pytermor

Release 2.48.0.dev13

Alexandr Shavykin

Apr 02, 2023

Contents

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

Provides *high-level* methods for working with text sections, colors, formats, alignment and wrapping, as well as *low-level* `ansi` module which allows operating with SGR (Select Graphic Rendition) *sequences* and also implements automatic "soft" format termination. Depending on the context and technical requirements either approach can be used. Also includes a set of additional number/string/date formatters for pretty output.

Key feature of this library is extendability and a variety of formatters (called *renderers*), which determine the output syntax:

- *SgrRenderer* (global default)

- *TmuxRenderer*

- *HtmlRenderer*

- *SgrDebugger* (mostly for development)

- etc.

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

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

We put these pieces together to create a SGR command. Thus, `ESC[1m` specifies bold (or bright) text, and `ESC[31m` specifies red foreground text. We can chain together parameters; for example, `ESC[32;47m` specifies green foreground text on a white background.

The following diagram shows a complete example for rendering the word "text" in red with a single underline.
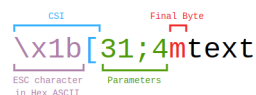
Notes

- For terminals that support bright foreground colors, `ESC[1;3Xm` is usually equivalent to `ESC[9Xm` (where `X` is a digit in 0-7). However, the reverse does not seem to hold, at least anecdotally: `ESC[2;9Xm` usually does not render the same as `ESC[3Xm`.
- Not all terminals support every effect.

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

Guide

1.1 Getting started

1.1.1 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
```

1.1.2 Structure

A L	Module	Class(es)	Purpose
Hi	*text*	*Text*	Container consisting of text pieces each with attached `Style`. Renders into specified format keeping all the formatting.
		Style *Styles*	Reusable abstractions defining colors and text attributes (text color, bg color, *bold* attribute, *underlined* attribute etc).
		SgrRenderer *HtmlRenderer* *TmuxRenderer* etc.	`SgrRenderer` transforms `Style` instances into `Color`, `Span` and `SequenceSGR` instances and assembles it all up. There are several other implementations depending on what output format is required.
	color	*Color16* *Color256* *ColorRGB*	Abstractions for color operations in different color modes (default 16-color, 256-color, RGB). Tools for color approximation and transformations.
		pytermor	Color registry.
Lo	*ansi*	Span	Abstraction consisting of "opening" SGR sequence defined by the developer (or taken from preset list) and complementary "closing" SGR sequence that is built automatically.
		Spans	Registry of predefined instances in case the developer doesn't need dynamic output formatting and just wants to colorize an error message.
		SequenceSGR *SeqIndex*	Abstractions for manipulating ANSI control sequences and classes-factories, plus a registry of preset SGRs.
		IntCodes	Registry of escape control sequence parameters.
	util	*	Additional formatters and common methods for manipulating strings with SGRs inside.

1.1.3 Features

One of the core concepts of the library is `Span` class. `Span` is a combination of two control sequences; it wraps specified string with pre-defined leading and trailing SGR definitions.

Example code:

```
1  from pytermor import Spans
2
3  print(Spans.RED('Feat') + Spans.BOLD('ures'))
```

Content-aware format nesting

Compose text spans with automatic content-aware span termination. Preset spans can safely overlap with each other (as long as they require different *breaker* sequences to reset).

```
1  from pytermor import Span
2
3  span1 = Span('blue', 'bold')
4  span2 = Span('cyan', 'inversed', 'underlined', 'italic')
5
6  msg = span1(f'Content{span2("-aware format")} nesting')
7  print(msg)
```
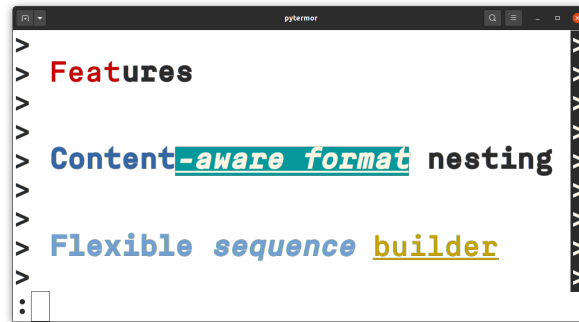
Flexible sequence builder

Create your own *SGR sequences* using default constructor, which accepts color/attribute keys, integer codes and even existing *SGRs*, in any amount and in any order. Key resolving is case-insensitive.

```python
from pytermor import SeqIndex, SequenceSGR

seq1 = SequenceSGR('hi_blue', 1)  # keys or integer codes
seq2 = SequenceSGR(seq1, SeqIndex.ITALIC)  # existing SGRs
seq3 = SequenceSGR('underlined', 'YELLOW')  # case-insensitive

msg = f'{seq1}Flexible{SeqIndex.RESET} ' + \
      f'{seq2}sequence{SeqIndex.RESET} ' + \
      str(seq3) + 'builder' + str(SeqIndex.RESET)
print(msg)
```

256 colors / True Color support

The library supports extended color modes:

- XTerm 256 colors indexed mode (see *ANSI preset list*);
- True Color RGB mode (16M colors).

```python
from pytermor import SequenceSGR, SeqIndex

start_color = 41
boxchr = "\u2588"
for idx, c in enumerate(range(start_color, start_color+(36*6), 36)):
    print(f'{SequenceSGR.new_color_256(c)}{boxchr*3}{SeqIndex.COLOR_OFF}', end='')

print('\n')
for idx, c in enumerate(range(0, 256, 256//17)):
    r = max(0, 255-c)
    g = max(0, min(255, 127-(c*2)))
    b = c
    print(f'{SequenceSGR.new_color_rgb(r, g, b)}{boxchr}{SeqIndex.COLOR_OFF}', end='')
```
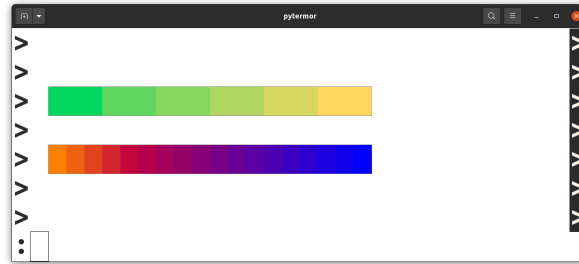
Customizable output formats

Todo: @TODOTODO

String and number formatters

Todo: @TODOTODO

1.1.4 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')"
```

- One-liner for virtual environment (venv) with *pytermor* pre-installed (see *Installation*) (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)
```

1.2 High-level API

1.2.1 Core methods

text.render([string, fmt, renderer, ...])	
.	
text.echo([string, fmt, renderer, ...])	
.	

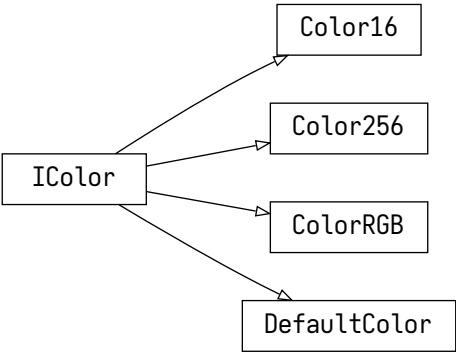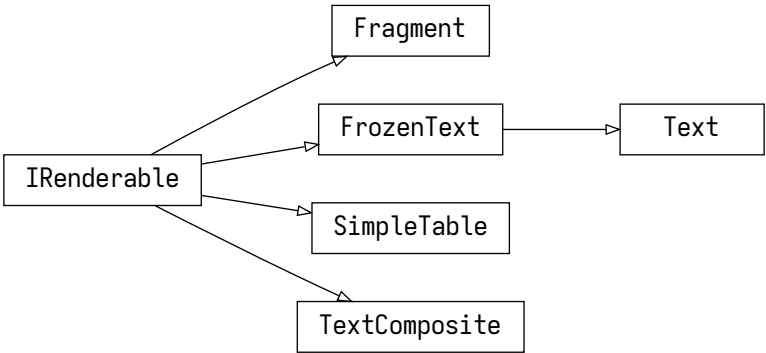|	
color.resolve_color(subject[, color_type])	Case-insensitive search through registry contents.
style.make_style([fmt])	General *Style* constructor.
style.merge_styles([base, fallbacks, overwrites])	Bulk style merging method.

Fig. 1: `IColor` inheritance tree

Fig. 2: *IRenderable* inheritance tree

1.2.2 Colors

1.2.3 Styles

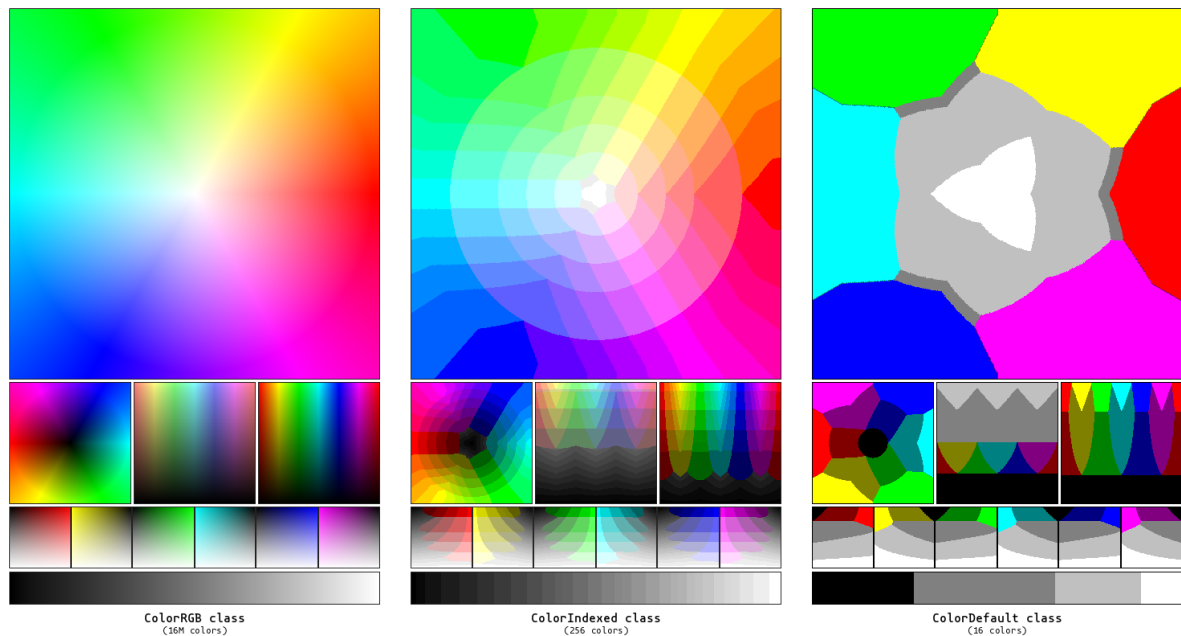## 1.2.4 Output format control

1.2.5 Color mode fallbacks

ColorRGB class
(16M colors)

ColorIndexed class
(256 colors)

ColorDefault class
(16 colors)

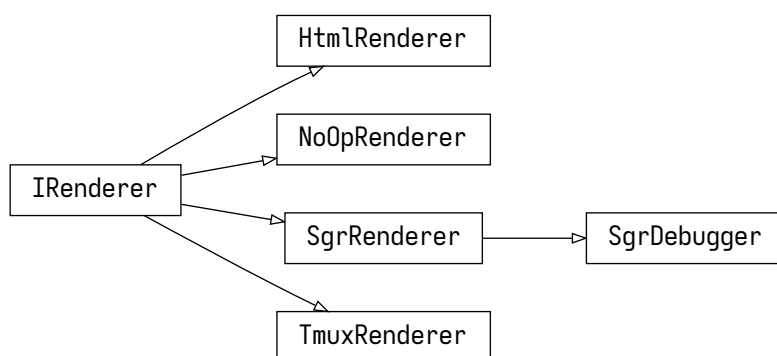Fig. 3: Color approximations for indexed modes

1.3 Renderers

Fig. 4: *IRenderer* inheritance tree
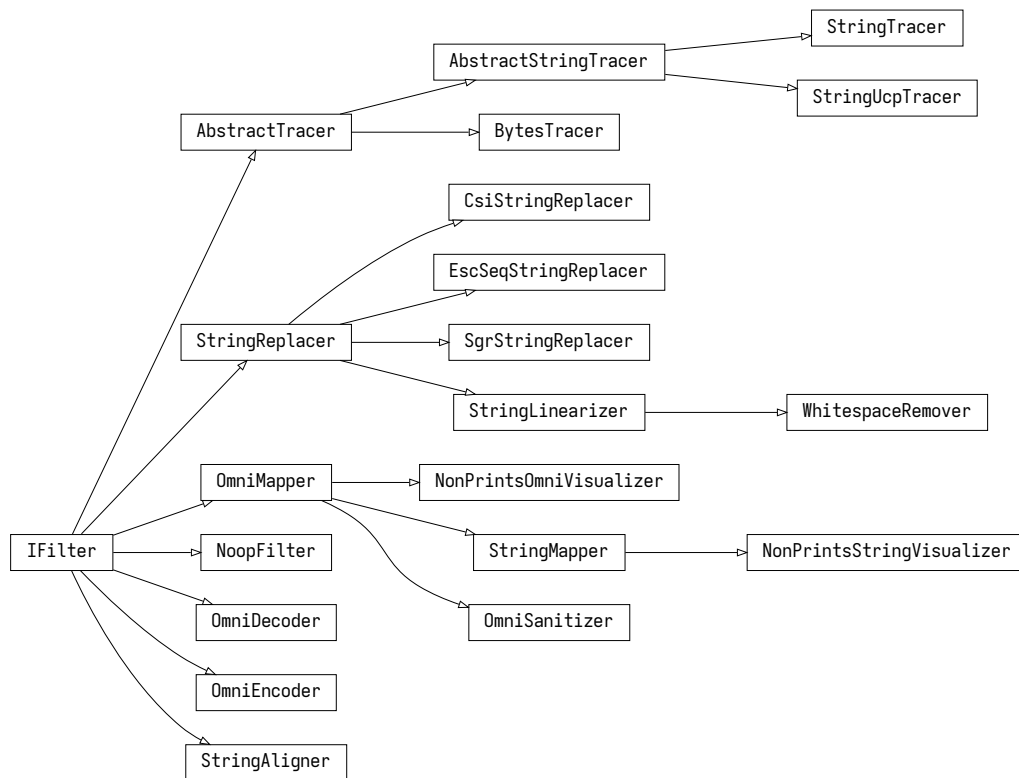
Todo: Win32Renderer ?

1.4 String filters

Fig. 5: *IFilter* inheritance tree

1.5 Number formatters
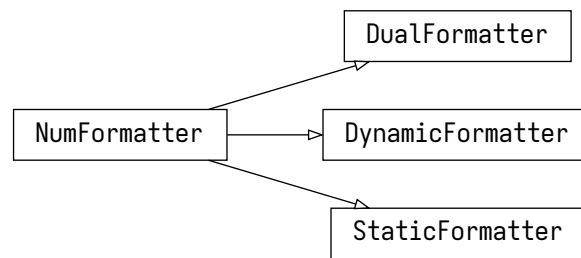
Todo: The library contains @TODO

Fig. 6: `NumFormatter` inheritance tree

1.5.1 Auto-float formatter

1.5.2 Prefixed-unit formatter

1.5.3 Time delta formatter

