args, **kwargs)	Special case of StringLinearizer.

pytermor.filter.**SGR_SEQ_REGEX**

Regular expression that matches *SGR* sequences. Group 3 can be used for sequence params extraction.

pytermor.filter.**CSI_SEQ_REGEX**

Regular expression that matches CSI sequences (a superset which includes *SGRs*).

pytermor.filter.**CONTROL_CHARS**

Set of ASCII control characters: 0x00-0x08, 0x0E-0x1F and 0x7F.

pytermor.filter.**WHITESPACE_CHARS**

Set of ASCII whitespace characters: 0x09-0x0D and 0x20.

pytermor.filter.**PRINTABLE_CHARS**

Set of ASCII "normal" characters, i.e. non-control and non-space ones: letters, digits and punctuation (0x21-0x7E).

pytermor.filter.**NON_ASCII_CHARS**

Set of bytes that are invalid in ASCII-7 context: 0x80-0xFF.

pytermor.filter.**IT**

input-type

alias of TypeVar('IT', str, bytes)

pytermor.filter.**OT**

output-type

alias of TypeVar('OT', str, bytes)

pytermor.filter.**PTT**

pattern type

alias of Union[*IT*, Pattern[*IT*]]

pytermor.filter.**RPT**

replacer type

alias of Union[*OT*, Callable[[Match[*OT*]], *OT*]]

pytermor.filter.**MPT**

map

alias of Dict[int, *IT*]

class pytermor.filter.**IFilter**(**args*, ***kwargs*)

Bases: Generic[*IT*, *OT*]

Main idea is to provide a common interface for string filtering, that can make possible working with filters like with objects rather than with functions/lambdas.

Return type
IFilter

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**IRefilter**(**args*, ***kwargs*)

Bases: *IFilter*[*IT*, str]

Refilters are rendering filters (output is *str* with SGRs).

Return type
IFilter

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**OmniPadder**(**args*, ***kwargs*)

Bases: *IFilter*[*IT*, *IT*]

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**StringReplacer**(**args*, ***kwargs*)

Bases: *IFilter*[str, str]

.

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**StringReplacerChain**(**args*, ***kwargs*)

Bases: *StringReplacer*

.

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns

transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type

OT

class pytermor.filter.**EscSeqStringReplacer**(**args*, ***kwargs*)

Bases: *StringReplacer*

,

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns

transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type

OT

class pytermor.filter.**SgrStringReplacer**(**args*, ***kwargs*)

Bases: *StringReplacer*

Find all SGR <SequenceSGR> seqs (e.g., `ESC [1;4m`) and replace with given string. More specific version of `CsiReplacer`.

Parameters

repl (*RPT[str]*) – Replacement, can contain regexp groups (see *apply_filters()*).

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns

transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type

OT

class pytermor.filter.**CsiStringReplacer**(**args*, ***kwargs*)

Bases: *StringReplacer*

Find all CSI <SequenceCSI> seqs (i.e., starting with ESC [) and replace with given string. Less specific version of SgrReplacer, as CSI consists of SGR and many other sequence subtypes.

Parameters
repl (*RPT[str]*) – Replacement, can contain regexp groups (see *apply_filters()*).

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**StringLinearizer**(**args*, ***kwargs*)

Bases: *StringReplacer*

Filter transforms all whitespace sequences in the input string into a single space character, or into a specified string. Most obvious application is pre-formatting strings for log output in order to keep the messages one-lined.

Parameters
repl (*RPT[str]*) – Replacement character(s).

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**WhitespaceRemover**(**args*, ***kwargs*)

Bases: *StringReplacer*

Special case of StringLinearizer. Removes all the whitespaces from the input string.

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**AbstractNamedGroupsRefilter**(**args*, ***kwargs*)

Bases: *IRefilter*[str], *StringReplacer*

Substitute the input by applying following rules:

- Named groups which name is found in `group_st_map` keys are replaced with themselves styled as specified in a corresponding map values.
- Regular/unnamed groups are kept as is, unless there is an "" (empty string) key in `group_st_map`, in which case a style corresponding to such key is applied to all these groups.
- Groups with names not present in the map, as well as lookaheads and lookbehinds, are kept as is (unstyled).
- Non-capturing groups' contents and matched characters not belonging to any group are thrown away.
- Not matched parts of the input are kept as is.

```
>>> import pytermor as pt
>>> class SgrNamedGroupsRefilter(AbstractNamedGroupsRefilter):
...     def _render(self, v: IT, st: FT) -> str:
...         return pt.render(v, st, pt.SgrRenderer(pt.OutputMode.XTERM_16))
...
>>> SgrNamedGroupsRefilter(
...     re.compile(r'<?(<)(?P<val>.+?)(>)>?'),
...     {"val": pt.cv.GREEN},
... ).apply("text <<link>> text")
'text <[32mlink[39m> text'
```

Parameters
group_st_map (*dict[str, FT]*) –

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**OmniMapper**(**args*, ***kwargs*)

Bases: *IFilter*[*IT*, *IT*]

Input type: *str*, *bytes*. Abstract mapper. Replaces every character found in map keys to corresponding map value. Map should be a dictionary of this type: `dict[int, str|bytes]`; moreover, length of *str*/*bytes* must be strictly 1 character (ASCII codepage). If there is a necessity to map Unicode characters, StringMapper should be used instead.

```
>>> OmniMapper({0x20: '.'}).apply(b'abc def ghi')
b'abc.def.ghi'
```

For mass mapping it is better to subclass OmniMapper and override two methods – _get_default_keys and _get_default_replacer. In this case you don't have to manually compose a replacement map with every character you want to replace.

Parameters
override (*MPT*) – a dictionary with mappings: keys must be *ints*, values must be either a single-char *strs* or *bytes*.

See
NonPrintsOmniVisualizer

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**StringMapper**(**args*, ***kwargs*)

Bases: *OmniMapper*[str]

a

Return type
IFilter

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**NonPrintsOmniVisualizer**(**args*, ***kwargs*)

Bases: *OmniMapper*

Input type: *str*, *bytes*. Replace every whitespace character with `.`.

Return type
IFilter

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**NonPrintsStringVisualizer**(**args*, ***kwargs*)

Bases: *StringMapper*

Input type: *str*. Replace every whitespace character with "·", except newlines. Newlines are kept and get prepneded with same char by default, but this behaviour can be disabled with `keep_newlines` = *False*.

```
>>> NonPrintsStringVisualizer(keep_newlines=False).apply("S"+os.linesep+"K")
'SK'
```

Parameters
keep_newlines (*bool*) – When *True*, transform newline characters into "\n", or into just "" otherwise.

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters
- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**OmniSanitizer**(**args*, ***kwargs*)

Bases: *OmniMapper*

Input type: *str*, *bytes*. Replace every control character and every non-ASCII character (0x80-0xFF) with ".", or with specified char. Note that the replacement should be a single ASCII character, because `Omni-` filters are designed to work with *str* inputs and *bytes* inputs on equal terms.

Parameters
repl (*IT*) – Value to replace control/non-ascii characters with. Should be strictly 1 character long.

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters
- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**AbstractTracer**(**args*, ***kwargs*)

Bases: *IFilter*[*IT*, `str`]

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**BytesTracer**(**args*, ***kwargs*)

Bases: *AbstractTracer*[bytes]

str/bytes as byte hex codes, grouped by 4

Listing 1: Example output

```
0x00 | 35 30 20 35  34 20 35 35  20 C2 B0 43  20 20 33 39  20 2B 30 20
0x14 | 20 20 33 39  6D 73 20 31  20 52 55 20  20 E2 88 86  20 35 68 20
0x28 | 31 38 6D 20  20 20 EE 8C  8D 20 E2 80  8E 20 2B 32  30 C2 B0 43
0x3C | 20 20 54 68  20 30 31 20  4A 75 6E 20  20 31 36 20  32 38 20 20
0x50 | E2 96 95 E2  9C 94 E2 96  8F 46 55 4C  4C 20
```

Return type
IFilter

get_max_chars_per_line(*inp*, *addr_shift*)

The amount of characters that will fit into one line (with taking into account all the formatting and the fact that chars are displayed in groups of 4) depends on terminal width and on max address value (the latter determines the size of the leftmost field – current line address). Let's express output line length L_O in a general way – through C_L (characters per line) and L_{adr} (length of maximum address value for given input):

$$L_O = L_{spc} + L_{sep} + L_{adr} + L_{hex},$$

$$L_{adr} = 2 + 2 \cdot ceil(\frac{L_{Ihex}}{2}), \qquad (1)$$

$$L_{hex} = 3C_L + floor(\frac{C_L}{4}),$$

where:

- $L_{spc} = 3$ is static whitespace total length,
- $L_{sep} = 1$ is separator (" | ") length,
- $L_{Ihex} = len(L_I)$ is *length* of (hexadecimal) *length* of input. Here is an example, consider input data I 10 bytes long:

$$L_I = len(I) = 10_{10} = A_{16},$$

$$L_{Ihex} = len(L_I) = len(A_{16}) = 1,$$

$$L_{adr} = 2 + 2 \cdot ceil(\frac{1}{2}) = 4,$$

which corresponds to address formatted as 0x0A. One more example – input data 1000 bytes long:

$$L_I = len(I) = 1000_{10} = 3E8_{16},$$

$$L_{Ihex} = len(L_I) = len(3E8_{16}) = 3,$$

$$L_{adr} = 2 + 2 \cdot ceil(\frac{3}{2}) = 6,$$

which matches the length of an actual address 0x03E8). Note that the expression $2 \cdot ceil(\frac{L_{Ihex}}{2})$ is used for rounding L_{adr} up to next even integer to avoid printing the addresses in 0x301 form, and displaying them more or less aligned instead. The first constant item 2 in (1) represents 0x prefix.

- L_{hex} represents amount of chars required to display C_L hexadecimal bytes. First item $3C_L$ is trivial and corresponds to every byte's hexadecimal value plus a space after (giving us $2 + 1 = 3$, e.g. `"34 "`), while the second one represents one extra space character per each 4-byte group.

Let's introduce L_T as current terminal width, then $\boxed{L_O \leqslant L_T}$, which leads to the following inequation:

$$L_{spc} + L_{sep} + L_{adr} + L_{hex} \leqslant L_T.$$

Substitute the variables:

$$3 + 1 + 2 + 2 \cdot ceil(\frac{L_{Ihex}}{2}) + 3C_L + floor(\frac{C_L}{4}) \leqslant L_T.$$

Suppose we limit C_L values to the integer factor of 4, then:

$$3C_L + floor(\frac{C_L}{4}) = 3.25C_L \qquad \forall C_L \in [4, 8, 12..), \qquad (2)$$

which gives us:

$$6 + 2 \cdot ceil(\frac{L_{Ihex}}{2}) + 3.25C_L \leqslant L_T,$$

$$3.25C_L \leqslant L_T - 2 \cdot ceil(\frac{L_{Ihex}}{2}) - 6,$$

$$13C_L \leqslant 4L_T - 8 \cdot ceil(\frac{L_{Ihex}}{2}) - 24.$$

Therefore:

$$C_{Lmax} = floor(\frac{4L_T - 4 \cdot ceil(\frac{L_{Ihex}}{2}) - 24}{13}).$$

Last step would be to round the result (down) to the nearest integer factor of 4 as we have agreed earlier in (2).

Parameters
inp (*bytes*) –

Return type
int

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**AbstractStringTracer**(**args*, ***kwargs*)

Bases: *AbstractTracer*[str]

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**StringTracer**(**args*, ***kwargs*)

Bases: *AbstractStringTracer*

str as byte hex codes (UTF-8), grouped by characters

Listing 2: Example output

```
  0 |     35     30     20 35 34 20 35     35     20   c2b0 43 20 |50␣54␣55␣°C␣
 12 |     20     33     39 20 2b 30 20     20     20     33 39 6d |␣39␣+0␣␣␣39m
 24 |     73     20     31 20 52 55 20     20 e28886     20 35 68 |s␣1␣RU␣␣␣5h
 36 |     20     31     38 6d 20 20 20 ee8c8d     20 e2808e 20 2b |␣18m␣␣␣␣␣+
 48 |     32     30   c2b0 43 20 20 54     68     20     30 31 20 |20°C␣␣Th␣01␣
 60 |     4a     75     6e 20 20 31 36     20     32     38 20 20 |Jun␣␣16␣28␣␣
 72 | e29695 e29c94 e2968f 46 55 4c 4c     20                     |✓FULL␣
```

Return type
IFilter

get_max_chars_per_line(*inp*, *addr_shift*)

For more detials on math behind these calculations see *BytesTracer*.

Calculations for this class are different, although the base formula for output line length L_O is the same:

$$L_O = L_{spc} + L_{sep} + L_{adr} + L_{hex},$$

$$L_{adr} = len(L_I),$$

$$L_{hex} = (2C_{Umax} + 1) \cdot C_L$$

where:

- $L_{spc} = 3$ is static whitespace total length,
- $L_{sep} = 2$ is separators "|" total length,

- L_{adr} is length of maximum address value and is equal to *length* of *length* of input data without any transformations (because the output is decimal, in contrast with `BytesTracer`),
- L_{hex} is hex representation length (2 chars multiplied to C_{Umax} plus 1 for space separator per each character),
- C_{Umax} is maximum UTF-8 bytes amount for a single codepoint encountered in the input (for example, C_{Umax} equals to 1 for input string consisting of ASCII-7 characters only, like "ABCDE", 2 for "", 3 for "" and 4 for "", which is U+10FFFF),
- $L_{chr} = C_L$ is char representation length (equals to C_L), and
- C_L is chars per line setting.

Then the condition of fitting the data to a terminal can be written as:

$$L_{spc} + L_{sep} + L_{adr} + L_{hex} + L_{chr} \leqslant L_T,$$

where L_T is current terminal width. Next:

$$3 + 2 + L_{adr} + (2C_{Umax} + 1) \cdot C_L + C_L, \leqslant L_T$$

$$L_{adr} + 5 + (2C_{Umax} + 2) \cdot C_L, \leqslant L_T$$

Express C_L through L_T, L_{adr} and C_{Umax}:

$$(2C_{Umax} + 2) \cdot C_L \leqslant L_T - L_{adr} - 5,$$

Therefore maximum chars per line equals to:

$$C_{Lmax} = floor(\frac{L_T - L_{adr} - 5}{2C_{Umax} + 2}).$$

Example

Consider terminal width is 80, input data is 64 characters long and consists of U+10FFFF codepoints only ($C_{Umax} = 4$). Then:

$$\begin{aligned} L_{adr} &= len(L_I) = len(64) = 2, \\ C_{Lmax} &= floor(\frac{78 - 2 - 5}{8 + 2}), \\ &= floor(7.1) = 7. \end{aligned}$$

Note: Max width value used in calculations is slightly smaller than real one, that's why output lines are 78 characters long (instead of 80) – there is a 2-char reserve to ensure that the output will fit to the terminal window regardless of terminal emulator type and implementation.

The calculations always consider the maximum possible length of input data chars, and even if it will consist of the highest order codepoints only, it will be perfectly fine.

```
 0 | f4808080 f4808080 f4808080 f4808080 f4808080 f4808080 f4808080 |
 7 | f4808080 f4808080 f4808080 f4808080 f4808080 f4808080 f4808080 |
14 | ...
```

More realistic example with various byte lengths is given in `class` documentation above.

Parameters
inp (*str*) –

Return type
int

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**StringUcpTracer**(**args*, ***kwargs*)

Bases: *AbstractStringTracer*

str as Unicode codepoints

Listing 3: Example output

```
  0 |U+   20   34   36 20 34 36 20 34   36   20 B0 43 20 20 33   39 20 2B␣
↪|␣46␣46␣46␣°C␣␣39␣+
 18 |U+   30   20   20 20 35 20 6D 73   20   31 20 52 55 20 20 2206 20 37␣
↪|0␣␣␣5␣ms␣1␣RU␣␣␣7
 36 |U+   68   20   32 33 6D 20 20 20 FA93 200E 20 2B 31 33 B0   43 20 20␣
↪|h␣23m␣␣␣␣+13°C␣␣
 54 |U+   46   72   20 30 32 20 4A 75   6E   20 20 30 32 3A 34   38 20 20␣
↪|Fr␣02␣Jun␣␣02:48␣␣
 72 |U+ 2595 2714 258F 46 55 4C 4C 20                              ␣
↪|✓FULL␣
```

Return type
IFilter

get_max_chars_per_line(*inp*, *addr_shift*)

Calculations for *StringUcpTracer* are almost the same as for *StringTracer*, expect that sum of static parts of L_O equals to 7 instead of 5 (because of "U+" prefix being displayed).