```
1   import pytermor.utilnum
2   from pytermor import RendererManager, SgrRenderer
3   from pytermor.util import time_delta
4
5   seconds_list = [2, 10, 60, 2700, 32340, 273600, 4752000, 864000000]
6   max_len_list = [3, 6, 10]
7
8   for max_len in max_len_list:
9       formatter = pytermor.utilnum.dual_registry.find_matching(max_len)
10
11  RendererManager.set_default(SgrRenderer)
12  for seconds in seconds_list:
13      for max_len in max_len_list:
14          formatter = pytermor.utilnum.dual_registry.get_by_max_len(max_len)
15          print(formatter.format(seconds), end=' ')
16      print()
```
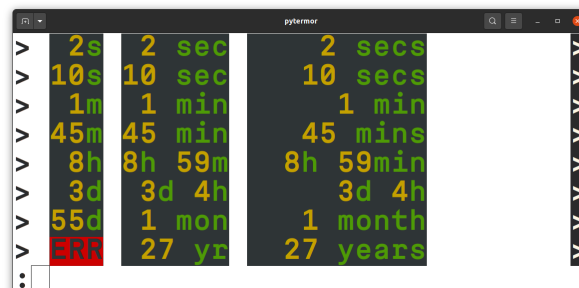
1.6 es7s color collection

Todo: @TODO

1.7 Low-level API

So, what's happening under the hood?

1.7.1 Core methods

|	
`ansi.SequenceSGR`(*args)	Class representing SGR-type escape sequence with varying amount of parameters.
`ansi.make_color_256`(code[, bg])	Wrapper for creation of *SequenceSGR* that sets foreground (or background) to one of 256-color palette value.:
`ansi.make_color_rgb`(r, g, b[, bg])	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 "0xRRGGBB". For example, sequence with color of 0xFF3300 can be created with::.
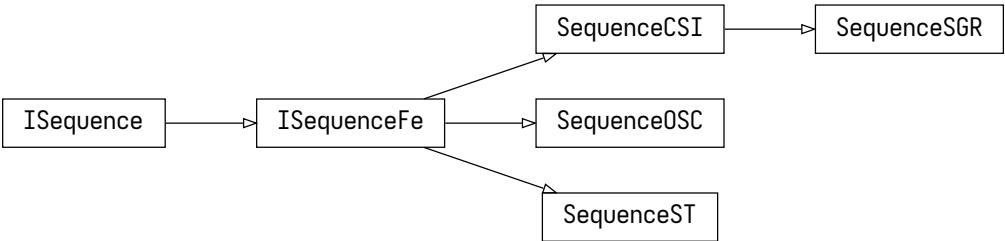
Fig. 7: *ISequence* inheritance tree

1.7.2 Format soft reset

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*, 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:

```
1  from pytermor import Span, Spans, SeqIndex
2
3  # implicitly:
4  span_warn = Span(93, 4)
5  # or explicitly:
6  span_warn = Span.init_explicit(
7      SeqIndex.HI_YELLOW + SeqIndex.UNDERLINED,  # sequences can be summed up, remember?
8      SeqIndex.COLOR_OFF + SeqIndex.UNDERLINED_OFF,  # "counteractive" sequences
9      hard_reset_after=False
10 )
11
12 orig_text = Spans.BOLD(f'this is {SeqIndex.BG_GRAY}the original{SeqIndex.RESET} string
   ↪')
13 updated_text = orig_text.replace('original', span_warn('updated'), 1)
14 print(orig_text, '\n', updated_text)
```
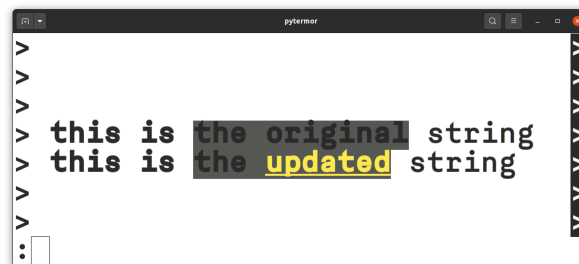

As you can see, the update went well – we kept all the previously applied formatting. Of course, this method cannot be 100% applicable; for example, imagine that original text was colored blue. After the update "string" word won't be blue anymore, as we used `SeqIndex.COLOR_OFF` escape sequence to neutralize 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).

1.7.3 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 preset list*);
- 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()`.

1.7.4 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 preset list* page.

Important: SequenceSGR with zero params was specifically implemented to translate into an empty string and not into ESC [m, which would make sense, but also could be very entangling, as terminal emulators interpret that sequence 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 preset list*);

- *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()* method or cast instance to *str*.

```python
from pytermor import SequenceSGR

seq = SequenceSGR(4, 7)
msg = f'({seq})'

print(msg + f'{SequenceSGR(0).assemble()}')
print(str(msg.assemble()))
print(msg.assemble().hex(':'))
```
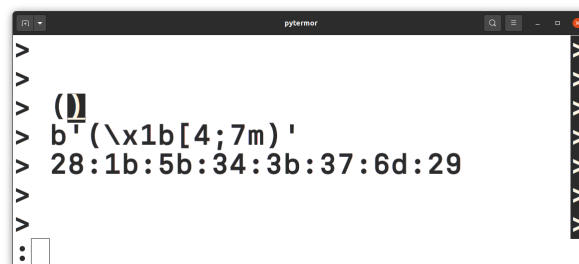

- First line is the string with encoded escape sequence;

- Second line shows up the string in raw mode, as if sequences were ignored by the terminal;

- Third line is hexadecimal string representation.

1.7.5 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 *introducer*; 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.

1.7.6 Combining SGRs

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

```
from pytermor import SequenceSGR, SeqIndex

combined = SequenceSGR(1, 31) + SequenceSGR(4)
print(f'{combined}combined{SeqIndex.RESET}', str(combined).assemble())
```

Sources

1. XTerm Control Sequences[5]
2. ECMA-48 specification[6]

1.8 ANSI preset list

Preset lists are omitted from API docs to avoid unnesessary duplication; summary list of all presets defined in the library (excluding `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)

---

[5] https://invisible-island.net/xterm/ctlseqs/ctlseqs.html
[6] https://www.ecma-international.org/publications-and-standards/standards/ecma-48/

1.8.1 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

1.8.2 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

1.8.3 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	

Table 2 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	XTERM_LIME	10		#00ff00	
	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	

Table 2 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	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	
	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	

Table 2 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	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	
	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	

Table 2 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	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**
	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	

Table 2 – continued from previous page

	Name	INT	STY	RGB code	XTerm name
	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	
	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	

---

[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.

1.8. ANSI preset list

Sources

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

2. https://www.ditig.com/256-colors-cheat-sheet

1.9 xterm color 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 . . .

000 #000000	001 #800000	002 #008000	003 #808000	004 #000080	005 #800080	006 #008080	007 #c0c0c0
008 #808080	009 #ff0000	010 #00ff00	011 #ffff00	012 #0000ff	013 #ff00ff	014 #00ffff	015 #ffffff

016 #000000	022 #005f00	028 #008700	034 #00af00	040 #00d700	046 #00ff00	082 #5fff00	076 #5fd700	070 #5faf00	064 #5f8700	058 #5f5f00	052 #5f0000
017 #00005f	023 #005f5f	029 #00875f	035 #00af5f	041 #00d75f	047 #00ff5f	083 #5fff5f	077 #5fd75f	071 #5faf5f	065 #5f875f	059 #5f5f5f	053 #5f005f
018 #000087	024 #005f87	030 #008787	036 #00af87	042 #00d787	048 #00ff87	084 #5fff87	078 #5fd787	072 #5faf87	066 #5f8787	060 #5f5f87	054 #5f0087
019 #0000af	025 #005faf	031 #0087af	037 #00afaf	043 #00d7af	049 #00ffaf	085 #5fffaf	079 #5fd7af	073 #5fafaf	067 #5f87af	061 #5f5faf	055 #5f00af
020 #0000d7	026 #005fd7	032 #0087d7	038 #00afd7	044 #00d7d7	050 #00ffd7	086 #5fffd7	080 #5fd7d7	074 #5fafd7	068 #5f87d7	062 #5f5fd7	056 #5f00d7
021 #0000ff	027 #005fff	033 #0087ff	039 #00afff	045 #00d7ff	051 #00ffff	087 #5fffff	081 #5fd7ff	075 #5fafff	069 #5f87ff	063 #5f5fff	057 #5f00ff
093 #8700ff	099 #875fff	105 #8787ff	111 #87afff	117 #87d7ff	123 #87ffff	159 #afffff	153 #afd7ff	147 #afafff	141 #af87ff	135 #af5fff	129 #af00ff
092 #8700d7	098 #875fd7	104 #8787d7	110 #87afd7	116 #87d7d7	122 #87ffd7	158 #afffd7	152 #afd7d7	146 #afafd7	140 #af87d7	134 #af5fd7	128 #af00d7
091 #8700af	097 #875faf	103 #8787af	109 #87afaf	115 #87d7af	121 #87ffaf	157 #afffaf	151 #afd7af	145 #afafaf	139 #af87af	133 #af5faf	127 #af00af
090 #870087	096 #875f87	102 #878787	108 #87af87	114 #87d787	120 #87ff87	156 #afff87	150 #afd787	144 #afaf87	138 #af8787	132 #af5f87	126 #af0087
089 #87005f	095 #875f5f	101 #87875f	107 #87af5f	113 #87d75f	119 #87ff5f	155 #afff5f	149 #afd75f	143 #afaf5f	137 #af875f	131 #af5f5f	125 #af005f
088 #870000	094 #875f00	100 #878700	106 #87af00	112 #87d700	118 #87ff00	154 #afff00	148 #afd700	142 #afaf00	136 #af8700	130 #af5f00	124 #af0000
160 #d70000	166 #d75f00	172 #d78700	178 #dfaf00	184 #dfdf00	190 #dfff00	226 #ffff00	220 #ffdf00	214 #ffaf00	208 #ff8700	202 #ff5f00	196 #ff0000
161 #d7005f	167 #d75f5f	173 #d7875f	179 #dfaf5f	185 #dfdf5f	191 #dfff5f	227 #ffff5f	221 #ffdf5f	215 #ffaf5f	209 #ff875f	203 #ff5f5f	197 #ff005f
162 #d70087	168 #d75f87	174 #d78787	180 #dfaf87	186 #dfdf87	192 #dfff87	228 #ffff87	222 #ffdf87	216 #ffaf87	210 #ff8787	204 #ff5f87	198 #ff0087
163 #d700af	169 #d75faf	175 #d787af	181 #dfafaf	187 #dfdfaf	193 #dfffaf	229 #ffffaf	223 #ffdfaf	217 #ffafaf	211 #ff87af	205 #ff5faf	199 #ff00af
164 #d700d7	170 #d75fd7	176 #d787d7	182 #dfafdf	188 #dfdfdf	194 #dfffdf	230 #ffffdf	224 #ffdfdf	218 #ffafdf	212 #ff87df	206 #ff5fdf	200 #ff00df
165 #d700ff	171 #d75fff	177 #d787ff	183 #dfafff	189 #dfdfff	195 #dfffff	231 #ffffff	225 #ffdfff	219 #ffafff	213 #ff87ff	207 #ff5fff	201 #ff00ff
232 #080808	233 #121212	234 #1c1c1c	235 #262626	236 #303030	237 #3a3a3a	238 #444444	239 #4e4e4e	240 #585858	241 #626262	242 #6c6c6c	243 #767676
244 #808080	245 #8a8a8a	246 #949494	247 #9e9e9e	248 #a8a8a8	249 #b2b2b2	250 #bcbcbc	251 #c6c6c6	252 #d0d0d0	253 #dadada	254 #e4e4e4	255 #eeeeee

Fig. 8: *Indexed* mode palette

Sources

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

1.10 Configuration

PYTERMOR_RENDERER_CLASS
    YES

PYTERMOR_OUTPUT_MODE
    YES

PYTERMOR_TRACE_RENDERS
    yare-yare-daze

PYTERMOR_PREFER_RGB
    YES

See also:

`Config` – class containing configuration variables.

1.11 Docs guidelines

(mostly as a reminder for myself)

1.11.1 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()*

- 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):

$$d_{min} = 350 * 10^{-3}$$

1.11.2 References

Type	Code	Example
Internal pydoc	use \`SgrRenderer.render()\`	use *SgrRenderer.render()*
Internal page	called \`renderers<guide. ↪renderers>\`	called *renderers*
Internal anchor	\`References\`_	*References*
External pydoc	see \`:class:\`logging. ↪NullHandler\`\`	see logging.NullHandler
External page	\`https://github.com\`	https://github.com

1.11.3 Headers

```
##########################
Docs guidelines
##########################
.. part header


================
Headers
================
.. chapter header
```

Section header

```
----------------
Section header
----------------
```

Subsection header

```
Subsection header
------------------
```

Paragraph header

```
Paragraph header
"""""""""""""""""
```

Rubric

```
.. rubric:: Rubric
```

2

API reference

Note: Almost all public classes are imported into the first package level on its initialization, i.e., "short" forms like `from pytermor import ColorRGB` are supported, not only "full forms" as `from pytermor.color import ColorRGB`.

`pytermor.cv = <pytermor.cval.CVAL object>`

Shortcut to *CVAL()* color registry.
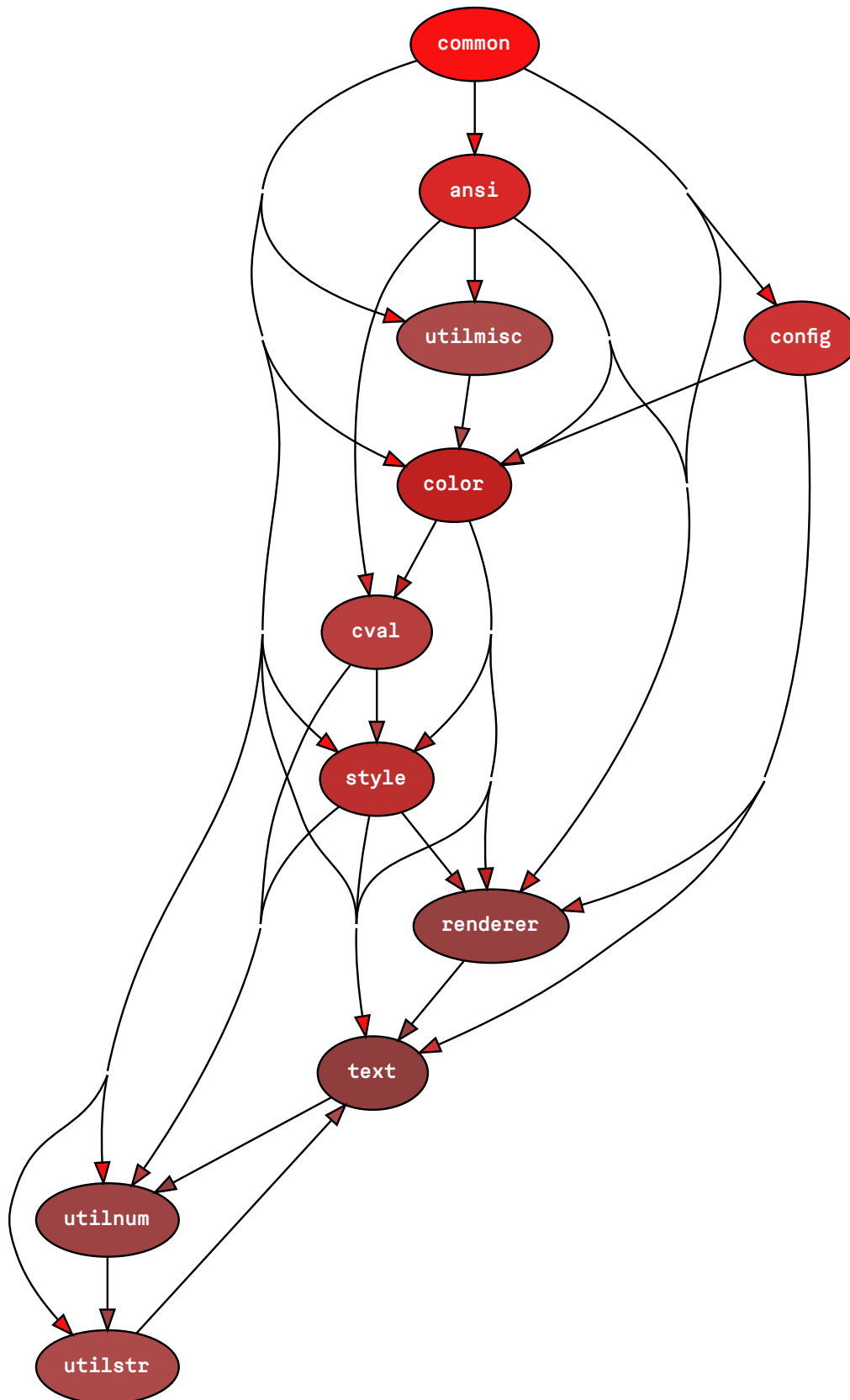
Fig. 1: Module dependency graph

ansi	Classes for working with ANSI sequences on lower level.
color	Color main classes and helper functions.
common	Shared code suitable for the package as well as any other.
config	Library fine tuning.
cval	Color preset list.
renderer	Output formatters.
style	
text	"Front-end" module of the library.
utilmisc	A
utilnum	utilnum
utilstr	Formatters for prettier output and utility classes to avoid writing boilerplate code when dealing with escape sequences.

2.1 pytermor.ansi

Classes for working with ANSI sequences on 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 cleraing);

- OSC (Operating System Command) sequences (varoius system commands).

Important: blah-blah-blah low-level @TODO

Module Attributes

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

Functions

assemble_hyperlink(url[, label])	
	param url
decompose_request_cursor_position(string)	Parse RCP (Report Cursor Position) sequence that generally comes from a terminal as a response to *QCP* sequence and contains a cursor's current row and column.
enclose(opening_seq, string)	
	param opening_seq
get_closing_seq(opening_seq)	
	param opening_seq
make_color_256(code[, bg])	Wrapper for creation of *SequenceSGR* that sets foreground (or background) to one of 256-color palette value.:
make_color_rgb(r, g, b[, bg])	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 "0xRRGGBB". For example, sequence with color of 0xFF3300 can be created with::.
make_erase_in_line([mode])	Create EL (Erase in Line) sequence that erases a part of the line or the entire line.
make_hyperlink_part([url])	
	param url
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*).
make_set_cursor_x_abs([x])	Create CHA (Cursor Horizontal Absolute) sequence that sets cursor horizontal position, or column, to x.

Classes

ISequence(*params)	Abstract ancestor of all escape sequences.
ISequenceFe(*params)	Wide range of sequence types that includes *CSI*, *OSC* and more.
IntCode(value)	Complete or almost complete list of reliably working SGR param integer codes.
SeqIndex()	Registry of static sequence presets.
SequenceCSI(terminator, short_name, *params)	Class representing CSI-type ANSI escape sequence.
SequenceOSC(*params)	OSC-type sequence.
SequenceSGR(*args)	Class representing SGR-type escape sequence with varying amount of parameters.
SequenceST(*params)	String Terminator sequence (ST).

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

> Bases: `Sized`

> Abstract ancestor of all escape sequences.

> > **Parameters**

> **params** (*int | str*) – Sequence internal parameters; amount varies depending on sequence type.

assemble()

> Build up actual byte sequence and return as an ASCII-encoded string.
>
> > **Return type**
> >
> > > str

property params: List[int | str]

> Return internal params as array.

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

> Bases: *ISequence*
>
> Wide range of sequence types that includes *CSI*, *OSC* and more.
>
> All subtypes of this sequence start with ESC plus ASCII byte from 0x40 to 0x5F (@, [, \,], _, ^ and capital letters A-Z).
>
> > **Parameters**
> >
> > > **params** (*int | str*) – Sequence internal parameters; amount varies depending on sequence type.
>
> **assemble()**
>
> > Build up actual byte sequence and return as an ASCII-encoded string.
> >
> > > **Return type**
> > >
> > > > str
>
> **property params: List[int | str]**
>
> > Return internal params as array.

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

> Bases: *ISequenceFe*
>
> String Terminator sequence (ST). Terminates strings in other control sequences. Encoded as ESC \ (0x1B 0x5C).
>
> > **Parameters**
> >
> > > **params** (*int | str*) – Sequence internal parameters; amount varies depending on sequence type.
>
> **assemble()**
>
> > Build up actual byte sequence and return as an ASCII-encoded string.
> >
> > > **Return type**
> > >
> > > > str
>
> **property params: List[int | str]**
>
> > Return internal params as array.

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

> Bases: *ISequenceFe*
>
> OSC-type sequence. Starts a control string for the operating system to use. Encoded as ESC], plus params separated by ;, and terminated with *SequenceST*.
>
> > **Parameters**
> >
> > > **params** (*int | str*) – Sequence internal parameters; amount varies depending on sequence type.
>
> **assemble()**
>
> > Build up actual byte sequence and return as an ASCII-encoded string.

> **Return type**
>> str

property params: List[int | str]
> Return internal params as array.

class pytermor.ansi.**SequenceCSI**(*terminator*, *short_name*, **params*)

> Bases: *ISequenceFe*

> 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.

```
>>> make_erase_in_line().assemble()
'[0K'
```

> **Parameters**
>> - **terminator** (*str*) –
>> - **short_name** (*str*) –
>> - **params** (*int*) –

assemble()
> Build up actual byte sequence and return as an ASCII-encoded string.

>> **Return type**
>>> str

property params: List[int | str]
> Return internal params as array.

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

> 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[96,4,1]>
```

> 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 was specifically implemented to translate into empty string and not into ESC [m, which would have made sense, but also would be entangling, as this sequence is the equivalent of ESC [0m – 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[34,3]>
```

> **Parameters**
>> * **args** (`str` | `int` | 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;
>>>
>>> – another *SequenceSGR* instance (params will be extracted).
>>
>> * **terminator** –
>> * **short_name** –
>> * **params** –

assemble()
> Build up actual byte sequence and return as an ASCII-encoded string.
>
>> **Return type**
>>> str

property params: List[int]

> **Returns**
>> Sequence params as integers or *IntCode* instances.

pytermor.ansi.**NOOP_SEQ** = <SGR[NOP]>

> Special sequence in case you *have to* provide one or another SGR, but do 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
[]
```

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

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* is 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 sequence presets.

RESET = <SGR[0]>
> Hard reset sequence.

BOLD = <SGR[1]>
> Bold or increased intensity.

DIM = <SGR[2]>
> Faint, decreased intensity.

ITALIC = <SGR[3]>
> Italic *(not widely supported)*.

UNDERLINED = <SGR[4]>
> Underline.

BLINK_SLOW = <SGR[5]>
> Set blinking to < 150 cpm.

BLINK_FAST = <SGR[6]>
> Set blinking to 150+ cpm *(not widely supported)*.

INVERSED = <SGR[7]>
> Swap foreground and background colors.

HIDDEN = <SGR[8]>
> Conceal characters *(not widely supported)*.

CROSSLINED = <SGR[9]>
> Strikethrough.

DOUBLE_UNDERLINED = <SGR[21]>
> Double-underline. *On several terminals disables BOLD instead*.

OVERLINED = <SGR[53]>
> Overline *(not widely supported)*.

BOLD_DIM_OFF = <SGR[22]>
> Disable `BOLD` and `DIM` attributes.
>
> *Special aspects. . . It's impossible to reliably disable them on a separate basis.*

`ITALIC_OFF = <SGR[23]>`

> Disable italic.

`UNDERLINED_OFF = <SGR[24]>`

> Disable underlining.

`BLINK_OFF = <SGR[25]>`

> Disable blinking.

`INVERSED_OFF = <SGR[27]>`

> Disable inversing.