The second difference is using C_{UCmax} instead of C_{Umax}; the former variable is the amount of "n" in U+nnnn identifier of the character, while the latter is amount of bytes required to encode the character in UTF-8. Final formula is:

$$C_{Lmax} = floor(\frac{L_T - L_{adr} - 7}{C_{UCmax} + 2}).$$

Parameters
inp –

Return type
int

apply(*inp*, *extra=None*)

Apply the filter to input *str* or *bytes*.

Parameters

- **inp** (*IT*) – input string
- **extra** (*Any*) – additional options

Returns
transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
OT

class pytermor.filter.**TracerExtra**(*label: 'str' = ''*, *addr_shift: 'int' = 0*, *hash: 'bool' = False*)

pytermor.filter.**dump**(*data*, *tracer_cls=None*, *extra=None*, *force_width=None*)

.

Return type
str

pytermor.filter.**get_max_ucs_chars_cp_length**(*string*)

.

Return type
int

pytermor.filter.**get_max_utf8_bytes_char_length**(*string*)

cc

Return type
int

pytermor.filter.**ljust_sgr**(*string*, *width*, *fillchar=' '*)

SGR-formatting-aware implementation of `str.ljust`.

Return a left-justified string of length `width`. Padding is done using the specified fill character (default is a space).

Return type
str

pytermor.filter.**center_sgr**(*string*, *width*, *fillchar=' '*)

SGR-formatting-aware implementation of `str.center`.

Return a centered string of length `width`. Padding is done using the specified fill character (default is a space).

Return type
str

pytermor.filter.**rjust_sgr**(*string*, *width*, *fillchar=' '*)

SGR-formatting-aware implementation of `str.rjust`.

Return a right-justified string of length `width`. Padding is done using the specified fill character (default is a space).

Return type
str

pytermor.filter.**apply_filters**(*inp*, **args*)

Method for applying dynamic filter list to a target string/bytes.

Example (will replace all ESC control characters to `E` and thus make SGR params visible):

```
>>> from pytermor import SeqIndex
>>> test_str = f'{SeqIndex.RED}test{SeqIndex.COLOR_OFF}'
>>> apply_filters(test_str, SgrStringReplacer('E\2\3\4'))
'E[31mtestE[39m'
```

(continues on next page)

(continued from previous page)

```
>>> apply_filters('[31mtest[39m', OmniSanitizer)
'.[31mtest.[39m'
```

Note that type of `inp` argument must be same as filter parameterized input type (IT), i.e. $StringReplacer$ is `IFilter[str, str]` type, so you can apply it only to *str*-type inputs.

Parameters

- **inp** (*IT*) – String/bytes to filter.
- **args** (*Union[IFilter, Type[IFilter]]*) – Instance(s) implementing $IFilter$ or their type(s).

Return type
OT

7.8 pytermor.log

Functions

get_logger()	
init_logger()	
measure()	Decorrator

pytermor.log.**measure**(*__origin: _F*) → _F
pytermor.log.**measure**(**, formatter: _MFT = None, level=TRACE*) → Callable[[_F], _F]

Decorrator

7.9 pytermor.numfmt

utilnum

Module Attributes

PREFIXES_SI_DEC	Prefix preset used by format_si() and format_bytes_human().

Functions

format_auto_float(val, req_len[, allow_exp_form])	Dynamically adjust decimal digit amount and format to fill up the output string with as many significant digits as possible, and keep the output length strictly equal to req_len at the same time.
format_bytes_human(val[, auto_color])	Invoke special case of fixed-length SI formatter optimized for processing byte-based values.
format_si(val[, unit, auto_color])	Invoke fixed-length decimal SI formatter; format `value` as a unitless value with SI-prefixes; a unit can be provided as an argument of format() method.
format_si_binary(val[, unit, auto_color])	Invoke fixed-length binary SI formatter which formats `value` as binary size ("KiB", "MiB") with base 1024.
format_thousand_sep(val[, separator])	Returns input `val` with integer part split into groups of three digits, joined then with `separator` string.
format_time(val_sec[, auto_color])	Invoke dynamic-length general-purpose time formatter, which supports a wide range of output units, including seconds, minutes, hours, days, weeks, months, years, milliseconds, microseconds, nanoseconds etc.
format_time_delta(val_sec[, max_len, auto_color])	Format time interval using the most suitable format with one or two time units, depending on `max_len` argument.
format_time_delta_longest(val_sec[, auto_color])	Wrapper around format_time_delta() with pre-set longest formatter.
format_time_delta_shortest(val_sec[, auto_color])	Wrapper around format_time_delta() with pre-set shortest formatter.
format_time_ms(value_ms[, auto_color])	Invoke a variation of formatter_time specifically configured to format small time intervals.
format_time_ns(value_ns[, auto_color])	Wrapper for format_time_ms() expecting input value as nanoseconds.
highlight(string)	

Classes

BaseUnit(oom[, unit, prefix, _integer])	
DualBaseUnit(name[, in_next, ...])	TU
DualFormatter([fallback, units, auto_color, ...])	Formatter designed for time intervals.
DualFormatterRegistry()	Simple DualFormatter registry for storing formatters and selecting the suitable one by max output length.
DynamicFormatter([fallback, units, ...])	A simplified version of static formatter for cases, when length of the result string doesn't matter too much (e.g., for log output), and you don't have intention to customize the output (too much).
Highlighter([dim_units])	S
NumFormatter(auto_color, highlighter)	
StaticFormatter([fallback, max_value_len, ...])	Format `value` using settings passed to constructor.
SupportsFallback()	

pytermor.numfmt.**PREFIXES_SI_DEC = ['q', 'r', 'y', 'z', 'a', 'f', 'p', 'n', 'µ', 'm', None, 'k', 'M', 'G', 'T', 'P', 'E', 'Z', 'Y', 'R', 'Q']**

Prefix preset used by format_si() and format_bytes_human(). Covers values from 10^{-30} to 10^{32}. Note lower-cased 'k' prefix.

class pytermor.numfmt.**Highlighter**(*dim_units=True*)

S

colorize(*string*)

parse and highlight

Parameters
string (*str*) –

Returns

Return type
Text

apply(*intp*, *frac*, *sep*, *pfx*, *unit*)

highlight already parsed

Parameters

- **intp** (*str*) –
- **frac** (*str*) –
- **sep** (*str*) –
- **pfx** (*str*) –
- **unit** (*str*) –

Returns

Return type
List[Fragment]

class pytermor.numfmt.**StaticFormatter**(*fallback=None*, ***, *max_value_len=None*, *auto_color=None*, *allow_negative=None*, *allow_fractional=None*, *discrete_input=None*, *unit=None*, *unit_separator=None*, *mcoef=None*, *pad=None*, *legacy_rounding=None*, *prefixes=None*, *prefix_refpoint_shift=None*, *value_mapping=None*, *highlighter=None*)

Bases: NumFormatter

Format `value` using settings passed to constructor. The purpose of this class is to fit into specified string length as much significant digits as it's theoretically possible by using multipliers and unit prefixes. Designed for metric systems with bases 1000 or 1024.

The key property of this formatter is maximum length – the output will not excess specified amount of characters no matter what (that's what is "static" for).

You can create your own formatters if you need fine tuning of the output and customization. If that's not the case, there are facade methods format_si(), format_si_binary() and format_bytes_human(), which will invoke predefined formatters and doesn't require setting up.

Parameters

- **fallback** (StaticFormatter) – For any (constructing) instance attribute without a value (=*None*): look up for this attribute in `fallback` instance, and if the value is specified, take it and save as yours own; if the attribute is undefined in `fallback` as well, use the default class value for this attribute instead.
- **max_value_len** (*int*) – [default: 4] Target string length. Must be at least **3**, because it's a minimum requirement for formatting values from 0 to 999. Next number to 999 is 1000, which will be formatted as "1k".

Setting `allow_negative` to *True* increases lower bound to **4** because the values now can be less than 0, and minus sign also occupies one char in the output.

Setting `mcoef` to anything other than 1000.0 also increases the minimum by 1, to **5**. The reason is that non-decimal coefficients like 1024 require additional char to render as switching to the next prefix happens later: "999 b", "1000 b", "1001 b", ..."1023 b", "1 Kb".

- **auto_color** (*bool*) – [default: *False*] Enable automatic colorizing of the result. Color depends on order of magnitude of the value, and always the same, e.g.: blue color for numbers in $[1000; 10^6)$ and $[10^{-3}; 1)$ ranges (prefixes nearest to 1, kilo- and milli-); cyan for values in $[10^6; 10^9)$ and $[10^{-6}; 10^{-3})$ ranges (next ones, mega-/micro-), etc. The values from $[1; 999]$ are colored in neutral gray. See *Highlighter*.
- **allow_negative** (*bool*) – [default: *True*] Allow negative numbers handling, or (if set to *False*) ignore the sign and round all of them to 0.0. This option effectively increases lower limit of `max_value_len` by 1 (when enabled).
- **allow_fractional** (*bool*) – [default: *True*] Allows the usage of fractional values in the output. If set to *False*, the results will be rounded. Does not affect lower limit of `max_value_len`.
- **discrete_input** (*bool*) – [default: *False*] If set to *True*, truncate the fractional part off the input and do not use floating-point format for *base output*, i.e., without prefix and multiplying coefficient. Useful when the values are originally discrete (e.g., bytes). Note that the same effect could be achieved by setting `allow_fractional` to *False*, except that it will influence prefixed output as well ("1.08 kB" -> "1kB").
- **unit** (*str*) – [default: empty *str*] Unit to apply prefix to (e.g., "m", 'B"). Can be empty.
- **unit_separator** (*str*) – [default: a space] String to place in between the value and the (prefixed) unit. Can be empty.
- **mcoef** (*float*) – [default: 1000.0] Multiplying coefficient applied to the value:

$$V_{out} = V_{in} * b^{(-m/3)},$$

where: V_{in} is an input value, V_{out} is a numeric part of the output, b is `mcoef` (base), and m is the order of magnitude corresponding to a selected unit prefix. For example, in case of default (decimal) formatter and input value equal to 17345989 the selected prefix will be "M" with the order of magnitude = 6:

$$V_{out} = 17345989 * 1000^{(-6/3)} = 17345989 * 10^{-6} = 17.346.$$

- **pad** (*bool* | *Align*) – [default: *False*] @TODO
- **legacy_rounding** (*bool*) – [default: *False*] @TODO
- **prefixes** (*list[str|None]*) – [default: *PREFIXES_SI_DEC*] Prefix list from min power to max. Reference point (with zero-power multiplier, or 1.0) is determined by searching for *None* in the list provided, therefore it's a requirement for the argument to have at least one *None* value. Prefix list for a formatter without fractional values support could look like this:

```
[None, "k", "M", "G", "T"]
```

Prefix step is fixed to $log_{10}1000 = 3$, as specified for metric prefixes.

- **prefix_refpoint_shift** (*int*) – [default: 0] Should be set to a non-zero number if input represents already prefixed value; e.g. to correctly format a variable, which stores the frequency in MHz, set prefix shift to 2; the formatter then will render 2333 as "2.33 GHz" instead of incorrect "2.33 kHz".
- **value_mapping** (*t.Dict[float, RT] | t.Callable[[float], RT]*) – @TODO

- **highlighter** (*t.Type[Highlighter] | Highlighter*) – ...

get_max_len(*unit=None*)

Parameters
unit (*Optional[str]*) – Unit override. Set to *None* to use formatter default.

Returns
Maximum length of the result. Note that constructor argument is `max_value_len`, which is a different parameter.

Return type
int

format(*val*, *unit=None*, *auto_color=None*)

Parameters

- **val** (*float*) – Input value.
- **unit** (*Optional[str]*) – Unit override. Set to *None* to use formatter default.
- **auto_color** (*Optional[bool]*) – Color mode, *bool* to enable/disable auto-colorizing, *None* to use formatter default value.

Returns
Formatted value, *Text* if colorizing is on, *str* otherwise.

Return type
RT

class pytermor.numfmt.**DynamicFormatter**(*fallback=None*, *units=None*, ***, *auto_color=None*, *allow_fractional=None*, *unit_separator=None*, *oom_shift=None*, *highlighter=None*)

Bases: `NumFormatter`

A simplified version of static formatter for cases, when length of the result string doesn't matter too much (e.g., for log output), and you don't have intention to customize the output (too much).

Note: Mp mp not note

---

format(*val*, *auto_color=False*, *oom_shift=None*)

,,, :param val: :param oom_shift: :param auto_color: :return:

Return type
RT

class pytermor.numfmt.**BaseUnit**(*oom: 'float'*, *unit: 'str' = ''*, *prefix: 'str' = ''*, *_integer: 'bool' = None*)

class pytermor.numfmt.**DualFormatter**(*fallback=None*, *units=None*, ***, *auto_color=None*, *allow_negative=None*, *allow_fractional=None*, *unit_separator=None*, *pad=None*, *plural_suffix=None*, *overflow_msg=None*, *highlighter=None*)

Bases: `NumFormatter`

Formatter designed for time intervals. Key feature of this formatter is ability to combine two units and display them simultaneously, e.g. return "3h 48min" instead of "228 mins" or "3 hours", etc.

It is possible to create custom formatters if fine tuning of the output and customization is necessary; otherwise use a facade method *format_time_delta()*, which selects appropriate formatter by specified max length from a preset list.

Example output:

```
"10 secs", "5 mins", "4h 15min", "5d 22h"
```

Parameters

- **fallback** (DualFormatter) –
- **units** (*t.List[DualBaseUnit]*) –
- **auto_color** (*bool*) – If *True*, the result will be colorized depending on unit type.
- **allow_negative** (*bool*) –
- **allow_fractional** (*bool*) –
- **unit_separator** (*str*) –
- **pad** (*bool | Align*) – Set to *True* to pad the value with spaces on the left side and ensure it's length is equal to *max_len*, or to *False* to allow shorter result strings.
- **plural_suffix** (*str*) –
- **overflow_msg** (*str*) –
- **highlighter** (*t.Type[Highlighter]*) –

property max_len: int

This property cannot be set manually, it is computed on initialization automatically.

Returns
Maximum possible output string length.

format(*val_sec*, *auto_color=None*)

Pretty-print difference between two moments in time. If input value is too big for the current formatter to handle, return "OVERFLOW" string (or a part of it, depending on `max_len`).

Parameters

- **val_sec** (*float*) – Input value in seconds.
- **auto_color** (*Optional[bool]*) – Color mode, *bool* to enable/disable colorizing, *None* to use formatter default value.

Returns
Formatted time delta, *Text* if colorizing is on, *str* otherwise.

Return type
RT

format_base(*val_sec*, *auto_color=None*)

Pretty-print difference between two moments in time. If input value is too big for the current formatter to handle, return *None*.

Parameters

- **val_sec** (*float*) – Input value in seconds.
- **auto_color** (*Optional[bool]*) – Color mode, *bool* to enable/disable colorizing, *None* to use formatter default value.

Returns
Formatted value as *Text* if colorizing is on; as *str* otherwise. Returns *None* on overflow.

Return type
Optional[*RT*]

class pytermor.numfmt.**DualBaseUnit**(*name*, *in_next=None*, *overflow_after=None*, *custom_short=None*, *collapsible_after=None*)

TU

Important: `in_next` and `overflow_after` are mutually exclusive, and either of them is required.

Parameters

- **name** (*str*) – A unit name to display.
- **in_next** (*int*) – The base – how many current units the next (single) unit contains, e.g., for an hour in context of days:

```
CustomBaseUnit("hour", 24)
```

- **overflow_after** (*int*) – Value upper limit.
- **custom_short** (*str*) – Use specified short form instead of first letter of `name` when operating in double-value mode.
- **collapsible_after** (*int*) – Min threshold for double output to become a regular one.

class pytermor.numfmt.**DualFormatterRegistry**

Simple DualFormatter registry for storing formatters and selecting the suitable one by max output length.

register(**formatters*)

…

find_matching(*max_len*)

…

Return type
pytermor.numfmt.DualFormatter | None

get_by_max_len(*max_len*)

…

Return type
pytermor.numfmt.DualFormatter | None

get_shortest()

…

Return type
pytermor.numfmt.DualFormatter | None

get_longest()

…

Return type
pytermor.numfmt.DualFormatter | None

pytermor.numfmt.**format_thousand_sep**(*val*, *separator=' '*)

Returns input `val` with integer part split into groups of three digits, joined then with `separator` string.

```
>>> format_thousand_sep(260341)
'260 341'
>>> format_thousand_sep(-9123123123.55, ',')
'-9,123,123,123.55'
```

Max output len
$(L + max(0, floor(M/3)))$,

where L is `val` length, and M is order of magnitude of `val`

Parameters

- **val** (*int | float*) – value to format
- **separator** (*str*) – character(s) to use as thousand separators

Return type
str

pytermor.numfmt.**format_auto_float**(*val*, *req_len*, *allow_exp_form=True*)

Dynamically adjust decimal digit amount and format to fill up the output string with as many significant digits as possible, and keep the output length strictly equal to `req_len` at the same time.