`HIDDEN_OFF = <SGR[28]>`

> Disable conecaling.

`CROSSLINED_OFF = <SGR[29]>`

> Disable strikethrough.

`OVERLINED_OFF = <SGR[55]>`

> Disable overlining.

`BLACK = <SGR[30]>`

> Set text color to 0x000000.

`RED = <SGR[31]>`

> Set text color to 0x800000.

`GREEN = <SGR[32]>`

> Set text color to 0x008000.

`YELLOW = <SGR[33]>`

> Set text color to 0x808000.

`BLUE = <SGR[34]>`

> Set text color to 0x000080.

`MAGENTA = <SGR[35]>`

> Set text color to 0x800080.

`CYAN = <SGR[36]>`

> Set text color to 0x008080.

`WHITE = <SGR[37]>`

> Set text color to 0xC0C0C0.

`COLOR_OFF = <SGR[39]>`

> Reset foreground color.

`BG_BLACK = <SGR[40]>`

> Set background color to 0x000000.

`BG_RED = <SGR[41]>`

> Set background color to 0x800000.

`BG_GREEN = <SGR[42]>`

> Set background color to 0x008000.

`BG_YELLOW = <SGR[43]>`

> Set background color to 0x808000.

`BG_BLUE = <SGR[44]>`

> Set background color to 0x000080.

BG_MAGENTA = <SGR[45]>

> Set background color to 0x800080.

BG_CYAN = <SGR[46]>

> Set background color to 0x008080.

BG_WHITE = <SGR[47]>

> Set background color to 0xC0C0C0.

BG_COLOR_OFF = <SGR[49]>

> Reset background color.

GRAY = <SGR[90]>

> Set text color to 0x808080.

HI_RED = <SGR[91]>

> Set text color to 0xFF0000.

HI_GREEN = <SGR[92]>

> Set text color to 0x00FF00.

HI_YELLOW = <SGR[93]>

> Set text color to 0xFFFF00.

HI_BLUE = <SGR[94]>

> Set text color to 0x0000FF.

HI_MAGENTA = <SGR[95]>

> Set text color to 0xFF00FF.

HI_CYAN = <SGR[96]>

> Set text color to 0x00FFFF.

HI_WHITE = <SGR[97]>

> Set text color to 0xFFFFFF.

BG_GRAY = <SGR[100]>

> Set background color to 0x808080.

BG_HI_RED = <SGR[101]>

> Set background color to 0xFF0000.

BG_HI_GREEN = <SGR[102]>

> Set background color to 0x00FF00.

BG_HI_YELLOW = <SGR[103]>

> Set background color to 0xFFFF00.

BG_HI_BLUE = <SGR[104]>

> Set background color to 0x0000FF.

BG_HI_MAGENTA = <SGR[105]>

> Set background color to 0xFF00FF.

BG_HI_CYAN = <SGR[106]>

> Set background color to 0x00FFFF.

BG_HI_WHITE = <SGR[107]>

> Set background color to 0xFFFFFF.

HYPERLINK = <OSC[8]>

> Create a hyperlink in the text *(supported by limited amount of terminals)*. Note that for a working hyperlink you'll need two sequences, not just one.
>
> **See also:**
>
> [make_hyperlink_part()](#) and [assemble_hyperlink()](#).

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.**make_set_cursor_x_abs**(*x=1*)

> Create CHA sequence that sets cursor horizontal position, or column, to x.
>
> **Parameters**
>   **x** (*int*) – New cursor horizontal position.
>
> **Example**
>   ESC [1G
>
> **Return type**
>   [SequenceCSI](#)

pytermor.ansi.**make_erase_in_line**(*mode=0*)

> Create EL sequence that erases a part of the line or the entire line. 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 beginning of the line.
>
> - If set to 2, clear the entire line.
>
> **Example**
>   ESC [0K
>
> **Return type**
>   [SequenceCSI](#)

pytermor.ansi.**make_query_cursor_position**()

> Create QCP sequence that requests an output device to respond with a structure containing current cursor coordinates (*RCP*).

> **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.ansi.**decompose_request_cursor_position**(*string*)

Parse RCP sequence that generally comes from a terminal as a response to *QCP* sequence and contains a cursor's current row and column.

Note: As the library in general provides sequence assembling methods, but not the disassembling ones, there is no dedicated class for RCP sequences yet.

```
>>> decompose_request_cursor_position('[18;2R')
(18, 2)
```

Parameters
    string (*str*) – Terminal response with a sequence.

Returns
    Current row and column if the expected sequence exists in `string`, *None* otherwise.

Return type
    Optional[*Tuple*[int, int]]

pytermor.ansi.**make_color_256**(*code*, *bg=False*)

Wrapper for creation of *SequenceSGR* that sets foreground (or background) to one of 256-color palette value.:

```
>>> make_color_256(141)
<SGR[38,5,141]>
```

See also:

Color256 class.

Parameters

- **code** (*int*) – Index of the color in the palette, 0 – 255.

- **bg** (*bool*) – Set to *True* to change the background color (default is foreground).

Example
    ESC [38;5;141m

Return type
    SequenceSGR

pytermor.ansi.**make_color_rgb**(*r*, *g*, *b*, *bg=False*)

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 "0xRRGGBB". For example, sequence with color of 0xFF3300 can be created with:

```
>>> make_color_rgb(255, 51, 0)
<SGR[38,2,255,51,0]>
```

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.

- **bg** (`bool`) – Set to *True* to change the background color (default is foreground).

Example
ESC [38;2;255;51;0m

Return type
SequenceSGR

pytermor.ansi.**make_hyperlink_part**(*url=None*)

Parameters
url (`Optional[str]`) –

Example
ESC]8;;http://localhost ESC \

Return type
SequenceOSC

pytermor.ansi.**assemble_hyperlink**(*url*, *label=None*)

Parameters

- **url** (`str`) –

- **label** (`Optional[str]`) –

Example
ESC]8;;http://localhost ESC \Text ESC]8;; ESC \

Return type
str

2.2 pytermor.color

Color main classes and helper functions.

Module Attributes

CDT	CDT (Color descriptor type) represents a RGB color value.
CT	Any non-abstract `IColor` type.
NOOP_COLOR	Special `IColor` instance always rendering into empty string.
DEFAULT_COLOR	Special `IColor` instance rendering to SGR sequence telling the terminal to reset fg or bg color; same for *TmuxRenderer*.

Functions

approximate(hex_value[, color_type, max_results])	Search for nearest to `hex_value` colors of specified `color_type` and return the first `max_results` of them.
find_closest(hex_value[, color_type])	Search and return nearest to `hex_value` instance of specified `color_type`.
resolve_color(subject[, color_type])	Case-insensitive search through registry contents.

Classes

ApxResult(color, distance)	Approximation result.
Color16(*args, **kwargs)	Variant of a `IColor` operating within the most basic color set -- **Xterm-16**.
Color256(*args, **kwargs)	Variant of a `IColor` operating within relatively modern **Xterm-256** indexed color table.
ColorRGB(*args, **kwargs)	Variant of a `IColor` operating within RGB color space.
DefaultColor(*args, **kwargs)	
IColor(*args, **kwargs)	Abstract superclass for other `Colors`.

Exceptions

ColorCodeConflictError(code, existing_color, ...)	
ColorNameConflictError(tokens, ...)	

pytermor.color.**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: *ANSI preset list* and *es7s color collection*;
>
> - *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.color.**CT**

> Any non-abstract `IColor` type.
>
> alias of TypeVar('CT', bound=IColor)

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

> Bases: `Generic[CT]`
>
> Approximation result.

color: CT
>   Found `IColor` instance.

distance: int
>   Squared sRGB distance from this instance to the approximation target.

property distance_real: float
>   Actual distance from instance to target:

$$distance_{real} = \sqrt{distance}$$

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

>   Bases: `IColor`
>
>   Variant of a `IColor` 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 *Color16 presets*).
>
> ---
>   **Note:** Arguments `register`, `index` and `aliases` are *kwonly*-type args.
>
> ---
>
>   **Parameters**
>   - **hex_value** (*int*) – Color RGB value, e.g. 0x800000.
>   - **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.
>   - **index** (*bool*) – If *True*, add color to approximation index.
>   - **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.

classmethod get_by_code(*code*)

>   Get a *Color16* instance with specified code. Only *foreground* (=text) colors are indexed, therefore it is impossible to look up for a *Color16* with given background color.
>
>   **Parameters**
>   >   **code** (*int*) – Foreground integer code to look up for (see *Color16 presets*).
>
>   **Raises**
>   >   **KeyError** – If no color with specified code is found.
>
>   **Return type**
>   >   Color16

to_sgr(*bg*, *upper_bound=None*)

>   Make an *SGR sequence* out of `IColor`. Used by *SgrRenderer*.
>
>   **Parameters**
>   - **bg** (*bool*) – Set to *True* if required SGR should change the background color, or *False* for the foreground (=text) color.
>   - **upper_bound** (*Optional[Type[pytermor.color.IColor]]*) – Required result `IColor` 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(*bg*)

> 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**
> > **bg** (`bool`) – Set to *True* if required tmux directive should change the background color, or *False* for the foreground (=text) color.
> >
> > **Return type**
> > str

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

> Search for the colors nearest to `hex_value` and return the first `max_results`.
>
> > **See**
> > *color.approximate()* for the details
> >
> > **Parameters**
> >
> > - **hex_value** (`int`) – Target RGB value.
> >
> > - **max_results** (`int`) – Result limit.
> >
> > **Return type**
> > *List*[ApxResult[pytermor.color.CT]]

classmethod find_closest(*hex_value*)

> Search and return nearest to `hex_value` color instance.
>
> > **See**
> > *color.find_closest()* for the details
> >
> > **Parameters**
> > **hex_value** (`int`) – Target RGB value.
> >
> > **Return type**
> > pytermor.color.CT

format_value(*prefix='0x'*)

> Format color value as "0xRRGGBB".
>
> > **Parameters**
> > **prefix** (`str`) – Can be customized.
> >
> > **Return type**
> > str

property hex_value: int

> Color value, e.g. 0x3AEB0C.

property name: str | None

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

classmethod resolve(*name*)

> Case-insensitive search through registry contents.
>
> > **See**
> > *resolve_color()* for the details
> >
> > **Parameters**
> > **name** (`str`) – IColor name to search for.
> >
> > **Return type**
> > pytermor.color.CT

to_hsv()

>   Wrapper around *hex_to_hsv()* for concrete instance.

>> **See**
>>   hex_to_hsv() for the details

>> **Return type**
>>   *Tuple*[float, float, float]

to_rgb()

>   Wrapper around *to_rgb()* for concrete instance.

>> **See**
>>   to_rgb() for the details

>> **Return type**
>>   *Tuple*[int, int, int]

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

>   Bases: `IColor`

>   Variant of a `IColor` operating within relatively modern **Xterm-256** indexed color table. Represents SGR complex codes `38;5;*` and `48;5;*` (see *Color256 presets*).

>   ---

>   **Note:** Arguments `register`, `index`, `aliases` and `color16_equiv` are *kwonly*-type args.

>   ---

>> **Parameters**
>>   - **hex_value** (*int*) – Color RGB value, e.g. 0x5f0000.
>>   - **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.
>>   - **index** (*bool*) – If *True*, add color to approximation index.
>>   - **aliases** (*t.List[str]*) – Alternative color names (used in *resolve_color()*).
>>   - **color16_equiv** (*Color16*) – *Color16* counterpart (applies only to codes 0-15).

>   **to_sgr**(*bg*, *upper_bound=None*)

>>   Make an *SGR sequence* out of `IColor`. Used by *SgrRenderer*.

>>   Each `IColor` 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**
>>>   - **bg** (*bool*) – Set to *True* if required SGR should change the background color, or *False* for the foreground (=text) color.
>>>   - **upper_bound** (*Optional[Type[pytermor.color.IColor]]*) – Required result `IColor` type upper boundary, i.e., the maximum acceptable color class, which will be the basis for SGR being made.

> **Return type**
> SequenceSGR

to_tmux(*bg*)

> 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**
> > **bg** (`bool`) – Set to *True* if required tmux directive should change the background color, or *False* for the foreground (=text) color.
> >
> > **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 *Color256 presets*).
> >
> > **Raises**
> > **KeyError** – If no color with specified code is found.
> >
> > **Return type**
> > Color256

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

> Search for the colors nearest to `hex_value` and return the first `max_results`.
>
> > **See**
> > *color.approximate()* for the details
> >
> > **Parameters**
> > - **hex_value** (`int`) – Target RGB value.
> > - **max_results** (`int`) – Result limit.
> >
> > **Return type**
> > *List*[ApxResult[pytermor.color.CT]]

classmethod find_closest(*hex_value*)

> Search and return nearest to `hex_value` color instance.
>
> > **See**
> > *color.find_closest()* for the details
> >
> > **Parameters**
> > **hex_value** (`int`) – Target RGB value.
> >
> > **Return type**
> > pytermor.color.CT

format_value(*prefix='0x'*)

> Format color value as "0xRRGGBB".
>
> > **Parameters**
> > **prefix** (`str`) – Can be customized.
> >
> > **Return type**
> > str

property hex_value: int

> Color value, e.g. 0x3AEB0C.

property name: str | None

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

classmethod resolve(*name*)

Case-insensitive search through registry contents.

> **See**
>
> > *resolve_color()* for the details
>
> **Parameters**
>
> > **name** (*str*) – IColor name to search for.
>
> **Return type**
>
> > pytermor.color.CT

to_hsv()

Wrapper around *hex_to_hsv()* for concrete instance.

> **See**
>
> > hex_to_hsv() for the details
>
> **Return type**
>
> > *Tuple*[float, float, float]

to_rgb()

Wrapper around *to_rgb()* for concrete instance.

> **See**
>
> > to_rgb() for the details
>
> **Return type**
>
> > *Tuple*[int, int, int]

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

Bases: IColor

Variant of a IColor operating within RGB color space. Presets include *es7s named 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.

Note: Arguments register, index, aliases and variation_map are *kwonly*-type args.

> **Parameters**
>
> > - **hex_value** (*int*) – Color RGB value, e.g. 0x73A9C2.
> >
> > - **name** (*str*) – Name of the color, e.g. "moonstone-blue".
> >
> > - **register** (*bool*) – If *True*, add color to registry for resolving by name.
> >
> > - **index** (*bool*) – If *True*, add color to approximation index.
> >
> > - **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(*bg*, *upper_bound=None*)

Make an *SGR sequence* out of IColor. Used by *SgrRenderer*.

> **Parameters**
>
> > - **bg** (*bool*) – Set to *True* if required SGR should change the background color, or *False* for the foreground (=text) color.

- **upper_bound** (*Optional[Type[pytermor.color.IColor]]*) – Required result IColor 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(*bg*)

> 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**
> **bg** (*bool*) – Set to *True* if required tmux directive should change the background color, or *False* for the foreground (=text) color.
>
> **Return type**
> str

property base: CT | None

> Parent color for color variations. Empty for regular colors.

property variations: Dict[str, pytermor.color.CT]

> 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(*hex_value*, *max_results=1*)

> Search for the colors nearest to `hex_value` and return the first `max_results`.
>
> **See**
> `color.approximate()` for the details
>
> **Parameters**
>
> - **hex_value** (*int*) – Target RGB value.
>
> - **max_results** (*int*) – Result limit.
>
> **Return type**
> *List*[ApxResult[pytermor.color.CT]]

classmethod find_closest(*hex_value*)

> Search and return nearest to `hex_value` color instance.
>
> **See**
> `color.find_closest()` for the details
>
> **Parameters**
> **hex_value** (*int*) – Target RGB value.
>
> **Return type**
> pytermor.color.CT

format_value(*prefix='0x'*)

> Format color value as "0xRRGGBB".
>
> **Parameters**
> **prefix** (*str*) – Can be customized.
>
> **Return type**
> str

property hex_value: int

> Color value, e.g. 0x3AEB0C.

property name: str | None

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

classmethod resolve(*name*)

> Case-insensitive search through registry contents.
>
> > **See**
> >
> > > *resolve_color()* for the details
> >
> > **Parameters**
> >
> > > **name** (`str`) – IColor name to search for.
> >
> > **Return type**
> >
> > > pytermor.color.CT

to_hsv()

> Wrapper around *hex_to_hsv()* for concrete instance.
>
> > **See**
> >
> > > hex_to_hsv() for the details
> >
> > **Return type**
> >
> > > *Tuple*[float, float, float]

to_rgb()

> Wrapper around *to_rgb()* for concrete instance.
>
> > **See**
> >
> > > to_rgb() for the details
> >
> > **Return type**
> >
> > > *Tuple*[int, int, int]

pytermor.color.**NOOP_COLOR** = <_NoopColor[NOP]>

> Special `IColor` instance always rendering into empty string.

pytermor.color.**DEFAULT_COLOR** = <DefaultColor[DEF]>

> Special `IColor` 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 which you don't need, but at the
> same time you don't actually want to set up any value whatsoever.

Important: *None* and *NOOP_COLOR* are always treated as placeholders for fallback values, i.e., they can't
be used as re-setters – that's what *DEFAULT_COLOR* is for.

```
>>> DEFAULT_COLOR.to_sgr(bg=False)
<SGR[39]>
```

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

> Case-insensitive search through registry contents. Search is performed for `IColor` instance named as speci-
> fied in `subject` argument, and of specified `color_type`, or for any type if argument is omitted: first it will
> be performed in the registry of *Color16* class, then – in *Color256*, and, if previous two were unsuccessful,
> in the largest *ColorRGB* registry. Therefore, the return value could be any of these types:
>
> ```
> >>> resolve_color('red')
> <Color16[#31,800000?,red]>
> ```
>
> If `color_type` is *ColorRGB* or if it is omitted, there is one more way to specify a color: in form of a hex-
> adecimal value "#RRGGBB" (or in short form, as "#RGB"), as well as just use an *int* in [0x00; 0xFFFFFF]
> range. In this case no actual searching is performed, and a new nameless instance of *ColorRGB* is created
> and returned.

```
>>> resolve_color("#333")
<ColorRGB[333333]>
>>> resolve_color(0xfafef0)
<ColorRGB[FAFEF0]>
```

Color names are stored in registries as tokens, which allows to use any form of input and get the correct result regardless. The only requirement is to split the words in any matter, so that tokenizer could distinguish the words from each other:

```
>>> 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 of registries
```

> **Parameters**
>
> - **subject** (*str*|*int*) – IColor name or hex value to search for. See *CDT*.
>
> - **color_type** (*Optional[Type[pytermor.color.CT]]*) – Target color type (*Color16*, *Color256* or *ColorRGB*).
>
> **Raises**
> > **LookupError** – If nothing was found in either of registries.
>
> **Returns**
> > IColor instance with specified name or value.
>
> **Return type**
> > pytermor.color.CT

pytermor.color.**find_closest**(*hex_value*, *color_type=None*)

> Search and return nearest to `hex_value` instance of specified `color_type`. If `color_type` is omitted, search for the closest *Color256* element.
>
> 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.
>
> ---
>
> **Note:** 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**
>
> - **hex_value** (*int*) – Target color RGB value.
>
> - **color_type** (*Optional[Type[pytermor.color.CT]]*) – Target color type (*Color16*, *Color256* or *ColorRGB*).
>
> **Returns**
> > Nearest to `hex_value` color instance of specified type.
>
> **Return type**
> > pytermor.color.CT

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

> Search for nearest to `hex_value` colors of specified `color_type` and return the first `max_results` of them. If `color_type` is omitted, search for the closest *Color256* elements. 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 `IColor` 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**
>>
>> - **hex_value** (*int*) – Target color RGB value.
>>
>> - **color_type** (*Optional[Type[pytermor.color.CT]]*) – Target color type (*Color16*, *Color256* or *ColorRGB*).
>>
>> - **max_results** (*int*) – Return no more than `max_results` items.
>>
>> **Returns**
>>
>> Pairs of closest `IColor` 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[pytermor.color.CT]]

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

> Bases: `Exception`

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

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

> Bases: `Exception`

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

2.3 pytermor.common

Shared code suitable for the package as well as any other.

Module Attributes

RGB(red, green, blue)	RGB.
HSV(hue, saturation, value)	HSV.
ALIGN_LEFT	Left align (add padding on the right side, if necessary).
ALIGN_RIGHT	Right align (add padding on the left side, if necessary).
ALIGN_CENTER	Center align (add paddings on both sides evenly, if necessary).

Functions

[*chunk*](items, size)	Split item list into chunks of size `size` and return these chunks as *tuples*.
[*flatten*](items)	
[*flatten1*](items)	Take a list of nested lists and unpack all nested elements one level up.
[*get_preferable_wrap_width*]([force_width])	Return preferable terminal width for comfort reading of wrapped text (max=120).
[*get_qname*](obj)	Convenient method for getting a class name for class instances as well as for the classes themselves.
[*get_terminal_width*]([fallback, pad])	Return current terminal width with an optional "safety buffer", which ensures that no unwanted line wrapping will happen.
[*median*](N[, key])	Find the median of a list of values.
[*percentile*](N, percent[, key])	Find the percentile of a list of values.

Classes

[*Align*](value)	Align type.
[*ExtendedEnum*](value)	Standard `Enum` with a few additional methods on top.
[*HSV*](hue, saturation, value)	HSV.
[*RGB*](red, green, blue)	RGB.

Exceptions

ArgCountError(actual, *expected)
ArgTypeError(actual_type[, arg_name, fn])
[*ConflictError*]
[*LogicError*]
[*UserAbort*]
[*UserCancel*]

class pytermor.common.**RGB**(*red*, *green*, *blue*)

> Bases: `tuple`
>
> RGB.
>
> **blue**
>> Alias for field number 2
>
> **count**(*value*, */*)
>> Return number of occurrences of value.
>
> **green**
>> Alias for field number 1

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

> Return first index of value.

> Raises ValueError if the value is not present.

red

> Alias for field number 0

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

> Bases: `tuple`

> HSV.

> **count**(*value*, */*)

> > Return number of occurrences of value.

> **hue**

> > Alias for field number 0

> **index**(*value*, *start=0*, *stop=9223372036854775807*, */*)

> > Return first index of value.

> > Raises ValueError if the value is not present.

> **saturation**

> > Alias for field number 1

> **value**

> > Alias for field number 2

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

> Bases: `Enum`

> Standard `Enum` with a few additional methods on top.

> **classmethod list**()

> > Return all enum values as list.

> > ```
> > >>> Align.list()
> > ['<', '>', '^']
> > ```

> **classmethod dict**()

> > Return mapping of all enum keys to corresponding enum values.

> > ```
> > >>> Align.dict()
> > {<Align.LEFT: '<'>: '<', <Align.RIGHT: '>'>: '>', <Align.CENTER: '^'>: '^'}
> > ```

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

> Bases: `str`, *ExtendedEnum*

> Align type.

exception pytermor.common.**UserCancel**

> Bases: `Exception`

> **with_traceback**()

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

exception pytermor.common.**UserAbort**

> Bases: `Exception`

> **with_traceback**()

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

exception pytermor.common.**LogicError**

> Bases: `Exception`

> **with_traceback()**

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

exception pytermor.common.**ConflictError**

> Bases: `Exception`

> **with_traceback()**

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

pytermor.common.**ALIGN_LEFT = Align.LEFT**

> Left align (add padding on the right side, if necessary).

pytermor.common.**ALIGN_RIGHT = Align.RIGHT**

> Right align (add padding on the left side, if necessary).

pytermor.common.**ALIGN_CENTER = Align.CENTER**

> Center align (add paddings on both sides evenly, if necessary).

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

> Convenient method for getting a class name for class instances as well as for the classes themselves. Suitable for debug output in `__repr__` methods, for example.

```
>>> get_qname("aaa")
'str'
>>> get_qname(threading.Thread)
'Thread'
```

> > **Return type**
> >
> > > str

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

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

```
>>> for c in chunk(range(5), 2):
...     print(c)
(0, 1)
(2, 3)
(4,)
```

Parameters

- **items** (*Iterable[T]*) – Input elements.

- **size** (*int*) – Chunk size.