For values impossible to fit into a string of required length and when rounding doesn't help (e.g. 12 500 000 and 5 chars) algorithm switches to scientific notation, and the result looks like '1.2e7'. If this feature is explicitly disabled with `allow_exp_form` = *False*, then:

1) if absolute value is less than 1, zeros will be returned ('0.0000');
2) if value is a big number (like 10^9), `ValueError` will be raised instead.

```
>>> format_auto_float(0.012345678, 5)
'0.012'
>>> format_auto_float(0.123456789, 5)
'0.123'
>>> format_auto_float(1.234567891, 5)
'1.235'
>>> format_auto_float(12.34567891, 5)
'12.35'
>>> format_auto_float(123.4567891, 5)
'123.5'
>>> format_auto_float(1234.567891, 5)
' 1235'
>>> format_auto_float(12345.67891, 5)
'12346'
```

Max output len
adjustable

Parameters

- **val** (*float*) – Value to format.
- **req_len** (*int*) – Required output string length.
- **allow_exp_form** (*bool*) – Allow scientific notation usage when that's the only way of fitting the value into a string of required length.

Raises
ValueError – When value is too long and `allow_exp_form` is *False*.

Return type
str

pytermor.numfmt.**format_si**(*val*, *unit=None*, *auto_color=None*)

Invoke fixed-length decimal SI formatter; format `value` as a unitless value with SI-prefixes; a unit can be provided as an argument of *format()* method. Suitable for formatting any SI unit with values from 10^{-30} to 10^{32}.

Total maximum length is `max_value_len + 2`, which is **6** by default (4 from value + 1 from separator and + 1 from prefix). If the unit is defined and is a non-empty string, the maximum output length increases by length of that unit.

Listing 4: Extending the formatter

```
my_formatter = StaticFormatter(formatter_si)
```

```
>>> format_si(1010, 'm²')
'1.01 km²'
>>> format_si(0.223, 'g')
'223 mg'
>>> format_si(1213531546, 'W')  # great scott
'1.21 GW'
>>> format_si(1.22e28, 'eV')  # the Planck energy
'12.2 ReV'
```

Max output len
6

Parameters

- **val** (*float*) – Input value (unitless).
- **unit** (*Optional[str]*) – A unit override [default unit is an empty string].
- **auto_color** (*Optional[bool]*) – Color mode override, *bool* to enable/disable colorizing depending on unit type, *None* to use formatters' setting value [*False* by default].

Returns
Formatted value, *Text* if colorizing is on, *str* otherwise.

Return type
RT

pytermor.numfmt.**format_si_binary**(*val*, *unit=None*, *auto_color=False*)

Invoke fixed-length binary SI formatter which formats `value` as binary size ("KiB", "MiB") with base 1024. Unit can be customized. Covers values from 0 to 10^{32}.

While being similar to `formatter_si`, this formatter differs in one aspect. Given a variable with default value = 995, formatting it results in "995 B". After increasing it by 20 it equals to 1015, which is still not enough to become a kilobyte – so returned value will be "1015 B". Only after one more increase (at 1024 and more) the value will morph into "1.00 KiB" form.

That's why the initial `max_value_len` should be at least 5 – because it is a minimum requirement for formatting values from 1023 to -1023. However, The negative values for this formatter are disabled by default and rendered as 0, which decreases the `max_value_len` minimum value back to 4.

Total maximum length of the result is `max_value_len + 4` = **8** (base + 1 from separator + 1 from unit + 2 from prefix, assuming all of them have default values defined in `formatter_si_binary`).

Listing 5: Extending the formatter

```
my_formatter = StaticFormatter(formatter_si_binary)
```

```
>>> format_si_binary(1010)  # 1010 b < 1 kb
'1010 B'
>>> format_si_binary(1080)
'1.05 KiB'
>>> format_si_binary(45200)
'44.1 KiB'
```

(continues on next page)

(continued from previous page)

```
>>> format_si_binary(1.258 * pow(10, 6), 'b')
'1.20 Mib'
```

Max output len
8

Parameters

- **val** (*float*) – Input value in bytes.
- **unit** (*Optional[str]*) – A unit override [default unit is "B"].
- **auto_color** (*bool*) – Color mode override, *bool* to enable/disable colorizing depending on unit type, *None* to use formatters' setting value [*False* by default].

Returns
Formatted value, *Text* if colorizing is on, *str* otherwise.

Return type
RT

pytermor.numfmt.**format_bytes_human**(*val*, *auto_color=False*)

Invoke special case of fixed-length SI formatter optimized for processing byte-based values. Inspired by default stats formatting used in htop. Comprises traits of both preset SI formatters, the key ones being:

- expecting integer inputs;
- prohibiting negative inputs;
- operating in decimal mode with the base of 1000 (not 1024);
- the absence of units and value-unit separators in the output, while prefixes are still present;
- (if colors allowed) utilizing *Highlighter* with a bit customized setup, as detailed below.

Total maximum length is `max_value_len + 1`, which is **5** by default (4 from value + 1 from prefix).

Highlighting options

Default highlighter for this formatter does not render units (as well as prefixes) dimmed. The main reason for that is the absence of actual unit in the output of this formatter, while prefixes are still there; this allows to format the fractional output this way: [1].57[k], where brackets [] indicate brighter colors.

This format is acceptable because only essential info gets highlighted; however, in case of other formatters with actual units in the output this approach leads to complex and mixed-up formatting; furthermore, it doesn't matter if the highlighting affects the prefix part only or both prefix and unit parts – in either case it's just too much formatting on a unit of surface: [1].53 [Ki]B (looks patchworky).

Table 1: Default formatters comparison

Value	`SI(unit='B')`	`SI_BINARY`	`BYTES_HUMAN`
1568	'1.57 kB'	'1.53 KiB'	'1.57k'
218371331	'218 MB'	'208 MiB'	'218M'
0.25	'250 mB'[1]	'0 B'	'0'
-1218371331232	'-1.2 TB'	'0 B'	'0'

[1] 250 millibytes is not something you would see every day

Listing 6: Extending the formatter

```
my_formatter = StaticFormatter(formatter_bytes_human, unit_separator=" ")
```

```
>>> format_bytes_human(990)
'990'
>>> format_bytes_human(1010)
'1.01k'
>>> format_bytes_human(45200)
'45.2k'
>>> format_bytes_human(1.258 * pow(10, 6))
'1.26M'
```

Max output len
5

Parameters

- **val** (*int*) – Input value in bytes.
- **auto_color** (*bool*) – Color mode override, *bool* to enable/disable colorizing depending on unit type, *None* to use formatters' setting value [*False* by default].

Returns
Formatted value, *Text* if colorizing is on, *str* otherwise.

Return type
RT

pytermor.numfmt.**format_time**(*val_sec*, *auto_color=None*)

Invoke dynamic-length general-purpose time formatter, which supports a wide range of output units, including seconds, minutes, hours, days, weeks, months, years, milliseconds, microseconds, nanoseconds etc.

Listing 7: Extending the formatter

```
my_formatter = DynamicFormatter(formatter_time, unit_separator=" ")
```

```
>>> format_time(12)
'12.0 s'
>>> format_time(65536)
'18 h'
>>> format_time(0.00324)
'3.2 ms'
```

Max output len
varying

Parameters

- **val_sec** (*float*) – Input value in seconds.
- **auto_color** (*Optional[bool]*) – Color mode override, *bool* to enable/disable colorizing depending on unit type, *None* to use formatters' setting value [*False* by default].

Return type
RT

pytermor.numfmt.**format_time_ms**(*value_ms*, *auto_color=None*)

Invoke a variation of `formatter_time` specifically configured to format small time intervals.

Listing 8: Extending the formatter

```
my_formatter = DynamicFormatter(formatter_time_ms, unit_separator=" ")
```

```
>>> format_time_ms(1)
'1ms'
>>> format_time_ms(344)
'344ms'
>>> format_time_ms(0.967)
'967µs'
```

Parameters

- **value_ms** (*float*) – Input value in milliseconds.
- **auto_color** (*Optional[bool]*) – Color mode override, *bool* to enable/disable colorizing depending on unit type, *None* to use formatters' setting value [*False* by default].

Returns

Return type
RT

pytermor.numfmt.**format_time_ns**(*value_ns*, *auto_color=None*)

Wrapper for *format_time_ms()* expecting input value as nanoseconds.

```
>>> format_time_ns(1003000)
'1ms'
>>> format_time_ns(3232332224)
'3s'
>>> format_time_ns(9932248284343.32)
'2h'
```

Parameters

- **value_ns** (*float*) – Input value in nanoseconds.
- **auto_color** (*Optional[bool]*) – Color mode override, *bool* to enable/disable colorizing depending on unit type, *None* to use formatters' setting value [*False* by default].

Returns

Return type
RT

pytermor.numfmt.**format_time_delta**(*val_sec*, *max_len=None*, *auto_color=None*)

Format time interval using the most suitable format with one or two time units, depending on `max_len` argument. Key feature of this formatter is an ability to combine two units and display them simultaneously, e.g. return "3h 48min" instead of "228 mins" or "3 hours", and on top of that – fixed-length output.

There are predefined formatters with output lengths of **3, 4, 5, 6** and **10** characters. Therefore, you can pass in any value from 3 inclusive and it's guarenteed that result's length will be less or equal to required length. If *max_len* is omitted, longest registred formatter will be used.

Note: Negative values are supported by formatters 5 and 10 only.

```
>>> format_time_delta(10, 3)
'10s'
```

(continues on next page)

(continued from previous page)

```
>>> format_time_delta(10, 6)
'10.0s'
>>> format_time_delta(15350, 4)
'4 h'
>>> format_time_delta(15350)
'4h 15min'
```

Max output len
3, 4, 5, 6, 10

Parameters

- **val_sec** (*float*) – Input value in seconds.
- **max_len** (*Optional[int]*) – Maximum output string length (total).
- **auto_color** (*Optional[bool]*) – Color mode override, *bool* to enable/disable colorizing depending on unit type, *None* to use formatters' setting value [*False* by default].

Return type
RT

pytermor.numfmt.**format_time_delta_shortest**(*val_sec*, *auto_color=None*)

Wrapper around *format_time_delta()* with pre-set shortest formatter.

Max output len
3

Parameters

- **val_sec** (*float*) – Input value in seconds.
- **auto_color** (*Optional[bool]*) – Color mode override, *bool* to enable/disable colorizing depending on unit type, *None* to use formatters' setting value [*False* by default].

Return type
RT

pytermor.numfmt.**format_time_delta_longest**(*val_sec*, *auto_color=None*)

Wrapper around *format_time_delta()* with pre-set longest formatter.

Max output len
10

Parameters

- **val_sec** (*float*) – Input value in seconds.
- **auto_color** (*Optional[bool]*) – Color mode override, *bool* to enable/disable colorizing depending on unit type, *None* to use formatters' setting value [*False* by default].

Return type
RT

pytermor.numfmt.**highlight**(*string*)

Todo: @TODO

Max output len
same as input

Parameters
string (*str*) – input text

Return type
RT

7.10 pytermor.renderer

Renderers transform *Style* instances into lower-level abstractions like *SGR sequences*, tmux-compatible directives, HTML markup etc., depending on renderer type. Default global renderer type is SgrRenderer.

Functions

force_ansi_rendering()	Shortcut for forcing all control sequences to be present in the output of a global renderer.
force_no_ansi_rendering()	Shortcut for disabling all output formatting of a global renderer.
init_renderer()	

Classes

HtmlRenderer()	Translate Styles <Style> attributes into a rudimentary HTML markup.
IRenderer(*[, allow_cache, allow_format])	Renderer interface.
NoOpRenderer()	Special renderer type that does nothing with the input string and just returns it as is (i.e.
OutputMode(value)	Determines what types of SGR sequences are allowed to use in the output.
RendererManager()	Class for global rendering mode setup.
SgrDebugger([output_mode])	Subclass of regular SgrRenderer with two differences -- instead of rendering the proper ANSI escape sequences it renders them with ESC character replaced by "␛", and encloses the whole sequence into '()' for visual separation.
SgrRenderer([output_mode, io])	Default renderer invoked by Text.render().
TmuxRenderer()	Translates Styles <Style> attributes into tmux-compatible markup.

class pytermor.renderer.**RendererManager**

Class for global rendering mode setup. For the details and recommendations see guide.renderer_setup.

classmethod set_default(*renderer=None*)

Select a global renderer. See also: guide.renderer_priority.

Parameters

renderer (*Optional[Union[IRenderer, Type[IRenderer]]]*) – Default renderer to use globally. Calling this method without arguments will result in library default renderer SgrRenderer being set as default.

All the methods with the `renderer` argument (e.g., text.render()) will use the global default one if said argument is omitted or set to *None*.

You can specify either the renderer class, in which case manager will instantiate it with the default parameters, or provide already instantiated and set up renderer, which will be registered as global.

classmethod get_default()

Get global renderer instance (SgrRenderer, or the one provided earlier with set_default()).

Return type
IRenderer

class pytermor.renderer.**IRenderer**(*, *allow_cache=None*, *allow_format=None*)

Renderer interface.

property is_caching_allowed: bool

Returns
True if caching of renderer's results makes any sense and *False* otherwise.

property is_format_allowed: bool

Returns
True if renderer is set up to produce formatted output and will do it on invocation, and *False* otherwise.

abstract render(*string*, *fmt=None*)

Apply colors and attributes described in `fmt` argument to `string` and return the result. Output format depends on renderer's class, which defines the implementation.

Important: Renderer's method IRenderer.render() can work only with primitive *str* instances. IRenderable instances like Fragment or Text should be rendered using module-level function render() or their own instance method IRenderable.render().

Parameters

- **string** (*str*) – String to format.
- **fmt** (*Optional[FT]*) – Style or color to apply. If `fmt` is a `IColor` instance, it is assumed to be a foreground color. See FT.

Returns
String with formatting applied, or without it, depending on renderer settings.

Return type
str

clone(**args*, ***kwargs*)

Make a copy of the renderer with the same setup.

Return type
_T

class pytermor.renderer.**OutputMode**(*value*)

Bases: *ExtendedEnum*

Determines what types of SGR sequences are allowed to use in the output.

NO_ANSI = 'no_ansi'

The renderer discards all color and format information completely.

XTERM_16 = 'xterm_16'

16-colors mode. Enforces the renderer to approximate all color types to Color16 and render them as basic mode selection SGR sequences (`ESC [31m`, `ESC [42m` etc). See Color.approximate() for approximation algorithm details.

XTERM_256 = 'xterm_256'

256-colors mode. Allows the renderer to use either Color16 or Color256 (but RGB will be approximated to 256-color pallette).

TRUE_COLOR = 'true_color'

RGB color mode. Does not apply restrictions to color rendering.

AUTO = 'auto'

Lets the renderer select the most suitable mode by itself. See guide.output_mode_select for the details.

class pytermor.renderer.**SgrRenderer**(*output_mode=OutputMode.AUTO*, *io=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: *IRenderer*

Default renderer invoked by Text.render(). Transforms Color instances defined in `fmt` into ANSI control sequence bytes and merges them with input string. Type of resulting SequenceSGR depends on type of Color instances in `fmt` argument and current output mode of the renderer.

1. ColorRGB can be rendered as True Color sequence, 256-color sequence or 16-color sequence depending on specified OutputMode and Config.prefer_rgb.
2. Color256 can be rendered as 256-color sequence or 16-color sequence.
3. Color16 will be rendered as 16-color sequence.
4. Nothing of the above will happen and all formatting will be discarded completely if output device is not a terminal emulator or if the developer explicitly set up the renderer to do so (OutputMode.NO_ANSI).

Renderer approximates RGB colors to closest **indexed** colors if terminal doesn't support RGB output. In case terminal doesn't support even 256 colors, it falls back to 16-color palette and picks closest samples again the same way. See OutputMode documentation for exact mappings.

```
>>> SgrRenderer(OutputMode.XTERM_256).render('text', Styles.WARNING_LABEL)
'[1;33mtext[22;39m'
>>> SgrRenderer(OutputMode.NO_ANSI).render('text', Styles.WARNING_LABEL)
'text'
```

Detailed OutputMode.AUTO algorithm is described in guide.output_mode_select.