Return type

Iterator[*Tuple*[*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], [[10, 11, 12]]])
[1, 2, 3, 4, 5, 6, [10, 11, 12]]
```

Parameters

items (*Iterable[Iterable[T]]*) – Input lists.

Return type

List[*T*]

pytermor.common.**flatten**(*items*)

Todo: recursrive

Return type

List[*T*]

pytermor.common.**percentile**(*N*, *percent*, *key=<function <lambda>>*)

Find the percentile of a list of values.

Parameters

- **N** (*Sequence[float]*) – List of values. MUST BE already sorted.

- **percent** (*float*) – Float value from 0.0 to 1.0.

- **key** (*Callable[[float], float]*) – Optional key function to compute value from each element of N.

Return type

float

pytermor.common.**median**(*N*, *key=<function <lambda>>*)

Find the median of a list of values. Wrapper around *percentile()* with fixed `percent` argument (=0.5).

Parameters

- **N** (*Sequence[float]*) – List of values. MUST BE already sorted.

- **key** (*Callable[[float], float]*) – Optional key function to compute value from each element of N.

Return type

float

2.4 pytermor.config

Library fine tuning.

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, output_mode, ...])	Configuration variables container.

class pytermor.config.**Config**(*renderer_class=<factory>*, *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 *Configuration* guide section.
>
> **Parameters**
>
> - **renderer_class** (*str*) – renderer_class
>
> - **output_mode** (*str*) – output_mode
>
> - **trace_renders** (*bool*) – Set to *True* to log hex dumps of rendered strings. Note that default logger is logging.NullHandler with WARNING level, so in order to see the traces attached handler is required.
>
> - **prefer_rgb** (*bool*) – By default SGR renderer transforms *Color256* instances to ESC [38;5;<N>m sequences even if True Color support is detected. With this flag set to *True*, the behaviour is different, and *Color256* will be rendered as ESC [38;2;<R>; <G>;<B>m sequence (if True Color is available).

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.

2.5 pytermor.cval

Color preset list.

Classes

CVAL()

---

class pytermor.cval.**CVAL**

2.6 pytermor.renderer

Output formatters. Default global renderer type is *SgrRenderer*.

Functions

---

init_renderer()

Classes

HtmlRenderer()	Translate *Styles* attributes into a rudimentary HTML markup.
IRenderer()	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* attributes into tmux-compatible[7] markup.

class pytermor.renderer.**RendererManager**

Class for global rendering mode setup.

Selecting the renderer can be accomplished in several ways:

a. By using general-purpose functions *text.render()* and *text.echo()* – both have an argument renderer (preferable; introduced in pytermor 2.x).

b. Method *RendererManager.set_default()* sets the default renderer globally. After that calling *text.render()* will automatically invoke a said renderer and apply the required formatting (that is, if renderer argument is left empty).

---

[7] https://man7.org/linux/man-pages/man1/tmux.1.html#STYLES

c. Alternatively, you can use renderer's instance method `render()` directly and avoid messing up with the manager, but that's not recommended and possibly will be deprecated in future versions).

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, use `RendererManager.set_default_format_always()` and then `render()`.

classmethod set_default(*renderer=None*)

Select a global renderer.

> **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 registred as global.

classmethod get_default()

Get global renderer instance (*SgrRenderer*, or the one provided earlier with *set_default()*).

> **Return type**
> IRenderer

classmethod set_default_format_always()

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.

classmethod set_default_format_never()

Shortcut for disabling all output formatting of a global renderer.

class pytermor.renderer.**IRenderer**

Renderer interface.

abstract property is_caching_allowed: bool

Class-level property.

> **Returns**
> *True* if caching of renderer's results makes any sense and *False* otherwise.

abstract property is_format_allowed: bool

> **Returns**
> *True* if renderer is set up to use the formatting 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.

> **Parameters**

- **string** (`str`) – String to format.

- **fmt** (`Optional[pytermor.style.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

abstract **clone**(**args*, ***kwargs*)

> Make a copy of the renderer with the same setup.

  Return type
  > self

class pytermor.renderer.**OutputMode**(*value*)

> Bases: `Enum`

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 *SgrRenderer* constructor documentation 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 `IColor` instances defined in `style` into ANSI control sequence bytes and merges them with input string. Type of resulting *SequenceSGR* depends on type of `IColor` instances in `style` 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 variable `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'
```

> **Parameters**
>> **output_mode** (`OutputMode`) – SGR output mode to use. Valid values are listed in `OutputMode` enum.
>>
>> With `OutputMode.AUTO` the renderer will first check if the output device is a terminal emulator, and use `OutputMode.NO_ANSI` when it is not. Otherwise, the renderer will read TERM environment variable and follow these rules:
>>
>> - `OutputMode.NO_ANSI` if TERM is set to `xterm`.
>>
>> - `OutputMode.XTERM_16` if TERM is set to `xterm-color`.
>>
>> - `OutputMode.XTERM_256` in all other cases.
>>
>> Special case is when TERM equals to `xterm-256color` **and** COLORTERM is either `truecolor` or `24bit`, then `OutputMode.TRUE_COLOR` will be used.

> **property is_caching_allowed: bool**
>
> Class-level property.
>
>> **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 use the formatting 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.
>
>> **Parameters**
>>
>> - **string** (`str`) – String to format.
>>
>> - **fmt** (`Optional[pytermor.style.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**
>>> self

class pytermor.renderer.**TmuxRenderer**

> Bases: *IRenderer*
>
> Translates *Styles* attributes into tmux-compatible[8] markup. tmux[9] is a commonly used terminal multiplexer.

```
>>> TmuxRenderer().render('text', Style(fg='blue', bold=True))
'#[fg=blue bold]text#[fg=default nobold]'
```

property is_caching_allowed: bool

Class-level property.

> **Returns**
> *True* if caching of renderer's results makes any sense and *False* otherwise.

property is_format_allowed: bool

> **Returns**
> Always *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 itself.

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.

> **Parameters**
>
> - **string** (`str`) – String to format.
>
> - **fmt** (`Optional[pytermor.style.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**
> self

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* 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'
```

property is_caching_allowed: bool

Class-level property.

> **Returns**
> *True* if caching of renderer's results makes any sense and *False* otherwise.

property is_format_allowed: bool

> **Returns**
> Nothing to apply → nothing to allow, thus the returned value is always *False*.

render(*string*, *fmt=None*)

Return the `string` argument untouched, don't mind the `fmt`.

> **Parameters**
>
> - **string** (`str`) – String to format ignore.
>
> - **fmt** (`Optional[pytermor.style.FT]`) – Style or color to appl discard.

---

[8] https://man7.org/linux/man-pages/man1/tmux.1.html#STYLES
[9] https://github.com/tmux/tmux

> **Return type**
> > str

clone()

> Make a copy of the renderer with the same setup.

> > **Return type**
> > > self

class pytermor.renderer.**HtmlRenderer**

> Bases: *IRenderer*

Translate *Styles* attributes into a rudimentary HTML markup. All the formatting is inlined into style attribute of the <span> 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>'
```

> **property is_caching_allowed: bool**

> > Class-level property.

> > > **Returns**
> > > *True* if caching of renderer's results makes any sense and *False* otherwise.

> **property is_format_allowed: bool**

> > > **Returns**
> > > Always *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.

> > > **Parameters**
> > > - **string** (*str*) – String to format.
> > > - **fmt** (*Optional[pytermor.style.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**
> > > self

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)'
```

property is_caching_allowed: bool

>Class-level property.

>>**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 use the formatting 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.

>>**Parameters**

>>- **string** (`str`) – String to format.

>>- **fmt** (`Optional[pytermor.style.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**

>>self

set_format_always()

>Force all control sequences to be present in the output.

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.

2.7 pytermor.style

Todo: S

Module Attributes

FT	FT (Format type) is a style descriptor.
NOOP_STYLE	Special style passing the text through without any modifications.

Functions

make_style([fmt])	General *Style* constructor.
merge_styles([base, fallbacks, overwrites])	Bulk style merging method.

Classes

Style([fallback, fg, bg, blink, bold, ...])	Create new text render descriptior.
Styles()	Some ready-to-use styles.

pytermor.style.**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, ~pytermor.color.IColor, Style, None)

class pytermor.style.**Style**(*fallback=None*, *fg=None*, *bg=None*, ***, *blink=None*, *bold=None*, *crosslined=None*, *dim=None*, *double_underlined=None*, *inversed=None*, *italic=None*, *overlined=None*, *underlined=None*, *class_name=None*)

> Create new text render descriptior.
>
> Both `fg` and `bg` can be specified as existing `IColor` instance as well as plain *str* or *int* (for the details see *resolve_color()*).

```
>>> Style(fg='green', bold=True)
<Style[green:NOP +BOLD]>
>>> Style(bg=0x0000ff)
<Style[NOP:0000FF]>
>>> Style(fg='DeepSkyBlue1', bg='gray3')
<Style[X39[00AFFF]:X232[080808]]>
```

> Attribute merging from `fallback` works this way:
>
> - If constructor argument is *not* empty (`True`, `False`, `IColor` etc.), keep it as attribute value.
>
> - If constructor argument is empty (*None*), 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.

Note: Both empty (i.e., *None*) attributes of type `IColor` 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*.

Note: All arguments except `fallback`, `fg` and `bg` are *kwonly*-type args.

> **Parameters**

- **fallback** (`Style`) – Copy unset attributes from speicifed fallback style. See *merge_fallback()*.

- **fg** (`CDT | IColor`) – Foreground (i.e., text) color.

- **bg** (`CDT | IColor`) – Background color.

- **blink** (`bool`) – Blinking effect; *supported by limited amount of Renderers*.

- **bold** (`bool`) – Bold or increased intensity.

- **crosslined** (`bool`) – Strikethrough.

- **dim** (`bool`) – Faint, decreased intensity.

- **double_underlined** (`bool`) – Faint, decreased intensity.

- **inversed** (`bool`) – Swap foreground and background colors.

- **italic** (`bool`) – Italic.

- **overlined** (`bool`) – Overline.

- **underlined** (`bool`) – Underline.

- **class_name** (`str`) – Arbitary string used by some _get_renderers, e.g. by `HtmlRenderer`.

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

---

> **Returns**
> self
>
> **Return type**
> Style

flip()

Swap foreground color and background color.

> **Returns**
> self
>
> **Return type**
> Style

clone()

> **Returns**
> self
>
> **Return type**
> Style

merge_fallback(*fallback*)

Merge current style with specified `fallback` *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 1: 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.
>
> **Returns**
> > self
>
> **Return type**
> > Style

merge_overwrite(*overwrite*)

> Merge current style with specified `overwrite` *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 2: 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.
>
> **Returns**
> > self

`pytermor.style.NOOP_STYLE = <_NoOpStyle[NOP]>`

>Special style passing the text through without any modifications.

>---

>**Important:** This class is immutable, i.e. *LogicError* will be raised upon an attempt to modify any of its attributes, which can lead to schrödinbugs:

>```
>st1.merge_fallback(Style(bold=True), [Style(italic=False)])
>```

>If `st1` is a regular style instance, the statement above will always work (and pass the tests), but if it happens to be a *NOOP_STYLE*, this will result in an exception. To protect from this outcome one could merge styles via frontend method *merge_styles* only, which always makes a copy of base argument and thus cannot lead to such behaviour.

>---

class `pytermor.style.Styles`

>Some ready-to-use styles. Can be used as examples.

>`WARNING = <Style[yellow:NOP]>`

>`WARNING_LABEL = <Style[yellow:NOP +BOLD]>`

>`WARNING_ACCENT = <Style[hi-yellow:NOP]>`

>`ERROR = <Style[red:NOP]>`

>`ERROR_LABEL = <Style[red:NOP +BOLD]>`

>`ERROR_ACCENT = <Style[hi-red:NOP]>`

>`CRITICAL = <Style[hi-white:X160[D70000]]>`

>`CRITICAL_LABEL = <Style[hi-white:X160[D70000] +BOLD]>`

>`CRITICAL_ACCENT = <Style[hi-white:X160[D70000] +BOLD +BLINK]>`

`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`(*base=<_NoOpStyle[NOP]>*, ***, *fallbacks=()*, *overwrites=()*)

>Bulk style merging method. First merge `fallbacks` *styles* with the `base` in the same order they are iterated, using *merge_fallback()* algorithm; then do the same for `overwrites` styles, but using *merge_overwrite()* merge method.

>The original *base* is left untouched, as all the operations are performed on its clone.

Listing 3: Dual mode merge diagram

```
                          +-----+                               +-----+
          >---->---->----->------->        >-------(B)-update--------------->   |
          |    |    |     |    |        |  |                            | | R |
          |    |    |     |    |        | B >=>Ø   [0]>-[1]>-[2]> .. -[n]>  | | E |
[0]>-[1]>-[2]>- .. >-[n]>->Ø  | A >=>Ø    |   |   |       |    | | S |
          |    |   >- .. ------>Ø  | S >=>Ø    >---(D)-update---->---> U |
          |   >----- .. ------>Ø  | E | (C) drop                  | | L |
          >--------- .. ------>Ø  |   |================(E)=keep=======> T |
                         (A)        |   |                            | | |
               FALLBACKS   drop   +-----+           OVERWRITES        +-----+
```
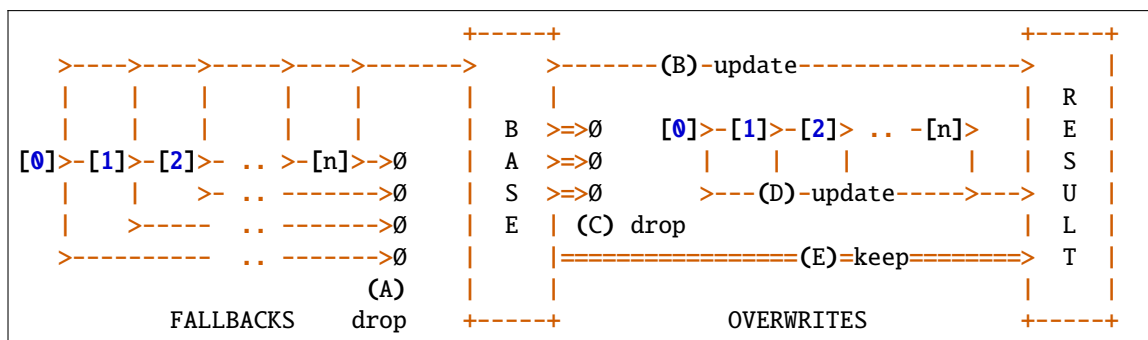
The key actions are marked with (**A**) to (**E**) letters. In reality the algorithm works in slightly different order, but the exact scheme would be less illustrative.

(**A**),(**B**)

Iterate `fallback` styles one by one; discard all the attributes of a current `fallback` style, that are already set in `base` style (i.e., that are not *Nones*). Update all `base` 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 4: Fallback merge algorithm example №1

```
>>> base = Style(fg='red')
>>> fallbacks = [Style(fg='blue'), Style(bold=True),
→Style(bold=False)]
>>> merge_styles(base, fallbacks=fallbacks)
<Style[red:NOP +BOLD]>
```

In the example above:

- the first fallback will be ignored, as `fg` is already set;

- the second fallback will be applied (`base` 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, `base` 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 base/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 5: Fallback merge algorithm example №2

```
>>> merge_styles(Style(fg=cv.BLUE), fallbacks=[Style(bold=True)])
<Style[blue:NOP +BOLD]>
>>> merge_styles(Style(fg=cv.GRAY, bold=False),
↪fallbacks=[Style(bold=True)])
<Style[gray:NOP -BOLD]>
```

(C),(D),(E)

Iterate `overwrite` styles one by one; discard all the attributes of a `base` style that have a non-empty counterpart in `overwrite` style, and put corresponding `overwrite` attribute values instead of them. Keep `base` 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 6: Overwrite merge algorithm example

```
>>> base = Style(fg='red')
>>> overwrites = [Style(fg='blue'), Style(bold=True),
↪Style(bold=False)]
>>> merge_styles(base, overwrites=overwrites)
<Style[blue:NOP -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

- **base** (*Style*) – Basis style instance.

- **fallbacks** (*Iterable[Style]*) – List of styles to be used as a backup attribute storage, when there is no value set for the attribute in question. Uses *merge_fallback()* merging strategy.

- **overwrites** (*Iterable[Style]*) – List of styles to be used as attribute storage force override regardless of actual *base* attribute valuse.

Returns

Clone of `base` style with all specified styles merged into.

Return type

Style

2.8 pytermor.text

"Front-end" module of the library. Contains 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.

Module Attributes

RT	RT (Renderable type) includes regular *str*s as well as *IRenderable* implementations.

Functions

as_fragments(string)	
distribute_padded()	**param max_len**
echo([string, fmt, renderer, ...])	.
echoi([string, fmt, renderer, ...])	echo inline
render([string, fmt, renderer, ...])	.
wrap_sgr(raw_input, width[, indent_first, ...])	A workaround to make standard library textwrap.wrap() more friendly to an SGR-formatted strings.

Classes

Fragment([string, fmt, close_this, close_prev])	<Immutable>
FrozenText()	**param align** default is left
IRenderable()	I
SimpleTable(*rows[, width, sep, border_st])	Table class with dynamic (not bound to each other) rows.
TemplateEngine([custom_styles])	
Text()	
TextComposite(*parts)	Simple class-container supporting concatenation of any *IRenderable* instances with each other without any extra logic on top of it.

pytermor.text.**RT**

> RT includes regular *str*s as well as *IRenderable* implementations.
>
> alias of TypeVar('RT', str, IRenderable)

---

class pytermor.text.**IRenderable**

    Bases: `Sized`, `ABC`

    I

    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*) –

    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**(*string: str, fmt: FT = NOOP_STYLE, *, width: int = None, align: str |* [pytermor.common.Align](#) *= None, fill: str = ' ', overflow: str = '', pad: int = 0, pad_styled: bool = True*)

class pytermor.text.**FrozenText**(**fragments:* [Fragment](#), *width: int = None, align: str |* [pytermor.common.Align](#) *= None, fill: str = ' ', overflow: str = '', pad: int = 0, pad_styled: bool = True*)

Bases: *[IRenderable](#)*

> **Parameters**
>> **align** (*str* | [Align](#)) – default is left

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

class pytermor.text.**Text**(*string: str, fmt: FT = NOOP_STYLE, *, width: int = None, align: str |* [pytermor.common.Align](#) *= None, fill: str = ' ', overflow: str = '', pad: int = 0, pad_styled: bool = True*)

class pytermor.text.**Text**(**fragments:* [Fragment](#), *width: int = None, align: str |* [pytermor.common.Align](#) *= None, fill: str = ' ', overflow: str = '', pad: int = 0, pad_styled: bool = True*)

Bases: *[FrozenText](#)*

set_width(*width*)

> raise NotImplementedError

property allows_width_setup: **bool**

> return False

property has_width: **bool**

> return self._width is not None

render(*renderer=None*)

> pass

>> **Return type**
>>> str

class pytermor.text.**TextComposite**(**parts*)

Bases: *[IRenderable](#)*

Simple class-container supporting concatenation of any [IRenderable](#) instances with each other without any extra logic on top of it. Renders parts joined by an empty string.

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[NOP]>*)

> 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("biiiiiiiiiiiiiiiiiiiiiiiiiiiiiig string"),
...         ],
...         width=30,
...         sep="|"
... ), file=sys.stdout)
|1| word |smol string         |
|2| padd |biiiiiiiiiiiiiiiiii|
```

> Create

> ---

> **Note:** All arguments except **rows* are *kwonly*-type args.

> ---

>> **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*) –

> **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[NOP]>*, *renderer=None*, *parse_template=False*, *,
> *no_log=False*)

.

> **Parameters**
>
> > - **string** (*Union[pytermor.text.RT, Iterable[pytermor.text.RT]]*) – 2
> > - **fmt** (*pytermor.style.FT*) – 2
> > - **renderer** ([IRenderer](#)) – 2
> > - **parse_template** (*bool*) – 2
> > - **no_log** (*bool*) – 2
>
> **Returns**
>
> **Return type**
> > *Union*[str, *List*[str]]

pytermor.text.**echo**(*string=''*, *fmt=<_NoOpStyle[NOP]>*, *renderer=None*, *parse_template=False*, *,
> *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[pytermor.text.RT, Iterable[pytermor.text.RT]]*) –
> > - **fmt** (*pytermor.style.FT*) –
> > - **renderer** (*Optional[*[IRenderer](#)*]*) –
> > - **parse_template** (*bool*) –
> > - **nl** (*bool*) –
> > - **file** (*IO*) –
> > - **flush** (*bool*) –
> > - **wrap** (*bool | int*) –
> > - **indent_first** (*int*) –
> > - **indent_subseq** (*int*) –

pytermor.text.**echoi**(*string=''*, *fmt=<_NoOpStyle[NOP]>*, *renderer=None*, *parse_template=False*, *,
> *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *flush=True*)

> echo inline

> **Parameters**
>
> > - **string** (*Union[pytermor.text.RT, Iterable[pytermor.text.RT]]*) –
> > - **fmt** (*pytermor.style.FT*) –

- **renderer** (*Optional[*[IRenderer]*]*) –

- **parse_template** (*bool*) –

- **file** (*IO*) –

- **flush** (*bool*) –

Returns

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**(*raw_input*, *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

- **raw_input** (*str | list[str]*) –

- **width** (*int*) –

- **indent_first** (*int*) –

- **indent_subseq** (*int*) –

Return type

str

2.9 pytermor.utilmisc

A

Functions

confirm([attempts, default, keymap, prompt, ...])	Ensure the next action is manually confirmed by user.
get_char_width(char, block)	General-purpose method for getting width of a character in terminal columns.
guess_char_width(c)	Determine how many columns are needed to display a character in a terminal.
hex_to_hsv(hex_value)	Transforms `hex_value` in *int* form into named tuple consisting of three floats corresponding to **hue**, **saturation** and **value** channel values respectively.
hex_to_rgb(hex_value)	Transforms `hex_value` in *int* format into a tuple of three integers corresponding to **red**, **blue** and **green** channel value respectively.
hsv_to_hex(h, s, v)	Transforms HSV value in three-floats form (where 0 <= h < 360, 0 <= s <= 1, and 0 <= v <= 1) into an one-integer form.
hsv_to_rgb(h, s, v)	Transforms HSV value in three-floats form (where 0 <= h < 360, 0 <= s <= 1, and 0 <= v <= 1) into RGB three-integer form ([0; 255], [0; 255], [0; 255]).
lab_to_rgb(l_s, a_s, b_s)	@TODO
measure_char_width(char[, clear_after])	Low-level function that returns the exact character width in terminal columns.
rgb_to_hex(r, g, b)	Transforms RGB value in a three-integers form ([0; 255], [0; 255], [0; 255]) to an one-integer form.
rgb_to_hsv(r, g, b)	Transforms RGB value in a three-integers form ([0; 255], [0; 255], [0; 255]) to an HSV in three-floats form such as (0 <= h < 360, 0 <= s <= 1, and 0 <= v <= 1).
total_size(o[, handlers, verbose])	Return the approximate memory footprint of an object and all of its contents.
wait_key([block])	Wait for a key press on the console and return it.

pytermor.utilmisc.**hex_to_rgb**(*hex_value*)

> Transforms `hex_value` in *int* format into a tuple of three integers corresponding to **red**, **blue** and **green** channel value respectively. Values are within [0; 255] range.

```
>>> hex_to_rgb(0x80ff80)
(128, 255, 128)
```

> > **Parameters**
> > > **hex_value** (*int*) – RGB value.
> >
> > **Returns**
> > > R, G, B channel values correspondingly.
> >
> > **Return type**
> > > *Tuple*[int, int, int]

pytermor.utilmisc.**rgb_to_hex**(*r*, *g*, *b*)

> Transforms RGB value in a three-integers form ([0; 255], [0; 255], [0; 255]) to an one-integer form.

```
>>> hex(rgb_to_hex(0, 128, 0))
'0x8000'
```

> > **Parameters**
> > > - **r** (*int*) – value of red channel.
> > > - **g** (*int*) – value of green channel.

- **b** (*int*) – value of blue channel.

    Returns
    RGB value.

    Return type
    int

pytermor.utilmisc.**hsv_to_rgb**(*h, s, v*)

Transforms HSV value in three-floats form (where 0 <= h < 360, 0 <= s <= 1, and 0 <= v <= 1) into RGB three-integer form ([0; 255], [0; 255], [0; 255]).

```
>>> hsv_to_rgb(270, 2/3, 0.75)
(128, 64, 192)
```

Parameters

- **h** (*float*) – hue channel value.

- **s** (*float*) – saturation channel value.

- **v** (*float*) – value channel value.

Returns
R, G, B channel values correspondingly.

Return type
Union[RGB, *Tuple*[int, int, int]]

pytermor.utilmisc.**rgb_to_hsv**(*r, g, b*)

Transforms RGB value in a three-integers form ([0; 255], [0; 255], [0; 255]) to an HSV in three-floats form such as (0 <= h < 360, 0 <= s <= 1, and 0 <= v <= 1).

```
>>> rgb_to_hsv(0, 0, 255)
HSV(hue=240.0, saturation=1.0, value=1.0)
```

Parameters

- **r** (*int*) – value of red channel.

- **g** (*int*) – value of green channel.

- **b** (*int*) – value of blue channel.

Returns
H, S, V channel values correspondingly.

Return type
Tuple[float, float, float]

pytermor.utilmisc.**hex_to_hsv**(*hex_value*)

Transforms `hex_value` in *int* form into named tuple consisting of three floats corresponding to **hue**, **saturation** and **value** channel values respectively. Hue is within [0, 359] range, both saturation and value are within [0; 1] range.

```
>>> hex_to_hsv(0x999999)
HSV(hue=0.0, saturation=0.0, value=0.6)
```

Parameters
hex_value (*int*) – RGB value.

Returns
named tuple with H, S and V channel values

> Return type
> > HSV

pytermor.utilmisc.**hsv_to_hex**(*h*, *s*, *v*)

> Transforms HSV value in three-floats form (where 0 <= h < 360, 0 <= s <= 1, and 0 <= v <= 1) into an one-integer form.