Cache allowed
True

Format allowed
False if output_mode is OutputMode.NO_ANSI, *True* otherwise.

Parameters

- **output_mode** (*str | OutputMode*) – can be set up explicitly, or kept at the default value OutputMode.AUTO; in the latter case the renderer will select the appropriate mode by itself (see guide.output_mode_select).
- **io** (*t.IO*) – specified in order to check if output device is a tty or not and can be omitted when output mode is set up explicitly.

render(*string*, *fmt=None*)

Apply colors and attributes described in `fmt` argument to `string` and return the result. Output format depends on renderer's class, which defines the implementation.

Important: Renderer's method IRenderer.render() can work only with primitive *str* instances. IRenderable instances like Fragment or Text should be rendered using module-level function render() or their own instance method IRenderable.render().

Parameters

- **string** (*str*) – String to format.
- **fmt** (*FT*) – Style or color to apply. If `fmt` is a `IColor` instance, it is assumed to be a foreground color. See FT.

Returns
String with formatting applied, or without it, depending on renderer settings.

Return type
str

`clone()`

Make a copy of the renderer with the same setup.

Return type
SgrRenderer

`property is_caching_allowed: bool`

Returns
True if caching of renderer's results makes any sense and *False* otherwise.

`property is_format_allowed: bool`

Returns
True if renderer is set up to produce formatted output and will do it on invocation, and *False* otherwise.

`class pytermor.renderer.TmuxRenderer`

Bases: *IRenderer*

Translates Styles <Style> attributes into tmux-compatible markup.[1]

```
>>> TmuxRenderer().render('text',  Style(fg='blue', bold=True))
'#[fg=blue bold]text#[fg=default nobold]'
```

Cache allowed
True

Format allowed
True, because tmux markup can be used without regard to the type of output device and its capabilities – all the dirty work will be done by the multiplexer himself.

`render(string, fmt=None)`

Apply colors and attributes described in `fmt` argument to `string` and return the result. Output format depends on renderer's class, which defines the implementation.

Important: Renderer's method IRenderer.render() can work only with primitive *str* instances. IRenderable instances like Fragment or Text should be rendered using module-level function render() or their own instance method IRenderable.render().

Parameters

- **string** (*str*) – String to format.
- **fmt** (*FT*) – Style or color to apply. If `fmt` is a `IColor` instance, it is assumed to be a foreground color. See FT.

Returns
String with formatting applied, or without it, depending on renderer settings.

[1] tmux is a commonly used terminal multiplexer.

Return type
str

clone(**args*, ***kwargs*)

Make a copy of the renderer with the same setup.

Return type
_T

property is_caching_allowed: bool

Returns
True if caching of renderer's results makes any sense and *False* otherwise.

property is_format_allowed: bool

Returns
True if renderer is set up to produce formatted output and will do it on invocation, and *False* otherwise.

class pytermor.renderer.**NoOpRenderer**

Bases: *IRenderer*

Special renderer type that does nothing with the input string and just returns it as is (i.e. raw text without any Styles<Style> applied. Often used as a default argument value (along with similar "NoOps" like NOOP_STYLE, NOOP_COLOR etc.)

```
>>> NoOpRenderer().render('text', Style(fg='green', bold=True))
'text'
```

Cache allowed
False

Format allowed
False, nothing to apply → nothing to allow.

render(*string*, *fmt=None*)

Return the string argument untouched, don't mind the fmt.

Parameters

- **string** (*str*) – String to format ignore.
- **fmt** (*Optional[FT]*) – Style or color to appl discard.

Return type
str

clone(**args*, ***kwargs*)

Make a copy of the renderer with the same setup.

Return type
_T

property is_caching_allowed: bool

Returns
True if caching of renderer's results makes any sense and *False* otherwise.

property is_format_allowed: bool

Returns
True if renderer is set up to produce formatted output and will do it on invocation, and *False* otherwise.

class pytermor.renderer.**HtmlRenderer**

Bases: *IRenderer*

Translate Styles <Style> attributes into a rudimentary HTML markup. All the formatting is inlined into `style` attribute of the elements. Can be optimized by extracting the common styles as CSS classes and referencing them by DOM elements instead.

```
>>> HtmlRenderer().render('text', Style(fg='red', bold=True))
'<span style="color: #800000; font-weight: 700">text</span>'
```

Cache allowed
True

Format allowed
True, because the capabilities of the terminal have nothing to do with HTML markup meant for web-browsers.

render(*string*, *fmt=None*)

Apply colors and attributes described in `fmt` argument to `string` and return the result. Output format depends on renderer's class, which defines the implementation.

Important: Renderer's method IRenderer.render() can work only with primitive *str* instances. IRenderable instances like Fragment or Text should be rendered using module-level function render() or their own instance method IRenderable.render().

Parameters

- **string** (*str*) – String to format.
- **fmt** (*FT*) – Style or color to apply. If `fmt` is a `IColor` instance, it is assumed to be a foreground color. See FT.

Returns
String with formatting applied, or without it, depending on renderer settings.

Return type
str

clone(**args*, ***kwargs*)

Make a copy of the renderer with the same setup.

Return type
_T

property is_caching_allowed: bool

Returns
True if caching of renderer's results makes any sense and *False* otherwise.

property is_format_allowed: bool

Returns
True if renderer is set up to produce formatted output and will do it on invocation, and *False* otherwise.

class pytermor.renderer.**SgrDebugger**(*output_mode=OutputMode.AUTO*)

Bases: *SgrRenderer*

Subclass of regular SgrRenderer with two differences – instead of rendering the proper ANSI escape sequences it renders them with `ESC` character replaced by "", and encloses the whole sequence into '()' for visual separation.

Can be used for debugging of assembled sequences, because such a transformation reliably converts a control sequence into a harmless piece of bytes completely ignored by the terminals.

```
>>> SgrDebugger(OutputMode.XTERM_16).render('text', Style(fg='red', bold=True))
'([1;31m)text([22;39m)'
```

Cache allowed
True

Format allowed
adjustable

property is_format_allowed: bool

Returns
True if renderer is set up to produce formatted output and will do it on invocation, and *False* otherwise.

render(*string*, *fmt=None*)

Apply colors and attributes described in `fmt` argument to `string` and return the result. Output format depends on renderer's class, which defines the implementation.

Important: Renderer's method IRenderer.render() can work only with primitive *str* instances. IRenderable instances like Fragment or Text should be rendered using module-level function render() or their own instance method IRenderable.render().

Parameters

- **string** (*str*) – String to format.
- **fmt** (*FT*) – Style or color to apply. If `fmt` is a `IColor` instance, it is assumed to be a foreground color. See FT.

Returns
String with formatting applied, or without it, depending on renderer settings.

Return type
str

clone()

Make a copy of the renderer with the same setup.

Return type
SgrDebugger

set_format_always()

Force all control sequences to be present in the output.

property is_caching_allowed: bool

Returns
True if caching of renderer's results makes any sense and *False* otherwise.

set_format_auto()

Reset the force formatting flag and let the renderer decide by itself (see SgrRenderer docs for the details).

set_format_never()

Force disabling of all output formatting.

pytermor.renderer.**force_ansi_rendering**()

Shortcut for forcing all control sequences to be present in the output of a global renderer.

Note that it applies only to the renderer that is set up as default at the moment of calling this method, i.e., all previously created instances, as well as the ones that will be created afterwards, are unaffected.

pytermor.renderer.**force_no_ansi_rendering**()

Shortcut for disabling all output formatting of a global renderer.

7.11 pytermor.style

Reusable data classes that control the appearance of the output – colors (text/background/underline) and attributes (*bold*, *underlined*, *italic*, etc.). Instances can inherit attributes from each other, which allows to avoid meaningless definition repetitions; multiple inheritance is also supported.

Module Attributes

NOOP_STYLE	Special style passing the text through without any modifications.

Functions

is_ft(arg)

make_style([fmt])	General *Style* constructor.
merge_styles([origin, fallbacks, overwrites])	Bulk style merging method.

Classes

FrozenStyle(*args, **kwargs)

MergeMode(value)	An enumeration.
Style([fallback, fg, bg, frozen, bold, dim, ...])	Create new text render descriptior.
Styles()	Some ready-to-use styles which also can be used as examples.

class pytermor.style.**MergeMode**(*value*)

Bases: str, Enum

An enumeration.

class pytermor.style.**Style**(*fallback=None*, *fg=None*, *bg=None*, *frozen=False*, *, *bold=None*, *dim=None*, *italic=None*, *underlined=None*, *overlined=None*, *crosslined=None*, *double_underlined=None*, *curly_underlined=None*, *underline_color=None*, *inversed=None*, *blink=None*, *framed=None*, *class_name=None*)

Create new text render descriptior.

Both fg and bg can be specified as existing Color instance as well as plain *str* or *int* (for the details see resolve_color()).

```
>>> Style(fg='green', bold=True)
<Style[green +BOLD]>
>>> Style(bg=0x0000ff)
<Style[|#0000ff]>
>>> Style(fg='DeepSkyBlue1', bg='gray3')
<Style[x39|x232]>
```

Attribute merging from `fallback` works this way:

- If constructor argument is *not* empty (*True*, *False*, `Color` etc.), keep it as attribute value.
- If constructor argument is empty (*None*, `NOOP_COLOR`), take the value from `fallback`'s corresponding attribute.

See merge_fallback() and merge_overwrite() methods and take the differences into account. The method used in the constructor is the first one.

Important: Both empty (i.e., *None*) attributes of type `Color` after initialization will be replaced with special constant NOOP_COLOR, which behaves like there was no color defined, and at the same time makes it safer to work with nullable color-type variables. Merge methods are aware of this and trear NOOP_COLOR as *None*.

Important: *None* and NOOP_COLOR are always treated as placeholders for fallback values, i.e., they can't be used as *resetters* – that's what DEFAULT_COLOR is for.

Parameters

- **fallback** (`Style`) – Copy empty attributes from speicifed fallback style. See merge_fallback().
- **fg** (*CDT* | `RenderColor`) – Foreground (=text) color.
- **bg** (*CDT* | `RenderColor`) – Background color.
- **frozen** (*bool*) – Set to *True* to make an immutable instance.
- **bold** (*bool*) – Bold or increased intensity.
- **dim** (*bool*) – Faint, decreased intensity.
- **italic** (*bool*) – Italic.
- **underlined** (*bool*) – Underline.
- **overlined** (*bool*) – Overline.
- **crosslined** (*bool*) – Strikethrough.
- **double_underlined** (*bool*) – Double underline.
- **curly_underlined** (*bool*) – Curly underline.
- **underline_color** (*CDT* | `RenderColor`) – Underline color, if applicable.
- **inversed** (*bool*) – Swap foreground and background colors.
- **blink** (*bool*) – Blinking effect.
- **framed** (*bool*) – Enclosed in a rectangle border.
- **class_name** (*str*) – Custom class name for the element.

property fg: *RenderColor*

Foreground (i.e., text) color. Can be set as CDT or `Color`, stored always as `Color`.

property bg: *RenderColor*

Background color. Can be set as CDT or `Color`, stored always as `Color`.

property underline_color: *RenderColor*

Underline color. Can be set as CDT or `Color`, stored always as `Color`.

bold: bool

Bold or increased intensity (depending on terminal settings).

dim: bool

Faint, decreased intensity.

Terminal-based rendering

Terminals apply this effect to foreground (=text) color, but when it's used together with inversed, they usually make the background darker instead.

Also note that usually it affects indexed colors only and has no effect on RGB-based ones (True Color mode).

---

italic: bool

Italic (some terminals may display it as inversed instead).

underlined: bool

Underline.

overlined: bool

Overline.

crosslined: bool

Strikethrough.

double_underlined: bool

Double underline.

curly_underlined: bool

Curly underline.

inversed: bool

Swap foreground and background colors. When inversed effect is active, changing the background color will actually change the text color, and vice versa.

blink: bool

Blinking effect. Supported by a limited set of renderers <IRenderer>.

framed: bool

Add a rectangular border around the text; the border color is equal to the text color. Supported by a limited set of renderers <IRenderer> and (even more) limited amount of terminal emulators.

class_name: str

Arbitary string used by some renderers <IRenderer>, e.g. by ` HtmlRenderer`, which will include the value of this property to an output element class list. This property is not inheritable.

clone(*frozen=False*)

Make a copy of the instance. Note that a copy is mutable by default even if an original was frozen.

Parameters
frozen – Set to *True* to make an immutable instance.

Return type
Style

autopick_fg()

Pick `fg_color` depending on `bg_color`. Set `fg_color` to either 3% gray (almost black) if background is bright, or to 80% gray (bright gray) if it is dark. If background is None, do nothing.

Todo: check if there is a better algorithm, because current thinks text on #000080 should be black

Modifies the instance in-place and returns it as well (for chained calls).

Return type
Style

flip()

Swap foreground color and background color. Modifies the instance in-place and returns it as well (for chained calls).

Return type
Style

merge(*mode*, *other*)

Method that allows specifying merging mode as an argument. Initially designed for template substitutions done by TemplateEngine. Invokes either of these (depending on `mode` value):

- merge_fallback()
- merge_overwrite()
- merge_replace()

Parameters

- **mode** (MergeMode) – Merge mode to use.
- **other** (Style) – Style to merge the attributes with.

Return type
Style

merge_fallback(*fallback*)

Merge current style with specified `fallback` style <Style>, following the rules:

1. `self` attribute value is in priority, i.e. when both `self` and `fallback` attributes are defined, keep `self` value.
2. If `self` attribute is *None*, take the value from `fallback`'s corresponding attribute, and vice versa.
3. If both attribute values are *None*, keep the *None*.

All attributes corresponding to constructor arguments except `fallback` are subject to merging. NOOP_COLOR is treated like *None* (default for fg and bg).

Modifies the instance in-place and returns it as well (for chained calls).

Listing 9: Merging different values in fallback mode

```
          FALLBACK    BASE(SELF)    RESULT
          +-------+   +------+   +------+
ATTR-1    | False --Ø | True ===>| True |  BASE val is in priority
ATTR-2    | True -----| None |-->| True |  no BASE val, taking FALLBACK val
ATTR-3    | None  |   | True ===>| True |  BASE val is in priority
ATTR-4    | None  |   | None |   | None |  no vals, keeping unset
          +-------+   +------+   +------+
```

See also:

merge_styles for the examples.

Parameters
fallback (Style) – Style to merge the attributes with.

Return type
Style

merge_overwrite(*overwrite*)

Merge current style with specified `overwrite` style <Style>, following the rules:

1. `overwrite` attribute value is in priority, i.e. when both `self` and `overwrite` attributes are defined, replace `self` value with `overwrite` one (in contrast to merge_fallback(), which works the opposite way).
2. If `self` attribute is *None*, take the value from `overwrite`'s corresponding attribute, and vice versa.
3. If both attribute values are *None*, keep the *None*.

All attributes corresponding to constructor arguments except `fallback` are subject to merging. NOOP_COLOR is treated like *None* (default for fg and bg).

Modifies the instance in-place and returns it as well (for chained calls).

Listing 10: Merging different values in overwrite mode

```
        BASE(SELF)  OVERWRITE    RESULT
          +------+   +-------+   +-------+
ATTR-1    | True ==Ø | False --->| False |  OVERWRITE val is in priority
ATTR-2    | None |   | True ---->| True  |  OVERWRITE val is in priority
ATTR-3    | True ====| None  |==>| True  |  no OVERWRITE val, keeping BASE val
ATTR-4    | None |   | None  |   | None  |  no vals, keeping unset
          +------+   +-------+   +-------+
```

See also:

merge_styles for the examples.

Parameters
overwrite (Style) – Style to merge the attributes with.

Return type
Style

merge_replace(*replacement*)

Not an actual "merge": discard all the attributes of the current instance and replace them with the values from replacement. Generally speaking, it makes sense only in TemplateEngine context, as style management using the template tags is quite limited, while there are far more elegant ways to do the same from the regular python code.

Modifies the instance in-place and returns it as well (for chained calls).

Listing 11: Merging different values in replace mode

```
        BASE(SELF)   REPLACE     RESULT
          +------+   +-------+  +-------+
ATTR-1    | False =Ø | True --->| True  |  REPLACE val is in priority
ATTR-2    | True ==Ø | False -->| False |  REPLACE val is in priority
ATTR-3    | None |   | False -->| False |  REPLACE val is in priority
ATTR-4    | True ==Ø | None --->| None  |   ... even when it is unset
          +------+   +-------+  +-------+
```

Parameters
replacement (Style) – Style to merge the attributes with.

Return type
Style

class pytermor.style.**FrozenStyle**(**args*, ***kwargs*)

Bases: *Style*

autopick_fg()

Pick `fg_color` depending on `bg_color`. Set `fg_color` to either 3% gray (almost black) if background is bright, or to 80% gray (bright gray) if it is dark. If background is None, do nothing.

Todo: check if there is a better algorithm, because current thinks text on #000080 should be black

Modifies the instance in-place and returns it as well (for chained calls).

Return type
Style

property bg: *RenderColor*

Background color. Can be set as CDT or `Color`, stored always as `Color`.

clone(*frozen=False*)

Make a copy of the instance. Note that a copy is mutable by default even if an original was frozen.

Parameters
frozen – Set to *True* to make an immutable instance.

Return type
Style

property fg: *RenderColor*

Foreground (i.e., text) color. Can be set as CDT or `Color`, stored always as `Color`.

flip()

Swap foreground color and background color. Modifies the instance in-place and returns it as well (for chained calls).

Return type
Style

merge(*mode*, *other*)

Method that allows specifying merging mode as an argument. Initially designed for template substitutions done by TemplateEngine. Invokes either of these (depending on `mode` value):

- merge_fallback()
- merge_overwrite()
- merge_replace()

Parameters

- **mode** (`MergeMode`) – Merge mode to use.
- **other** (`Style`) – Style to merge the attributes with.

Return type
Style

merge_fallback(*fallback*)

Merge current style with specified `fallback` style <Style>, following the rules:

1. `self` attribute value is in priority, i.e. when both `self` and `fallback` attributes are defined, keep `self` value.

2. If `self` attribute is *None*, take the value from `fallback`'s corresponding attribute, and vice versa.
3. If both attribute values are *None*, keep the *None*.

All attributes corresponding to constructor arguments except `fallback` are subject to merging. NOOP_COLOR is treated like *None* (default for fg and bg).

Modifies the instance in-place and returns it as well (for chained calls).

Listing 12: Merging different values in fallback mode

```
         FALLBACK   BASE(SELF)   RESULT
         +-------+   +------+   +------+
ATTR-1   | False --Ø | True ===>| True |  BASE val is in priority
ATTR-2   | True -----| None |-->| True |  no BASE val, taking FALLBACK val
ATTR-3   | None  |   | True ===>| True |  BASE val is in priority
ATTR-4   | None  |   | None |   | None |  no vals, keeping unset
         +-------+   +------+   +------+
```

See also:

merge_styles for the examples.

Parameters
fallback (Style) – Style to merge the attributes with.

Return type
Style

merge_overwrite(*overwrite*)

Merge current style with specified `overwrite` style <Style>, following the rules:

1. `overwrite` attribute value is in priority, i.e. when both `self` and `overwrite` attributes are defined, replace `self` value with `overwrite` one (in contrast to merge_fallback(), which works the opposite way).
2. If `self` attribute is *None*, take the value from `overwrite`'s corresponding attribute, and vice versa.
3. If both attribute values are *None*, keep the *None*.

All attributes corresponding to constructor arguments except `fallback` are subject to merging. NOOP_COLOR is treated like *None* (default for fg and bg).

Modifies the instance in-place and returns it as well (for chained calls).

Listing 13: Merging different values in overwrite mode

```
        BASE(SELF)  OVERWRITE    RESULT
          +------+   +-------+   +-------+
ATTR-1    | True ==Ø | False --->| False |  OVERWRITE val is in priority
ATTR-2    | None |   | True ---->| True  |  OVERWRITE val is in priority
ATTR-3    | True ====| None  |==>| True  |  no OVERWRITE val, keeping BASE val
ATTR-4    | None |   | None  |   | None  |  no vals, keeping unset
          +------+   +-------+   +-------+
```

See also:

merge_styles for the examples.

Parameters
overwrite (Style) – Style to merge the attributes with.

Return type
Style

merge_replace(*replacement*)

Not an actual "merge": discard all the attributes of the current instance and replace them with the values from replacement. Generally speaking, it makes sense only in TemplateEngine context, as style management using the template tags is quite limited, while there are far more elegant ways to do the same from the regular python code.

Modifies the instance in-place and returns it as well (for chained calls).

Listing 14: Merging different values in replace mode

```
        BASE(SELF)   REPLACE     RESULT
         +------+   +-------+  +-------+
ATTR-1   | False =Ø | True --->| True  |  REPLACE val is in priority
ATTR-2   | True ==Ø | False -->| False |  REPLACE val is in priority
ATTR-3   | None |   | False -->| False |  REPLACE val is in priority
ATTR-4   | True ==Ø | None --->| None  |   ... even when it is unset
         +------+   +-------+  +-------+
```

Parameters
replacement (Style) – Style to merge the attributes with.

Return type
Style

property underline_color: *RenderColor*

Underline color. Can be set as CDT or Color, stored always as Color.

bold: **bool**

Bold or increased intensity (depending on terminal settings).

dim: **bool**

Faint, decreased intensity.

Terminal-based rendering

Terminals apply this effect to foreground (=text) color, but when it's used together with inversed, they usually make the background darker instead.

Also note that usually it affects indexed colors only and has no effect on RGB-based ones (True Color mode).

---

italic: **bool**

Italic (some terminals may display it as inversed instead).

underlined: **bool**

Underline.

overlined: **bool**

Overline.

crosslined: **bool**

Strikethrough.

double_underlined: **bool**

Double underline.

curly_underlined: **bool**

Curly underline.

inversed: bool

Swap foreground and background colors. When inversed effect is active, changing the background color will actually change the text color, and vice versa.

blink: bool

Blinking effect. Supported by a limited set of renderers <IRenderer>.

framed: bool

Add a rectangular border around the text; the border color is equal to the text color. Supported by a limited set of renderers <IRenderer> and (even more) limited amount of terminal emulators.

class_name: str

Arbitary string used by some renderers <IRenderer>, e.g. by ` HtmlRenderer`, which will include the value of this property to an output element class list. This property is not inheritable.

pytermor.style.**NOOP_STYLE = <*_NoOpStyle[]>**

Special style passing the text through without any modifications.

Important: Casting to *bool* results in **False** for all NOOP instances in the library (NOOP_SEQ, NOOP_COLOR and NOOP_STYLE). This is intended.

This class is immutable, i.e. LogicError will be raised upon an attempt to modify any of its attributes, which could potentially lead to schrödinbugs:

```
st1.merge_fallback(Style(bold=True), [Style(italic=False)])
```

If st1 is a regular style instance, it's safe to call self-modifying methods, but if it happens to be a NOOP_STYLE, the statement could have been alter the internal state of the style, which is referenced all over the library, which could lead to the changes appearing in an unexpected places.

To be safe from this outcome one could merge styles via frontend method merge_styles, which always makes a copy of `origin` argument and thus cannot lead to such results.

class pytermor.style.**Styles**

Some ready-to-use styles which also can be used as examples. All instances are immutable.

WARNING = <*Style[yellow]>

WARNING_LABEL = <*Style[yellow +BOLD]>

WARNING_ACCENT = <*Style[hi-yellow]>

ERROR = <*Style[red]>

ERROR_LABEL = <*Style[red +BOLD]>

ERROR_ACCENT = <*Style[hi-red]>

CRITICAL = <*Style[hi-white|x160]>

CRITICAL_LABEL = <*Style[hi-white|x160 +BOLD]>

CRITICAL_ACCENT = <*Style[hi-white|x160 +BLIN +BOLD]>

INCONSISTENCY = <*Style[hi-yellow|x160]>

pytermor.style.**make_style**(*fmt=None*)

General *Style* constructor. Accepts a variety of argument types:

- **CDT (*str* or *int*)**
  This argument type implies the creation of basic *Style* with the only attribute set being fg (i.e., text color). For the details on color resolving see resolve_color().

- *Style*
  Existing style instance. Return it as is.
- *None*
  Return NOOP_STYLE.

Parameters
fmt (*FT*) – See FT.

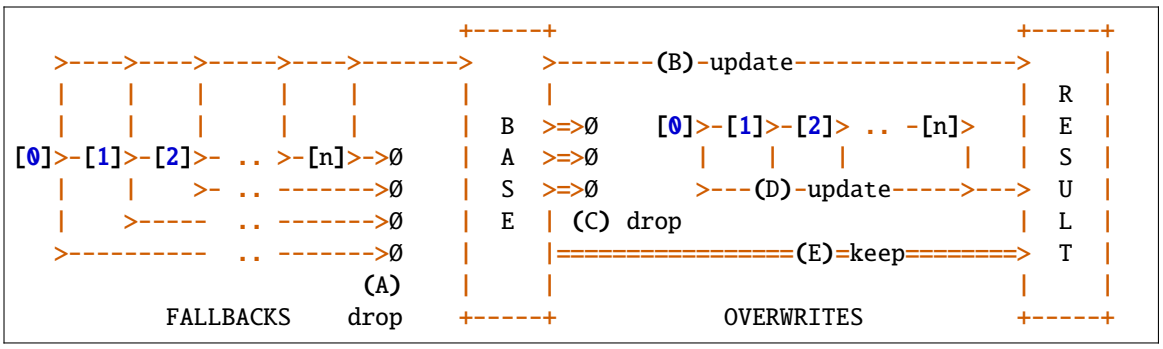
Return type
Style

pytermor.style.**merge_styles**(*origin=<*_NoOpStyle[]>*, *, *fallbacks=()*, *overwrites=()*)

Bulk style merging method. First merge fallbacks styles <.Style> with the `origin` in the same order they are iterated, using merge_fallback() algorithm; then do the same for overwrites styles, but using merge_overwrite() merge method.

Important: The original origin is left untouched, as all the operations are performed on its clone. To make things clearer the name of the argument differs from the ones that are modified in-place (`base` and `origin`).

Listing 15: Dual mode merge diagram



The key actions are marked with (**A**) to (**E**) letters. In reality the algorithm works in slightly different order, but the exact scheme would be less illustrative.

(A),(B)
Iterate `fallback` styles one by one; discard all the attributes of a current `fallback` style, that are already set in `origin` style (i.e., that are not *Nones*). Update all `origin` style empty attributes with corresponding `fallback` values, if they exist and are not empty. Repeat these steps for the next `fallback` in the list, until the list is empty.

Listing 16: Fallback merge algorithm example №1

```
>>> origin = Style(fg='red')
...
>>> fallbacks = [Style(fg='blue'), Style(bold=True),␣
↪Style(bold=False)]
...
>>> merge_styles(origin, fallbacks=fallbacks)
<Style[red +BOLD]>
```

In the example above:

- the first fallback will be ignored, as fg is already set;
- the second fallback will be applied (`origin` style will now have bold set to *True*;
- which will make the handler ignore third fallback completely; if third fallback was encountered earlier than the 2nd one, `origin` bold attribute would have been set to *False*,

but alas.

Note: Fallbacks allow to build complex style conditions, e.g. take a look into Highlighter.colorize() method:

```
int_st = merge_styles(st, fallbacks=[Style(bold=True)])
```

Instead of using `Style(st, bold=True)` the merging algorithm is invoked. This changes the logic of "bold" attribute application – if there is a necessity to explicitly forbid bold text at origin/parent level, one could write:

```
STYLE_NUL = Style(STYLE_DEFAULT, cv.GRAY, bold=False)
STYLE_PRC = Style(STYLE_DEFAULT, cv.MAGENTA)
STYLE_KIL = Style(STYLE_DEFAULT, cv.BLUE)
...
```

As you can see, resulting `int_st` will be bold for all styles other than `STYLE_NUL`.

Listing 17: Fallback merge algorithm example №2

```
>>> merge_styles(Style(fg=cv.BLUE), fallbacks=[Style(bold=True)])
<Style[blue +BOLD]>
>>> merge_styles(Style(fg=cv.GRAY, bold=False),␣
↪fallbacks=[Style(bold=True)])
<Style[gray -BOLD]>
```

(C),(D),(E)

Iterate `overwrite` styles one by one; discard all the attributes of a `origin` style that have a non-empty counterpart in `overwrite` style, and put corresponding `overwrite` attribute values instead of them. Keep `origin` attribute values that have no counterpart in current `overwrite` style (i.e., if attribute value is *None*). Then pick next `overwrite` style from the input list and repeat all these steps.

Listing 18: Overwrite merge algorithm example

```
>>> origin = Style(fg='red')
...
>>> overwrites = [Style(fg='blue'), Style(bold=True),␣
↪Style(bold=False)]
...
>>> merge_styles(origin, overwrites=overwrites)
<Style[blue -BOLD]>
```

In the example above all the `overwrites` will be applied in order they were put into *list*, and the result attribute values are equal to the last encountered non-empty values in `overwrites` list.

Parameters

- **origin** (Style) – Initial style, or the source of attributes.
- **fallbacks** (*Iterable*[Style]) – List of styles to be used as a backup attribute storage, or. in other words, to be "merged up" with the origin; affects the unset attributes of the current style and replaces these values with its own. Uses merge_fallback() merging strategy.
- **overwrites** (*Iterable*[Style]) – List of styles to be used as attribute storage force override regardless of actual origin attribute valuse (so called "merging down" with the origin).

Returns
Clone of `origin` style with all specified styles merged into.

Return type
Style

7.12 pytermor.template

Functions

`render`(tpl, renderer)	
`substitute`(tpl)	

Classes

`TemplateEngine`([custom_styles, global_style])	

7.13 pytermor.term

A

Module Attributes

RCP_REGEX	Regular expression for RCP (Report Cursor Position) sequence parsing.

Functions

compose_clear_line_fill_bg(basis[, line, column])	**param basis**
compose_hyperlink(url[, label])	Syntax: `(OSC 8 ; ;) (url) (ST) (label) (OSC 8 ; ;) (ST)`, where OSC <SequenceOSC> is `ESC ]`.
confirm([attempts, default, keymap, prompt, ...])	Ensure the next action is manually confirmed by user.
decompose_report_cursor_position(string)	Parse RCP sequence that usually comes from a terminal as a response to QCP <make_query_cursor_position> sequence and contains a cursor's current line and column.
get_char_width(char, block)	General-purpose method for getting width of a character in terminal columns.
get_preferable_wrap_width([force_width])	Return preferable terminal width for comfort reading of wrapped text (max=120).

continues on next page

Table 2 – continued from previous page

get_terminal_width([fallback, pad])	Return current terminal width with an optional "safety buffer", which ensures that no unwanted line wrapping will happen.
guess_char_width(c)	Determine how many columns are needed to display a character in a terminal.
make_clear_display()	Create ED sequence that clears an entire screen.
make_clear_display_after_cursor()	Create ED sequence that clears a part of the screen from cursor to the end of the screen.
make_clear_display_before_cursor()	Create ED sequence that clears a part of the screen from cursor to the beginning of the screen.
make_clear_history()	Create ED sequence that clears history, i.e., invisible lines on the top that can be scrolled back down.
make_clear_line()	Create EL sequence that clears an entire line at the cursor position.
make_clear_line_after_cursor()	Create EL sequence that clears a part of the line from cursor to the end of the same line.
make_clear_line_before_cursor()	Create EL sequence that clears a part of the line from cursor to the beginning of the same line.
make_color_256(code[, target])	Wrapper for creation of SequenceSGR that sets foreground (or background) to one of 256-color palette value.:
make_color_rgb(r, g, b[, target])	Wrapper for creation of SequenceSGR operating in True Color mode (16M). Valid values for `r`, `g` and `b` are in range of [0; 255]. This range linearly translates into [0x00; 0xFF] for each channel. The result value is composed as "#RRGGBB". For example, a sequence with color of #ff3300 can be created with::.
make_disable_alt_screen_buffer()	C
make_enable_alt_screen_buffer()	C
make_erase_in_display([mode])	Create ED sequence that clears a part of the screen or the entire screen.
make_erase_in_line([mode])	Create EL sequence that clears a part of the line or the entire line at the cursor position.
make_hide_cursor()	C
make_hyperlink()	Create a hyperlink in the text *(supported by limited amount of terminals)*.
make_move_cursor_down([lines])	Create CUD (Cursor Down) sequence that moves the cursor down by specified amount of lines.
make_move_cursor_down_to_start([lines])	Create CNL (Cursor Next Line) sequence that moves the cursor to the beginning of the line and down by specified amount of lines.
make_move_cursor_left([columns])	Create CUB (Cursor Back) sequence that moves the cursor left by specified amount of columns.
make_move_cursor_right([columns])	Create CUF (Cursor Forward) sequence that moves the cursor right by specified amount of columns.
make_move_cursor_up([lines])	Create CUU (Cursor Up) sequence that moves the cursor up by specified amount of lines.
make_move_cursor_up_to_start([lines])	Create CPL (Cursor Previous Line) sequence that moves the cursor to the beginning of the line and up by specified amount of lines.
make_query_cursor_position()	Create QCP (Query Cursor Position) sequence that requests an output device to respond with a structure containing current cursor coordinates (RCP <decompose_request_cursor_position()>).

continues on next page

Table 2 – continued from previous page

make_reset_cursor()	Create CUP sequence without params, which moves the cursor to top left corner of the screen.
make_restore_cursor_position()	**example** ESC 8
make_restore_screen()	C
make_save_cursor_position()	**example** ESC 7
make_save_screen()	C
make_set_cursor([line, column])	Create CUP sequence that moves the cursor to specified amount line and column.
make_set_cursor_column([column])	Create CHA (Cursor Character Absolute) sequence that sets cursor horizontal position to column.
make_set_cursor_line([line])	Create VPA (Vertical Position Absolute) sequence that sets cursor vertical position to line.
make_show_cursor()	C
measure_char_width(char[, clear_after])	Low-level function that returns the exact character width in terminal columns.
wait_key([block])	Wait for a key press on the console and return it.

pytermor.term.**RCP_REGEX**

Regular expression for RCP sequence parsing. See decompose_report_cursor_position().

pytermor.term.**make_color_256**(*code*, *target=ColorTarget.FG*)

Wrapper for creation of SequenceSGR that sets foreground (or background) to one of 256-color palette value.:

```
>>> make_color_256(141)
<SGR[38;5;141m]>
```

See also:

Color256 class.

Parameters

- **code** (*int*) – Index of the color in the palette, 0 – 255.
- **target** (ColorTarget) –

Example
ESC [38;5;141m

Return type
SequenceSGR

pytermor.term.**make_color_rgb**(*r*, *g*, *b*, *target=ColorTarget.FG*)

Wrapper for creation of SequenceSGR operating in True Color mode (16M). Valid values for r, g and b are in range of [0; 255]. This range linearly translates into [0x00; 0xFF] for each channel. The result value is composed as "#RRGGBB". For example, a sequence with color of #ff3300 can be created with:

```
>>> make_color_rgb(255, 51, 0)
<SGR[38;2;255;51;0m]>
```

See also:

ColorRGB class.

Parameters

- **r** (*int*) – Red channel value, 0 – 255.
- **g** (*int*) – Blue channel value, 0 – 255.
- **b** (*int*) – Green channel value, 0 – 255.
- **target** (ColorTarget) –

Example
ESC [38;2;255;51;0m

Return type
SequenceSGR

pytermor.term.**make_reset_cursor**()

Create CUP sequence without params, which moves the cursor to top left corner of the screen. See make_set_cursor().

Example
ESC [H

Return type
SequenceCSI

pytermor.term.**make_set_cursor**(*line=1*, *column=1*)

Create CUP sequence that moves the cursor to specified amount line and column. The values are 1-based, i.e. (1; 1) is top left corner of the screen.

Note: Both sequence params are optional and defaults to 1 if omitted, e.g. ESC [;3H is effectively ESC [1;3H, and ESC [4H is the same as ESC [4;H or ESC [4;1H.

Example
ESC [9;15H

Return type
SequenceCSI

pytermor.term.**make_move_cursor_up**(*lines=1*)

Create CUU sequence that moves the cursor up by specified amount of lines. If the cursor is already at the top of the screen, this has no effect.

Example
ESC [2A

Return type
SequenceCSI

pytermor.term.**make_move_cursor_down**(*lines=1*)

Create CUD sequence that moves the cursor down by specified amount of lines. If the cursor is already at the bottom of the screen, this has no effect.

Example
ESC [3B

Return type
SequenceCSI

pytermor.term.**make_move_cursor_left**(*columns=1*)

Create CUB sequence that moves the cursor left by specified amount of columns. If the cursor is already at the left edge of the screen, this has no effect.

Example
ESC [4D

Return type
SequenceCSI

pytermor.term.**make_move_cursor_right**(*columns=1*)

Create CUF sequence that moves the cursor right by specified amount of columns. If the cursor is already at the right edge of the screen, this has no effect.

Example
ESC [5C

Return type
SequenceCSI

pytermor.term.**make_move_cursor_up_to_start**(*lines=1*)

Create CPL sequence that moves the cursor to the beginning of the line and up by specified amount of lines.

Example
ESC [2F

Return type
SequenceCSI

pytermor.term.**make_move_cursor_down_to_start**(*lines=1*)

Create CNL sequence that moves the cursor to the beginning of the line and down by specified amount of lines.

Example
ESC [3E

Return type
SequenceCSI

pytermor.term.**make_set_cursor_line**(*line=1*)

Create VPA sequence that sets cursor vertical position to line.

Example
ESC [9d

Return type
SequenceCSI

pytermor.term.**make_set_cursor_column**(*column=1*)

Create CHA sequence that sets cursor horizontal position to column.

Parameters
column (*int*) – New cursor horizontal position.

Example
ESC [15G

Return type
SequenceCSI

pytermor.term.**make_query_cursor_position**()

Create QCP sequence that requests an output device to respond with a structure containing current cursor coordinates (RCP <decompose_request_cursor_position()>).

> **Warning:** Sending this sequence to the terminal may **block** infinitely. Consider using a thread or set a timeout for the main thread using a signal.

Example
ESC [6n

Return type
SequenceCSI

pytermor.term.**make_erase_in_display**(*mode=0*)

Create ED sequence that clears a part of the screen or the entire screen. Cursor position does not change.

Parameters
mode (*int*) – Sequence operating mode.

- If set to 0, clear from cursor to the end of the screen.
- If set to 1, clear from cursor to the beginning of the screen.
- If set to 2, clear the entire screen.
- If set to 3, clear terminal history (xterm only).

Example
ESC [0J

Return type
SequenceCSI

pytermor.term.**make_clear_display_after_cursor**()

Create ED sequence that clears a part of the screen from cursor to the end of the screen. Cursor position does not change.

Example
ESC [0J

Return type
SequenceCSI

pytermor.term.**make_clear_display_before_cursor**()

Create ED sequence that clears a part of the screen from cursor to the beginning of the screen. Cursor position does not change.

Example
ESC [1J

Return type
SequenceCSI

pytermor.term.**make_clear_display**()

Create ED sequence that clears an entire screen. Cursor position does not change.

Example
ESC [2J

Return type
SequenceCSI

pytermor.term.**make_clear_history**()

Create ED sequence that clears history, i.e., invisible lines on the top that can be scrolled back down. Cursor position does not change. This is a xterm extension.

Example
ESC [3J

Return type
SequenceCSI

pytermor.term.**make_erase_in_line**(*mode=0*)

Create EL sequence that clears a part of the line or the entire line at the cursor position. Cursor position does not change.

Parameters
mode (*int*) – Sequence operating mode.

- If set to 0, clear from cursor to the end of the line.
- If set to 1, clear from cursor to the beginning of the line.
- If set to 2, clear the entire line.

Example
`ESC [0K`

Return type
SequenceCSI

pytermor.term.**make_clear_line_after_cursor**()

Create EL sequence that clears a part of the line from cursor to the end of the same line. Cursor position does not change.

Example
`ESC [0K`

Return type
SequenceCSI

pytermor.term.**make_clear_line_before_cursor**()

Create EL sequence that clears a part of the line from cursor to the beginning of the same line. Cursor position does not change.

Example
`ESC [1K`

Return type
SequenceCSI

pytermor.term.**make_clear_line**()

Create EL sequence that clears an entire line at the cursor position. Cursor position does not change.

Example
`ESC [2K`

Return type
SequenceCSI

pytermor.term.**make_show_cursor**()

C

Return type
SequenceCSI

pytermor.term.**make_hide_cursor**()

C

Return type
SequenceCSI

pytermor.term.**make_save_screen**()

C

Return type
SequenceCSI

pytermor.term.**make_restore_screen**()

C

Return type
SequenceCSI

pytermor.term.**make_enable_alt_screen_buffer**()

C

Return type
SequenceCSI

pytermor.term.**make_disable_alt_screen_buffer**()

C

Return type
SequenceCSI

pytermor.term.**make_hyperlink**()

Create a hyperlink in the text *(supported by limited amount of terminals)*. Note that a complete set of commands to define a hyperlink consists of 4 oh them (two OSC-8 <SequenceOSC> and two ST <SequenceST>).

See also:

compose_hyperlink()`.

Return type
SequenceOSC

pytermor.term.**make_save_cursor_position**()

Example
`ESC 7`

Return type
SequenceFp

pytermor.term.**make_restore_cursor_position**()

Example
`ESC 8`

Return type
SequenceFp

pytermor.term.**compose_clear_line_fill_bg**(*basis*, *line=None*, *column=None*)

Parameters

- **basis** (SequenceSGR) –
- **line** (*Optional[int]*) –
- **column** (*Optional[int]*) –

Return type
str

pytermor.term.**compose_hyperlink**(*url*, *label=None*)

Syntax: `(OSC 8 ; ;) (url) (ST) (label) (OSC 8 ; ;) (ST)`, where OSC <SequenceOSC> is `ESC ]`.

Parameters

- **url** (*str*) –
- **label** (*Optional[str]*) –

Example
`ESC ]8;;http://localhost ESC \Text ESC ]8;; ESC \`

Return type
str

pytermor.term.**decompose_report_cursor_position**(*string*)

Parse RCP sequence that usually comes from a terminal as a response to QCP <make_query_cursor_position> sequence and contains a cursor's current line and column.

Todo: make a separate Seq class for this?

```
>>> decompose_report_cursor_position('[9;15R')
(9, 15)
```

Parameters
string (*str*) – Terminal response with a sequence.

Returns
Current line and column if the expected sequence exists in `string`, *None* otherwise.

Return type
Optional[*Tuple*[int, int]]

pytermor.term.**get_terminal_width**(*fallback=80*, *pad=2*)

Return current terminal width with an optional "safety buffer", which ensures that no unwanted line wrapping will happen.

Parameters

- **fallback** (*int*) – Default value when shutil is unavailable and environment variable COLUMNS is unset.
- **pad** (*int*) – Additional safety space to prevent unwanted line wrapping.

Return type
int

pytermor.term.**get_preferable_wrap_width**(*force_width=None*)

Return preferable terminal width for comfort reading of wrapped text (max=120).

Parameters
force_width (*Optional[int]*) – Ignore current terminal width and use this value as a result.

Return type
int

pytermor.term.**wait_key**(*block=True*)

Wait for a key press on the console and return it.

Parameters
block (*bool*) – Determines setup of O_NONBLOCK flag.

Return type
Optional[AnyStr]

pytermor.term.**confirm**(*attempts=1*, *default=False*, *keymap=None*, *prompt=None*, *quiet=False*, *required=False*)

Ensure the next action is manually confirmed by user. Print the terminal prompt with `prompt` text and wait for a keypress. Return *True* if user pressed `Y` and *False* in all the other cases (by default).

Valid keys are `Y` and `N` (case insensitive), while all the other keys and combinations are considered invalid, and will trigger the return of the `default` value, which is *False* if not set otherwise. In other words, by default the user is expected to press either `Y` or `N`, and if that's not the case, the confirmation request will be automatically failed.

`Ctrl+C` instantly aborts the confirmation process regardless of attempts count and raises UserAbort.

Example keymap (default one):

```
keymap = {"y": True, "n": False}
```

Parameters

- **attempts** (*int*) – Set how many times the user is allowed to perform the input before auto-cancellation (or auto-confirmation) will occur. 1 means there will be only one attempt, the first one. When set to -1, allows to repeat the input infinitely.
- **default** (*bool*) – Default value that will be returned when user presses invalid key (e.g. `Backspace`, `Ctrl+Q` etc.) and his `attempts` counter decreases to 0. Setting this to *True* effectively means that the user's only way to deny the request is to press `N` or `Ctrl+C`, while all the other keys are treated as `Y`.
- **keymap** (*Optional[Mapping[str, bool]]*) – Key to result mapping.
- **prompt** (*Optional[str]*) – String to display before each input attempt. Default is: `"Press Y to continue, N to cancel, Ctrl+C to abort:  "`
- **quiet** (*bool*) – If set to *True*, suppress all messages to stdout and work silently.
- **required** (*bool*) – If set to *True*, raise UserCancel or UserAbort when user rejects to confirm current action. If set to *False*, do not raise any exceptions, just return *False*.

Raises

- ***UserAbort*** – On corresponding event, if required is *True*.
- ***UserCancel*** – On corresponding event, if required is *True*.

Returns

True if there was a confirmation by user's input or automatically, *False* otherwise.

Return type

bool

pytermor.term.**get_char_width**(*char*, *block*)

General-purpose method for getting width of a character in terminal columns.

Uses guess_char_width() method based on unicodedata package, or/and QCP-RCP ANSI control sequence communication protocol.

Parameters

- **char** (*str*) – Input char.
- **block** (*bool*) – Set to *True* if you prefer slow, but 100% accurate measuring <measure_char_width> (which **blocks** and requires an output tty), or *False* for a device-independent, deterministic and non-blocking guessing <guess_char_width>, which works most of the time, although there could be rare cases when it is not precise enough.

Return type

int

pytermor.term.**measure_char_width**(*char*, *clear_after=True*)

Low-level function that returns the exact character width in terminal columns.

The main idea is to reset a cursor position to 1st column, print the required character and QCP <make_query_cursor_position()> control sequence; after that wait for the response and parse <decompose_request_cursor_position()> it. Normally it contains the cursor coordinates, which can tell the exact width of a character in question.

After reading the response clear it from the screen and reset the cursor to column 1 again.

Important: The `stdout` must be a tty. If it is not, consider using guess_char_width() instead, or `IOError` will be raised.

Warning: Invoking this method produces a bit of garbage in the output stream, which looks like this: `[3;2R`. By default, it is hidden using screen line clearing (see `clear_after`).

Warning: Invoking this method may **block** infinitely. Consider using a thread or set a timeout for the main thread using a signal if that is unwanted.

Parameters

- **char** (*str*) – Input char.
- **clear_after** (*bool*) – Send EL <make_erase_in_line()> control sequence after the terminal response to hide excessive utility information from the output if set to *True*, or leave it be otherwise.

Raises

IOError – If `stdout` is not a terminal emulator.

Return type

int

pytermor.term.**guess_char_width**(*c*)

Determine how many columns are needed to display a character in a terminal.

Returns -1 if the character is not printable. Returns 0, 1 or 2 for other characters.

Utilizes unicodedata table. A terminal emulator is unnecessary.

Parameters

c (*str*) –

Return type

int

7.14 pytermor.text

"Front-end" module of the library. Contains *renderables* – classes supporting high-level operations such as nesting-aware style application, concatenating and cropping of styled strings before the rendering, text alignment and wrapping, etc. Also provides rendering entrypoints render() and echo().

Functions

apply_style_selective(regex, string[, st])	Main purpose: application of under(over|cross)lined styles to strings containing more than one word.
apply_style_words_selective(string, st)	
distribute_padded()	**param max_len**
echo([string, fmt, renderer, nl, file, ...])	.
echoi([string, fmt, renderer, file, flush])	echo inline
is_rt(arg)	
render([string, fmt, renderer])	.
wrap_sgr(rendered, width[, indent_first, ...])	A workaround to make standard library `textwrap.wrap()` more friendly to an SGR-formatted strings.

Classes

Composite(*parts)	Simple class-container supporting concatenation of any IRenderable instances with each other without extra logic on top of it.
Fragment([string, fmt, close_this, close_prev])	<Immutable>
FrozenText(*fargs[, width, align, fill, ...])	Multi-fragment text with style nesting support.
IRenderable()	I
SimpleTable(*rows[, width, sep, border_st])	Table class with dynamic (not bound to each other) rows.
Text(*fargs[, width, align, fill, overflow, ...])	

class pytermor.text.**IRenderable**

Bases: Sized, ABC

I

abstract as_fragments()

a-s

Return type
List[Fragment]

abstract raw()

pass

Return type
str

abstract render(*renderer=None*)

pass

Return type
str

abstract set_width(*width*)

raise NotImplementedError

abstract property has_width: bool

return self._width is not None

abstract property allows_width_setup: bool

return False

class pytermor.text.**Fragment**(*string=''*, *fmt=None*, ***, *close_this=True*, *close_prev=False*)

Bases: *IRenderable*

<Immutable>

Can be formatted with f-strings. The text `:s` mode is required. Supported features:

- width [of the result];
- max length [of the content];
- alignment;
- filling.

```
>>> f"{Fragment('1234567890'):*^8.4s}"
'**1234**'
```

Parameters

- **string** (*str*) –
- **fmt** (*FT*) –
- **close_this** (*bool*) –
- **close_prev** (*bool*) –

as_fragments()

a-s

Return type
List[Fragment]

raw()

pass

Return type
str

property has_width: bool

return self._width is not None

property allows_width_setup: bool

return False

render(*renderer=None*)

pass

Return type
str

set_width(*width*)

raise NotImplementedError

class pytermor.text.**FrozenText**(**fargs*, *width=None*, *align=None*, *fill=' '*, *overflow=''*, *pad=0*, *pad_styled=True*)

Bases: *IRenderable*

Multi-fragment text with style nesting support.

Parameters
align (*str | Align*) – default is left

as_fragments()

a-s

Return type
List[Fragment]

raw()

pass

Return type
str

render(*renderer=None*)

Core rendering method

Parameters
renderer –

Returns

Return type

property allows_width_setup: bool

return False

property has_width: bool

return self._width is not None

set_width(*width*)

raise NotImplementedError

class pytermor.text.**Text**(**fargs*, *width=None*, *align=None*, *fill=' '*, *overflow=''*, *pad=0*, *pad_styled=True*)

Bases: *FrozenText*

set_width(*width*)

raise NotImplementedError

property allows_width_setup: bool

return False

as_fragments()

a-s

Return type
List[Fragment]

property has_width: bool

return self._width is not None

`raw`()

pass

Return type
str

`render`(*renderer=None*)

Core rendering method

Parameters
renderer –

Returns

Return type

`class pytermor.text.Composite`(**parts*)

Bases: *IRenderable*

Simple class-container supporting concatenation of any IRenderable instances with each other without extra logic on top of it. Renders parts joined by an empty string.

Parameters
parts (*RT*) – text parts in any format implementing IRenderable interface.

`as_fragments`()

a-s

Return type
List[Fragment]

`raw`()

pass

Return type
str

`render`(*renderer=None*)

pass

Return type
str

`set_width`(*width*)

raise NotImplementedError

`property has_width: bool`

return self._width is not None

`property allows_width_setup: bool`

return False

`class pytermor.text.SimpleTable`(**rows*, *width=None*, *sep=' '*, *border_st=<*_NoOpStyle[]>*)

Bases: *IRenderable*

Table class with dynamic (not bound to each other) rows. By defualt expands to the maximum width (terminal size).

Allows 0 or 1 dynamic-width cell in each row, while all the others should be static, i.e., be instances of FrozenText.

```
>>> echo(
...     SimpleTable(
...     [
...         Text("1", width=1),
...         Text("word", width=6, align='center'),
...         Text("smol string"),
...     ],
...     [
...         Text("2", width=1),
...         Text("padded word", width=6, align='center', pad=2),
...         Text("biiiiiiiiiiiiiiiiiiiiiiiiiiiiig string"),
...     ],
...     width=30,
...     sep="|"
... ), file=sys.stdout)
|1| word |smol string        |
|2| padd |biiiiiiiiiiiiiiiiii|
```

Create

Parameters

- **rows** (*t.Iterable[RT]*) –
- **width** (*int*) – Table width, in characters. When omitted, equals to terminal size if applicable, and to fallback value (80) otherwise.
- **sep** (*str*) –
- **border_st** (Style) –

as_fragments()

a-s

Return type
List[Fragment]

raw()

pass

Return type
str

property allows_width_setup: bool

return False

property has_width: bool

return self._width is not None

render(*renderer=None*)

pass

Return type
str

set_width(*width*)

raise NotImplementedError

pytermor.text.**render**(*string=''*, *fmt=<*_NoOpStyle[]>*, *renderer=None*)

.

Parameters

- **string** (*Union[RT, Iterable[RT]]*) – 2
- **fmt** (*FT*) – 2
- **renderer** (*Optional[Union[IRenderer, Type[IRenderer]]]*) – 2

Returns

Return type
Union[str, *List*[str]]

pytermor.text.**echo**(*string=''*, *fmt=<*_NoOpStyle[]>*, *renderer=None*, ***, *nl=True*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *flush=True*, *wrap=False*, *indent_first=0*, *indent_subseq=0*)

.

Parameters

- **string** (*Union[RT, Iterable[RT]]*) –
- **fmt** (*FT*) –
- **renderer** (*Optional[IRenderer]*) –
- **nl** (*bool*) –
- **file** (*IO*) –
- **flush** (*bool*) –
- **wrap** (*bool | int*) –
- **indent_first** (*int*) –
- **indent_subseq** (*int*) –

pytermor.text.**echoi**(*string=''*, *fmt=<*_NoOpStyle[]>*, *renderer=None*, ***, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *flush=True*)

echo inline

Parameters

- **string** (*Union[RT, Iterable[RT]]*) –
- **fmt** (*FT*) –
- **renderer** (*Optional[IRenderer]*) –
- **file** (*IO*) –
- **flush** (*bool*) –

Returns

Return type
None

pytermor.text.**distribute_padded**(*max_len: int*, **values: str*, *pad_left: int = 0*, *pad_right: int = 0*) → str
pytermor.text.**distribute_padded**(*max_len: int*, **values: RT*, *pad_left: int = 0*, *pad_right: int = 0*) → *Text*

Parameters

- **max_len** –
- **values** –
- **pad_left** –
- **pad_right** –

Returns

pytermor.text.**wrap_sgr**(*rendered*, *width*, *indent_first=0*, *indent_subseq=0*)

A workaround to make standard library `textwrap.wrap()` more friendly to an SGR-formatted strings.

The main idea is

Parameters

- **rendered** (*str | list[str]*) –
- **width** (*int*) –
- **indent_first** (*int*) –
- **indent_subseq** (*int*) –

Return type
str

pytermor.text.**apply_style_selective**(*regex*, *string*, *st=<*_NoOpStyle[]>*)

Main purpose: application of under(over|cross)lined styles to strings containing more than one word. Although the method can be used with any style and splitting rule provided. The result is a sequence of Fragments <Fragment> with styling applied only to specified parts of the original string.

Regex should consist of two groups, first for parts to apply style to, second for parts to return without any style (see NOOP_STYLE). This regex is used internally for python's re.findall() method.

The example below demonstrates how to color all the capital letters in the string in red color:

```
>>> render([
...     *apply_style_selective(
...         re.compile(R'([A-Z]+)([^A-Z]+|$)'),
...         "A few CAPITALs",
...         Style(fg='red'),
...     )
... ], renderer=SgrRenderer(OutputMode.XTERM_16))
['[31mA[39m', ' few ', '[31mCAPITAL[39m', 's']
```

A few CAPITALs

Parameters

- **regex** (*Pattern*) –
- **string** (*str*) –
- **st** (Style) –

Return type
Sequence[Fragment]

8

CONFIGURATION

The library initializes it's own config class just after being imported (*init_config()*). There are two ways to customize the setup:

1) create new *Config* instance from scratch and activate with *replace_config()*;
2) or preliminarily set the corresponding environment variables to intended values, and the default config instance will catch them up on initialization. Environment variable names are rendered in the documentation like this: PYTERMOR_VARIABLE_NAME.

8.1 Variables

Config.renderer_class
Explicitly set default renderer class (e.g. `TmuxRenderer`). Default renderer class is used for rendering if there is no explicitly specified one. Corresponding environment variable is PYTERMOR_RENDERER_CLASS. See also: guide.renderer_priority.

Config.force_output_mode
is a standard for in-band signaling to control cursor location, color, font styling, and other options on video text terminals and terminal emulators. Certain sequences of bytes, most starting with an ASCII escape character and a bracket character, are embedded into text. The terminal interprets these sequences as commands, rather than text to display verbatim. Corresponding environment variable is PYTERMOR_FORCE_OUTPUT_MODE.

Config.default_output_mode
is a standard for in-band signaling to control cursor location, color, font styling, and other options on video text terminals and terminal emulators. Certain sequences of bytes, most starting with an ASCII escape character and a bracket character, are embedded into text. The terminal interprets these sequences as commands, rather than text to display verbatim. Corresponding environment variable is PYTERMOR_DEFAULT_OUTPUT_MODE.

Config.prefer_rgb
is a standard for in-band signaling to control cursor location, color, font styling, and other options on video text terminals and terminal emulators. Certain sequences of bytes, most starting with an ASCII escape character and a bracket character, are embedded into text. The terminal interprets these sequences as commands, rather than text to display verbatim. Corresponding environment variable is PYTERMOR_PREFER_RGB.

Config.trace_renders

is a standard for in-band signaling to control cursor location, color, font styling, and other options on video text terminals and terminal emulators. Certain sequences of bytes, most starting with an ASCII escape character and a bracket character, are embedded into text. The terminal interprets these sequences as commands, rather than text to display verbatim. yare-yare-daze Corresponding environment variable is PYTERMOR_TRACE_RENDERS.

9

CLI USAGE

Commands like the ones below can be used for quick experimenting without loading the IDE:

- One-liner for system-wide installation (which is not recommended):

```
$ python -c "import pytermor as pt; pt.echo('RED', 'red')"
```

Todo: Find a solution for embedding colored text into PDF (as SVG -> PNG maybe?)

- One-liner for virtual environment (venv) with pytermor pre-installed (see `install`_) (note that the library source code root folder should be used as current working directory):

```
$ PYTHONPATH=. venv/bin/python -c "import pytermor as pt; pt.echo('GREEN', 'green
↪')"
```

- Interactive mode for virtual environment with pytermor pre-installed (again, current working directory should be sources root dir):

```
$ PYTHONSTARTUP=.run-startup.py PYTHONPATH=. venv/bin/python -qi
```

```
python 3.8.10
pytermor 2.41.1-dev1
>>> pt.echo("This is warning, be warned", pt.Styles.WARNING)
```

10

CHANGELOG

10.1 Releases

This project uses Semantic Versioning – https://semver.org *(starting from v2.0)*

10.1.1 pending

- …
- changelog update
- [FIX] noop color .id read
- [FIX] legacy virtual SequenceSGR descendants
- [NEW] DynamicColor
- [REFACTOR] split color hierarchy into ResolvableColor, RenderColor and RealColor

10.1.2 2.99-dev

Aug 23

- [CI/CD] artifact uploading
- [DOCS] Renderers and ANSI sequences review pages
- [DOCS] library structure diagram optimized for dark mode
- [FIX] logging
- [FIX] format_auto_float edge case
- [FIX] DualFormatter tuning
- [FIX] imports
- [FIX] template splitter mode

- [FIX] compose_clear_line_fill_bg now correctlyl handle requests to fill line from the middle
- [FIX] SequenceNf assembling
- docker image based on python 3.10 <- 3.8
- test dependencies
- missing imports
- [NEW] common helpers: only, but, ours, others, isiterable, flatten, char_range
- [NEW] auto-normalization of RGB values
- [NEW] substitute, is_rt, cut, fit
- [NEW] AbstractNamedGroupsRefilter, AbstractRegexValRefilter, AbstractStringTracer, AbstractTracer, IRefilter, OmniPadder
- [NEW] highlighter._multiapply
- [NEW] Text constructor fragment in args autodetect
- [NEW] TestSgrVisualizer
- [NEW] automated customizing of library structure diagram generation
- [NEW] added tuple support into fargs-parsing
- [NEW] http colors
- [NEW] template option STYLE_WORDS_SELECTIVE_COMMA
- [NEW] +16 named colors
- [NEW] +1 more named color
- [NEW] addr_offset param for Tracers
- [NEW] fit support for fillchar customizing
- [NEW] Tracers handling empty input
- [NEW] +1 named color
- [NEW] TempateEngine global_style argument TempalteEngine.render() method
- [NEW] color difference formula updated to CIE76 E*
- [NEW] xkcd named colors
- [NEW] fargs now support arbitrary order of arguments independent of their types
- [REFACTOR] transferred make_* methods from ansi to term. and parser to ansi
- [REFACTOR] moved trace() from filter to log
- [REFACTOR] render tracing log level
- [REFACTOR] simplified ArgTypeError
- [REFACTOR] optimized imports
- [REFACTOR] TemplateEngine
- [REFACTOR] measure fit
- [REFACTOR] merged conv and color modules into sole color, also merged two class hierarchies into one
- [REFACTOR] color transformation methods
- [REMOVE] AbstractRegexValRefilter, StringAligner
- [REMOVE] TemplateRenderer
- [TESTS] common module

- [TESTS] covered filter module
- [TESTS]
- [TESTS] template
- [TESTS] 99% coverage
- [TESTS] 100% coverage
- [TESTS] fix params
- [TESTS] 100% coverage again

10.1.3 v2.75-dev

Jun 23

- [DOCS] fixed pydoc escaped spaces to stop python's warnings whining that breaks the CI
- [FIX] ESCAPE_SEQ_REGEX
- [FIX] ESC_SEQ_REGEX
- [FIX] filter.AbstractTracer faulty offset rendering
- [FIX] flake8
- [FIX] make_clear_display_and_history() -> make_clear_history()
- [FIX] numfmt exports
- [FIX] pydeps invocation
- [FIX] template options parsing issue
- add __updated__ field to init file
- add updated field in _version.py
- CI coverage now running on python 3.10 (was 3.8)
- cleanup
- disabled verbose mode on CI
- pdf documentation
- replaced GITHUB_TOKEN secret to COVERALLS_REPO_TOKEN
- upload to coveralls debug mode !@#$
- [NEW] IRenderable.raw() method
- [NEW] Text.split_by_spaces(), Composite
- [NEW] "frozen" `Style` attribute
- [NEW] 'skylight-blue' named color
- [NEW] +3 base sequence classes, +26 preset sequences
- [NEW] __str__ methods override for named tuples RGB, HSV
- [NEW] contains_sgr method
- [NEW] cval atlassian colors
- [NEW] parser module
- [NEW] force_ansi_rendering, force_no_ansi_rendering
- [NEW] LAB, XYZ named tuples + conversions
- [NEW] StringReplacerChain filter

- [NEW] *Style*, SgrRenderer and TmuxRenderer support of all the above
- [NEW] TemplateEngine comment support
- [NEW] Tracers auto-width mode
- [NEW] utilmisc color transform methods overloaded
- [NEW] add ColorTarget enum as there are three extended color modes instead of two
- [NEW] add SubtypedParam support that allows specifying SGRs with subparams like 'ESC[4:3m'
- [NEW] implement missing 1st-level sequence classes
- [NEW] IntCodes: `FRAMED` (+``_OFF``), `UNDERLINE_COLOR_EXTENDED` (+``_OFF``)
- [NEW] math rendering as png
- [NEW] SeqIndex: `CURLY_UNDERLINED`, `FRAMED`, `FRAMED_OFF`
- [REFACTOR] split commons into log and excepiton modules
- [REFACTOR] TemplateEngine
- [REFACTOR] color resolver
- [REFACTOR] made measure and trace private
- [REFACTOR] sequence internal composition
- [REFACTOR] split `PYTERMOR_OUTPUT_MODE` env var into `PYTERMOR_FORCE_OUTPUT_MODE` and `PYTERMOR_AUTO_OUTPUT_MODE`
- [REWORK] util* -> numfmt, filter, conv
- [REWORK] doc pages tree
- [TESTS] 83% coverage
- [TESTS] *Style*/IColor reprs
- [TESTS] coverage 87%
- [TESTS] moar
- [UPDATE] Update coverage.yml

10.1.4 v2.48-dev

Apr 23

- [DOCS] small fixes
- [DOCS] updated changelog
- [FIX] measure_char_width and get_char_width internal logic
- [FIX] pipelines
- [FIX] AbstractTracer failure on empty input
- [FIX] StaticFormatter padding
- [FIX] bug in SimpleTable renderer when row is wider than a terminal
- [FIX] debug logging
- coverage git ignore
- cli-docker make command
- Dockerfile for repeatable builds
- hatch as build backend

- copyrights update
- host system/docker interchangable building automations
- [NEW] format_time, format_time_ms, format_time_ns
- [NEW] Hightlighter from static methods to real class
- [NEW] lab_to_rgb()
- [NEW] numeric formatters fallback mechanics
- [REFACTOR] `TDF_REGISTRY` -> dual_registry- ``FORMATTER_` constants from top-level imports
- [REFACTOR] `utilnum._TDF_REGISTRY` -> TDF_REGISTRY
- [REFACTOR] edited highlighter styles
- [REFACTOR] naming:
  - `CustomBaseUnit` -> DualBaseUnit
  - `DynamicBaseFormatter` -> DynamicFormatter
  - `StaticBaseFormatter` -> StaticFormatter
- [TESTS] numeric formatters colorizing
- [UPDATE] README
- [UPDATE] license is now Lesser GPL v3

10.1.5 v2.40-dev

Feb 23

- [DOCS] changelog update
- [DOCS] utilnum module
- [DOCS] rethinking of references style
- [FIX] parse method of TemplateEngine
- [FIX] Highlighter
- [FIX] critical Styles color
- 2023 copytight update
- [NEW] coveralls.io integration
- [NEW] echoi, flatten, flatten1 methods; SimpleTable class
- [NEW] StringLinearizer, WhitespaceRemover
- [NEW] text Fragments validation
- [NEW] Config class
- [NEW] hex rst text role
- [NEW] utilnum.format_bytes_human()
- [NEW] add es7s C45/Kalm to rgb colors list
- [NEW] methods percentile and median ; render_benchmark example
- [REFACTOR] IRenderable rewrite
- [REFACTOR] distribute_padded overloads
- [REFACTOR] attempt to break cyclic dependency of util.* modules
- [REFACTOR] moved color transformations and type vars from _commons

- [TESTS] additional coverage for utilnum

10.1.6 v2.32-dev

Jan 23

- [DOCS] utilnum update
- [DOCS] docstrings, typing
- [DOCS] utilnum module
- [FIX] format_prefixed and format_auto_float inaccuracies
- [FIX] Text.prepend typing
- [FIX] TmuxRenderer RGB output
- [NEW] Color256 aliases "colorNN"
- [NEW] Highlighter from es7s, colorizing options of utilnum helpers
- [NEW] IRenderable result caching
- [NEW] pad, padv helpers
- [NEW] prefix_refpoint_shift argument of PrefixedUnitFormatter
- [NEW] PrefixedUnitFormatter inheritance
- [NEW] String and FixedString base renderables
- [NEW] style.merge_styles()
- [NEW] Renderable __eq__ methods
- [NEW] StyledString
- [NEW] utilmisc get_char_width(), guess_char_width(), measure_char_width()
- [NEW] style merging strategies: merge_fallback(), merge_overwrite
- [NEW] subsecond delta support for TimeDeltaFormatter
- [TESTS] utilnum update
- [TESTS] integrated in-code doctests into pytest

10.1.7 v2.23-dev

- [FIX] OmniHexPrinter missed out newlines
- [NEW] dump printer caching
- [NEW] Printers and Mappers
- [NEW] SgrRenderer now supports non-default IO stream specifying
- [NEW] utilstr.StringHexPrinter and utilstr.StringUcpPrinter
- [NEW] add missing hsv_to_rgb function
- [NEW] extracted resolve, approximate, find_closest from Color class to module level, as well as color transform functions
- [NEW] split Text to Text and FrozenText

10.1.8 v2.18-dev

- [FIX] Disabled automatic rendering of echo() and render().
- [NEW] ArgCountError migrated from es7s/core.
- [NEW] black code style.
- [NEW] cval autobuild.
- [NEW] Add OmniHexPrinter and chunk() helper.
- [NEW] Typehinting.

10.1.9 v2.14-dev

Dec 22

- [DOCS] Docs design fixes.
- [NEW] confirm() helper command.
- [NEW] EscapeSequenceStringReplacer filter.
- [NEW] examples/terminal_benchmark script.
- [NEW] StringFilter and OmniFilter classes.
- [NEW] Minor core improvements.
- [NEW] RGB and variations full support.
- [TESTS] Tests for color module.

10.1.10 v2.6-dev

Nov 22

- [NEW] TemplateEngine implementation.
- [NEW] Text nesting.
- [REFACTOR] Changes in ConfigurableRenderer.force_styles logic.
- [REFACTOR] Got rid of Span class.
- [REFACTOR] Package reorganizing.
- [REFACTOR] Rewrite of color module.

10.1.11 v2.2-dev

Oct 22

- [NEW] TmuxRenderer
- [NEW] wait_key() input helper.
- [NEW] Color config.
- [NEW] IRenderable` interface.
- [NEW] Named colors list.

10.1.12 v2.1-dev

Aug 22

- [NEW] Color presets.
- [TESTS] More unit tests for formatters.

10.1.13 v2.0-dev

Jul 22

- [REWORK] Complete library rewrite.
- [DOCS] `sphinx` and `readthedocs` integraton.
- [NEW] High-level abstractions Color, Renderer <SgrRenderer> and *Style*.
- [TESTS] `pytest` and `coverage` integration.
- [TESTS] Unit tests for formatters and new modules.

10.1.14 v1.8

Jun 22

- [NEW] `format_prefixed_unit` extended for working with decimal and binary metric prefixes.
- [NEW] `sequence.NOOP` SGR sequence and `span.NOOP` format.
- [NEW] format_time_delta extended with new settings.
- [NEW] Added 3 formatters: `format_prefixed_unit`, format_time_delta, format_auto_float.
- [NEW] Max decimal points for auto_float extended from (2) to (max-2).
- [REFACTOR] Utility classes reorganization.
- [REFACTOR] Value rounding transferred from format_auto_float to `format_prefixed_unit`.
- [TESTS] Unit tests output formatting.

10.1.15 v1.7

May 22

- [FIX] Print reset sequence as `\e[m` instead of `\e[0m`.
- [NEW] Span constructor can be called without arguments.
- [NEW] Added `span.BG_BLACK` format.
- [NEW] Added ljust_sgr, rjust_sgr, center_sgr util functions to align strings with SGRs correctly.
- [NEW] Added SGR code lists.

10.1.16 v1.6

- [REFACTOR] Renamed `code` module to `sgr` because of conflicts in PyCharm debugger (`pydevd_console_integration.py`).
- [REFACTOR] Ridded of `EmptyFormat` and `AbstractFormat` classes.
- [TESTS] Excluded `tests` dir from distribution package.

10.1.17 v1.5

- [REFACTOR] Removed excessive `EmptySequenceSGR` – default `SGR` class was specifically implemented to print out as empty string instead of `\e[m` if constructed without params.

10.1.18 v1.4

- [NEW] Span.wrap() now accepts any type of argument, not only *str*.
- [NEW] Added equality methods for SequenceSGR and Span classes/subclasses.
- [REFACTOR] Rebuilt `Sequence` inheritance tree.
- [TESTS] Added some tests for `fmt.*` and `seq.*` classes.

10.1.19 v1.3

- [NEW] Added `span.GRAY` and `span.BG_GRAY` format presets.
- [REFACTOR] Interface revisioning.

10.1.20 v1.2

- [NEW] `EmptySequenceSGR` and `EmptyFormat` classes.
- [NEW] opening_seq and closing_seq properties for Span class.

10.1.21 v1.1

Apr 22

- [NEW] Autoformat feature.

10.1.22 v1.0

- First public version.

10.1.23 v0.90

Mar 22

- First commit.

11

LICENSE

```
                    Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.
```

```
the terms and conditions of version 3 of the GNU General Public
License, supplemented by the additional permissions listed below.

  0. Additional Definitions.

  As used herein, "this License" refers to version 3 of the GNU Lesser
General Public License, and the "GNU GPL" refers to version 3 of the GNU
General Public License.

  "The Library" refers to a covered work governed by this License,
other than an Application or a Combined Work as defined below.

  An "Application" is any work that makes use of an interface provided
by the Library, but which is not otherwise based on the Library.
Defining a subclass of a class defined by the Library is deemed a mode
of using an interface provided by the Library.

  A "Combined Work" is a work produced by combining or linking an
Application with the Library.  The particular version of the Library
with which the Combined Work was made is also called the "Linked
Version".

  The "Minimal Corresponding Source" for a Combined Work means the
Corresponding Source for the Combined Work, excluding any source code
for portions of the Combined Work that, considered in isolation, are
based on the Application, and not on the Linked Version.
```

(continues on next page)

(continued from previous page)

```
  The "Corresponding Application Code" for a Combined Work means the
object code and/or source code for the Application, including any data
and utility programs needed for reproducing the Combined Work from the
Application, but excluding the System Libraries of the Combined Work.

  1. Exception to Section 3 of the GNU GPL.

  You may convey a covered work under sections 3 and 4 of this License
without being bound by section 3 of the GNU GPL.

  2. Conveying Modified Versions.

  If you modify a copy of the Library, and, in your modifications, a
facility refers to a function or data to be supplied by an Application
that uses the facility (other than as an argument passed when the
facility is invoked), then you may convey a copy of the modified
version:

   a) under this License, provided that you make a good faith effort to
   ensure that, in the event an Application does not supply the
   function or data, the facility still operates, and performs
   whatever part of its purpose remains meaningful, or

   b) under the GNU GPL, with none of the additional permissions of
   this License applicable to that copy.

  3. Object Code Incorporating Material from Library Header Files.

  The object code form of an Application may incorporate material from
a header file that is part of the Library.  You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

   a) Give prominent notice with each copy of the object code that the
   Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the object code with a copy of the GNU GPL and this license
   document.

  4. Combined Works.

  You may convey a Combined Work under terms of your choice that,
taken together, effectively do not restrict modification of the
portions of the Library contained in the Combined Work and reverse
engineering for debugging such modifications, if you also do each of
the following:

   a) Give prominent notice with each copy of the Combined Work that
   the Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the Combined Work with a copy of the GNU GPL and this license
```

(continues on next page)

(continued from previous page)

```
   document.

   c) For a Combined Work that displays copyright notices during
   execution, include the copyright notice for the Library among
   these notices, as well as a reference directing the user to the
   copies of the GNU GPL and this license document.

   d) Do one of the following:

       0) Convey the Minimal Corresponding Source under the terms of this
       License, and the Corresponding Application Code in a form
       suitable for, and under terms that permit, the user to
       recombine or relink the Application with a modified version of
       the Linked Version to produce a modified Combined Work, in the
       manner specified by section 6 of the GNU GPL for conveying
       Corresponding Source.

       1) Use a suitable shared library mechanism for linking with the
       Library.  A suitable mechanism is one that (a) uses at run time
       a copy of the Library already present on the user's computer
       system, and (b) will operate properly with a modified version
       of the Library that is interface-compatible with the Linked
       Version.

   e) Provide Installation Information, but only if you would otherwise
   be required to provide such information under section 6 of the
   GNU GPL, and only to the extent that such information is
   necessary to install and execute a modified version of the
   Combined Work produced by recombining or relinking the
   Application with a modified version of the Linked Version. (If
   you use option 4d0, the Installation Information must accompany
   the Minimal Corresponding Source and Corresponding Application
   Code. If you use option 4d1, you must provide the Installation
   Information in the manner specified by section 6 of the GNU GPL
   for conveying Corresponding Source.)

  5. Combined Libraries.

  You may place library facilities that are a work based on the
Library side by side in a single library together with other library
facilities that are not Applications and are not covered by this
License, and convey such a combined library under terms of your
choice, if you do both of the following:

   a) Accompany the combined library with a copy of the same work based
   on the Library, uncombined with any other library facilities,
   conveyed under the terms of this License.

   b) Give prominent notice with the combined library that part of it
   is a work based on the Library, and explaining where to find the
   accompanying uncombined form of the same work.

  6. Revised Versions of the GNU Lesser General Public License.

  The Free Software Foundation may publish revised and/or new versions
of the GNU Lesser General Public License from time to time. Such new
```

(continues on next page)

(continued from previous page)

```
versions will be similar in spirit to the present version, but may
differ in detail to address new problems or concerns.

  Each version is given a distinguishing version number. If the
Library as you received it specifies that a certain numbered version
of the GNU Lesser General Public License "or any later version"
applies to it, you have the option of following the terms and
conditions either of that published version or of any later version
published by the Free Software Foundation. If the Library as you
received it does not specify a version number of the GNU Lesser
General Public License, you may choose any version of the GNU Lesser
General Public License ever published by the Free Software Foundation.

  If the Library as you received it specifies that a proxy can decide
whether future versions of the GNU Lesser General Public License shall
apply, that proxy's public statement of acceptance of any version is
permanent authorization for you to choose that version for the
Library.
```

11.1 Disclaimer of Warranty

```
LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER
PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER
EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE
QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE
DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
```

11.2 Limitation of Liability

```
COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS
PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL,
INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE
THE PROGRAM, INCLUDING BUT NOT LIMITED TO LOSSES OF DATA, FAILURES OF THE
PROGRAM TO OPERATE WITH OTHER PROGRAMS, OTHER LOSSES AS WELL AS ACQUISITIONS,
BREAKDOWNS, REPAIRS, UNSCREWINGS, BLACKOUTS, FAINTINGS, INJURIES, BURNS, SNOW
AVALANCHES, EARTHQUAKES, VOLCANIC, GEYSER AND LIMNIC ERUPTIONS, TYPHOONS,
METEORITE AND SATELLITE FALLS AND OTHER NATURAL DISASTERS, AS WELL AS
BEHAVIORAL DEVIATIONS OF PEOPLE, SHARKS, SNAKES AND OTHER ANIMALS, ROBBERIES,
ASSAULTS, RAPES, THEFTS AND BURGLARIES, DRUNKEN BRAWLS AND RIOTS, INCESTS,
ABORTIONS, SHOULDER DISLOCATIONS, MILITARY CONSCRIPTIONS, DISFELLOWSHIPPINGS,
CONFINEMENTS AND EXTRADITIONS, DIVORCEMENTS, DEMOTIONS AND PROMOTIONS, EVEN IF
SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
```

12

DOCS GUIDELINES

(mostly as a reminder for myself)

12.1 General

- Basic types and built-in values should be surrounded with asterisks:

  `*True*` → *True*

  `*None*` → *None*

  `*int*` → *int*

- Library classes, methods, etc. should be enclosed in single backticks in order to become a hyperlinks:

  `` `SgrRenderer.render()` `` → SgrRenderer.render()

  If class name is ambiguous (e.g., there is a glossary term with the same name), the solution is to specify the type explicitly:

  `:class:`.Style`` → *Style*

- Argument names and string literals that include escape sequences or their fragments should be wrapped in double backticks:

  ``` ``arg1`` ``` → `arg1`

  ``` ``ESC [31m ESC [m`` ``` → `ESC [31m ESC [m`

  On the top of that, ESC control char should be padded with spaces for better readability. This also triggers automatic application of custom style for even more visual difference.

- Any formula should be formatted using LaTeX syntax (`:math:` role or `.. math::` directive):

$$d_{min} = 350 * 10^{-3}$$

12.2 Hexadecimals

Hexadecimal numbers should be displayed using `:hex:` role (applies to all examples below except the last one). In general, when the characters are supposed to be typed manually, or when the result length is 6+ chars, it's better to use lower case; when the numbers are distinct or "U+" notation is used, the upper case is acceptable:

separate bytes
: 0x1B 0x23 0x88

Unicode codepoints
: U+21BC ; U+F0909

hex dump
: "0x 00 AF 00 BB 11 BD AA B5"

UTF-8
: e0a489 efbfbe efbfaf f0af8cb3

RGB colors (*int*/*str* forms)
: 0xeb0c0c ; #ff00ff

escaped strings

```
"\u21bc", "\U000f0909", re.compile(R"\x1b\[[0-9;]*m")
```

12.3 References

External pages	github and gitlab	`` `github`_ and `gitlab <//gitlab.com>`_ ``
		
		`.. _github: //github.com`

External pydoc	`re.Match`	`:class:`re.Match``
Internal page	guide-lo or high-level <guide-hi>	`` `guide-lo` or `high-level <guide-hi>` ``
Internal page setup [1]	`.. _guide.core-api-1:`	
Internal pydoc	wait_key(), *Style*	`` `wait_key()`, :class:`.Style` ``
Internal anchor	*References*	`` `References`_ ``
Term in glossary	*rendering*	`:term:`rendering``
Inlined definition	classifier for 1st time … … or *classifier* later	`:def:`classifier` for 1st time ...` `... or *classifier* later`
Abbreviation	EL	`:abbr:`EL (Erase in Line)``

12.4 Headers

12.4.1 Section header

Subsection header

Paragraph header

Rubric

```
##########################
Docs guidelines
##########################
.. part header

=================
```

(continues on next page)

(continued from previous page)

```
Headers
================
.. chapter header

---------------
Section header
---------------

Subsection header
-------------------

Paragraph header
""""""""""""""""""""

.. rubric:: Rubric

.. code-block:: rst

   ...
```

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z