```
>>> hex(hsv_to_hex(90, 0.5, 0.5))
'0x608040'
```

> **Parameters**
>
> - **h** (*float*) – hue channel value.
>
> - **s** (*float*) – saturation channel value.
>
> - **v** (*float*) – value channel value.
>
> **Returns**
> > RGB value.
>
> **Return type**
> > int

pytermor.utilmisc.**lab_to_rgb**(*l_s*, *a_s*, *b_s*)

> @TODO
>
> **Parameters**
>
> - **l_s** (*float*) –
>
> - **a_s** (*float*) –
>
> - **b_s** (*float*) –
>
> **Returns**
>
> **Return type**
> > *Tuple*[int, int, int]

pytermor.utilmisc.**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.utilmisc.**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.utilmisc.**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* (which **blocks** and requires an output tty), or *False* for a device-independent, deterministic and non-blocking *guessing*, which works most of the time, although there could be rare cases when it is not precise enough.

Return type
    int

pytermor.utilmisc.**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* control sequence; after that wait for the response and *parse* 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* 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.utilmisc.**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

pytermor.utilmisc.**total_size**(*o*, *handlers=None*, *verbose=False*)

> Return the approximate memory footprint of an object and all of its contents.
>
> Automatically finds the contents of the following builtin containers and their subclasses: *tuple, list, deque, dict, set* and *frozenset*. To search other containers, add handlers to iterate over their contents:

```
handlers = {ContainerClass: iter, ContainerClass2: ContainerClass2.get_elements}
```

> **Parameters**
>> - **o** (*Any*) –
>> - **handlers** (*Optional[Dict[Any, Iterator]]*) –
>> - **verbose** (*bool*) –
>
> **Return type**
>> int

2.10 pytermor.utilnum

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.utilnum.**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.utilnum.**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.utilnum.**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.

Note: All arguments except fallback are *kwonly*-type arguments.

Parameters

- **fallback** (*StaticFormatter*) – Take missing (i.e., *None*) attribute values from this instance.

- **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`) – [default: *False*]

- **legacy_rounding** (`bool`) – [default: *False*]

- **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** (*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**
> pytermor.text.RT

class pytermor.utilnum.**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: All arguments except `fallback` and `units` are *kwonly*-type arguments.

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

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

> **Return type**
> pytermor.text.RT

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

class pytermor.utilnum.**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*) – 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** (*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

            pytermor.text.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** (`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

            RT | None

class pytermor.utilnum.**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.utilnum.**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.utilnum.DualFormatter* | None

get_by_max_len(*max_len*)

…

> **Return type**
>> *pytermor.utilnum.DualFormatter* | None

get_shortest()

…

> **Return type**
>> *pytermor.utilnum.DualFormatter* | None

get_longest()

…

> **Return type**
>> *pytermor.utilnum.DualFormatter* | None

pytermor.utilnum.**highlight**(*string*)

Todo: @TODO

> **Max output len**
>> *same as input*

> **Parameters**
>> **string** (*str*) – input text

> **Return type**
>> pytermor.text.RT

pytermor.utilnum.**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.utilnum.**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.utilnum.**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 7: 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
    pytermor.text.RT

pytermor.utilnum.**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 8: 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'
>>> 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
    pytermor.text.RT

pytermor.utilnum.**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[10]. 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: 1.57k, where underline indicates 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 KiB (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'

Listing 9: 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 depend-
ing on unit type, *None* to use formatters' setting value [*False* by default].

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

Return type
> pytermor.text.RT

pytermor.utilnum.**format_time**(*val_sec*, *auto_color=None*)

> Invoke dynamic-length general-purpose time formatter, which supports a wide range of output units, includ-
ing seconds, minutes, hours, days, weeks, months, years, milliseconds, microseconds, nanoseconds etc.

Listing 10: 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 col-
orizing depending on unit type, *None* to use formatters' setting value [*False* by default].

Return type
> pytermor.text.RT

pytermor.utilnum.**format_time_ms**(*value_ms*, *auto_color=None*)

> Invoke a variation of `formatter_time` specifically configured to format small time intervals.

Listing 11: 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 col-
orizing depending on unit type, *None* to use formatters' setting value [*False* by default].

---

[10] https://htop.dev/
[1] 250 millibytes is not something you would see every day

> **Returns**

> **Return type**
>> pytermor.text.RT

pytermor.utilnum.**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**
>> pytermor.text.RT

pytermor.utilnum.**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'
>>> 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**
>> pytermor.text.RT

pytermor.utilnum.**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**
> > > pytermor.text.RT

pytermor.utilnum.**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**
> > > pytermor.text.RT

2.11 pytermor.utilstr

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

ESCAPE_SEQ_REGEX	s
SGR_SEQ_REGEX	s
CSI_SEQ_REGEX	sssss
CONTROL_CHARS	s
WHITESPACE_CHARS	s
PRINTABLE_CHARS	s
NON_ASCII_CHARS	s
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(s, width[, fillchar, actual_len])	SGR-formatting-aware implementation of `str.center`.
dump(data[, label, max_len_shift])	
ljust_sgr(s, width[, fillchar, actual_len])	SGR-formatting-aware implementation of `str.ljust`.
measure([level, template])	**param level**
pad(n)	
padv(n)	
rjust_sgr(s, width[, fillchar, actual_len])	SGR-formatting-aware implementation of `str.rjust`.
trace([enabled, level, label])	**param enabled**

Classes

AbstractStringTracer(char_per_line)	
AbstractTracer(char_per_line)	
BytesTracer([char_per_line])	str/bytes as byte hex codes, grouped by 4
CsiStringReplacer([repl])	Find all *CSI* seqs (i.e., starting with ESC [) and replace with given string.
EscSeqStringReplacer([repl])	
IFilter()	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.
NonPrintsOmniVisualizer([override])	Input type: *str*, *bytes*.
NonPrintsStringVisualizer([keep_newlines])	Input type: *str*. Replace every whitespace character with "·", except
NoopFilter()	
OmniDecoder()	
OmniEncoder()	
OmniMapper([override])	Input type: *str*, *bytes*.
OmniSanitizer([repl])	Input type: *str*, *bytes*.
SgrStringReplacer([repl])	Find all *SGR* seqs (e.g., ESC [1;4m) and replace with given string.
StringAligner(align, width, *[, sgr_aware])	
StringLinearizer([repl])	Filter transforms all whitespace sequences in the input string into a single space character, or into a specified string.
StringMapper([override])	a
StringReplacer(pattern, repl)	.
StringTracer([char_per_line])	str as byte hex codes (UTF-8), grouped by characters
StringUcpTracer([char_per_line])	str as Unicode codepoints
TracerExtra(label)	
WhitespaceRemover()	Special case of *StringLinearizer*.

pytermor.utilstr.**ljust_sgr**(*s*, *width*, *fillchar=' '*, *actual_len=None*)

> 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.utilstr.**rjust_sgr**(*s*, *width*, *fillchar=' '*, *actual_len=None*)

> 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.utilstr.**center_sgr**(*s*, *width*, *fillchar=' '*, *actual_len=None*)

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).

Todo: (.) – f-

> **Return type**
>> str

pytermor.utilstr.**ESCAPE_SEQ_REGEX** = **re.compile('\n (?P<escape_char>\\x1b)\n (?P<data>\n (?P<nf_class_seq>\n (?P<nf_interm>[\\x20-\\x2f]+)\n (?P<nf_final>[\\x30-\\x7e])\n)|\n (?P<fp_class_seq>\n (?P<fp_cla, re.VERBOSE)**

> s

pytermor.utilstr.**SGR_SEQ_REGEX** = **re.compile('(\\x1b)(\\[)([0-9;]*)(m)')**

> s

pytermor.utilstr.**CSI_SEQ_REGEX** = **re.compile('(\\x1b)(\\[)(([0-9;:<=>?])*)([@A-Za-z])')**

> *sssss*

pytermor.utilstr.**CONTROL_CHARS** = **[0, 1, 2, 3, 4, 5, 6, 7, 8, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 127]**

> s

pytermor.utilstr.**WHITESPACE_CHARS** = **[9, 10, 11, 12, 13, 32]**

> s

pytermor.utilstr.**PRINTABLE_CHARS** = **[33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126]**

> s

pytermor.utilstr.**NON_ASCII_CHARS** = **[128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255]**

> s

pytermor.utilstr.**IT**

> input-type
>
> alias of TypeVar('IT', str, bytes)

pytermor.utilstr.**OT**

> output-type
>
> alias of TypeVar('OT', str, bytes)

---

pytermor.utilstr.**PTT**

> pattern type
>
> alias of Union[*IT*, Pattern[*IT*]]

pytermor.utilstr.**RPT**

> replacer type
>
> alias of Union[*OT*, Callable[[Match[*OT*]], *OT*]]

pytermor.utilstr.**MPT**

> # map
>
> alias of Dict[int, *IT*]

class pytermor.utilstr.**IFilter**

> 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.
>
> **abstract apply**(*inp*, *extra=None*)
>
> > Apply the filter to input *str* or *bytes*.
> >
> > **Parameters**
> >
> > - **inp** (`pytermor.utilstr.IT`) – input string
> > - **extra** (`Optional[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**
> > pytermor.utilstr.OT

class pytermor.utilstr.**StringAligner**(*align*, *width*, ***, *sgr_aware=True*)

> Bases: *IFilter*[str, str]
>
> ---
>
> **Note:** sgr_aware is *kwonly*-type arg.
>
> ---
>
> **Parameters**
>
> - **align** (`Align`) –
> - **width** (`int`) –
> - **sgr_aware** (`bool`) –
>
> **apply**(*inp*, *extra=None*)
>
> > Apply the filter to input *str* or *bytes*.
> >
> > **Parameters**
> >
> > - **inp** (`str`) – input string
> > - **extra** (`Optional[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**
> > str

class pytermor.utilstr.**AbstractTracer**(*char_per_line*)

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

> **apply**(*inp*, *extra=None*)
>
> > Apply the filter to input *str* or *bytes*.
> >
> > > **Parameters**
> > >
> > > > - **inp** (*pytermor.utilstr.IT*) – input string
> > > >
> > > > - **extra** (*Optional[TracerExtra]*) – additional options
> > >
> > > **Returns**
> > >
> > > > transformed string; the type can match the input type, as well as be different – that depends on filter type.
> > >
> > > **Return type**
> > >
> > > > str

class pytermor.utilstr.**BytesTracer**(*char_per_line=32*)

> Bases: *AbstractTracer*[bytes]

> str/bytes as byte hex codes, grouped by 4

Listing 12: Example output

```
0000   0A 20 32 31 36 20 20 20   E2 94 82 20 20 75 70 6C   |a
0010   20 20 20 20 20 20 20 20   20 20 20 20 20 20 20 20   |a
```

> **apply**(*inp*, *extra=None*)
>
> > Apply the filter to input *str* or *bytes*.
> >
> > > **Parameters**
> > >
> > > > - **inp** (*pytermor.utilstr.IT*) – input string
> > > >
> > > > - **extra** (*Optional[TracerExtra]*) – additional options
> > >
> > > **Returns**
> > >
> > > > transformed string; the type can match the input type, as well as be different – that depends on filter type.
> > >
> > > **Return type**
> > >
> > > > str

class pytermor.utilstr.**AbstractStringTracer**(*char_per_line*)

> Bases: *AbstractTracer*[str]

> **apply**(*inp*, *extra=None*)
>
> > Apply the filter to input *str* or *bytes*.
> >
> > > **Parameters**
> > >
> > > > - **inp** (*pytermor.utilstr.IT*) – input string
> > > >
> > > > - **extra** (*Optional[TracerExtra]*) – additional options
> > >
> > > **Returns**
> > >
> > > > transformed string; the type can match the input type, as well as be different – that depends on filter type.

Return type
>   str

class pytermor.utilstr.**StringTracer**(*char_per_line=16*)

>   Bases: *AbstractStringTracer*

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

Listing 13: Example output

```
0056   45 4D 20 43 50 55      20      4F 56 48 20 4E   45 3E 0A 20   |E|
0072   20 20 20 20 20 20 E29482      20 20 20 20 20    20 20 20 20   |␣|
0088   20 20 20 20 37 20      2B      30 20 20 20 20 CE94 20 32 68   |␣|
0104   20 33 33 6D 20 20      20 EFAA8F 20 2D 35 20 C2B0 43 20 20   |␣|
```

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

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

>   **Parameters**
>
>   - **inp** (*pytermor.utilstr.IT*) – input string
>
>   - **extra** (*Optional[TracerExtra]*) – additional options
>
>   **Returns**
>>   transformed string; the type can match the input type, as well as be different – that depends on filter type.
>
>   **Return type**
>>   str

class pytermor.utilstr.**StringUcpTracer**(*char_per_line=16*)

>   Bases: *AbstractStringTracer*

>   str as Unicode codepoints

Todo: venv/lib/python3.8/site-packages/pygments/lexers/hexdump.py

---

Listing 14: Example output

```
 56 |U+ 45 4d 20 43 50 55      20      4f 56 48 20 4e   45 3e 0a 20 | EM␣CPU␣OVH␣NE>␣
 72 |U+ 20 20 20 20 20 20 2502      20 20 20 20 20    20 20 20 20 | ␣␣␣␣␣␣|␣␣␣␣␣␣␣␣
 88 |U+ 20 20 20 20 37 20      2b      30 20 20 20 20 394 20 32 68 | ␣␣␣␣7␣+0␣␣␣␣␣2h
104 |U+ 20 33 33 6d 20 20      20 fa8f 20 2d 35 20   b0 43 20 20 | ␣33m␣␣␣␣-5␣°C␣␣
```

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

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

>   **Parameters**
>
>   - **inp** (*pytermor.utilstr.IT*) – input string
>
>   - **extra** (*Optional[TracerExtra]*) – additional options
>
>   **Returns**
>>   transformed string; the type can match the input type, as well as be different – that depends on filter type.

>> **Return type**
>> str

class pytermor.utilstr.**TracerExtra**(*label: 'str'*)

class pytermor.utilstr.**StringReplacer**(*pattern*, *repl*)

> Bases: [*IFilter*](#)[str, str]

> .

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

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

>> **Parameters**

>>> • **inp** (*str*) – input string

>>> • **extra** (*Optional[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**

>>> str

class pytermor.utilstr.**SgrStringReplacer**(*repl=''*)

> Bases: [*StringReplacer*](#)

> Find all [*SGR*](#) 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** (*str*) – input string

>>> • **extra** (*Optional[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**

>>> str

class pytermor.utilstr.**CsiStringReplacer**(*repl=''*)

> Bases: [*StringReplacer*](#)

> Find all [*CSI*](#) 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** (*str*) – input string

- **extra** (*Optional[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**
> > str

class pytermor.utilstr.**StringLinearizer**(*repl=' '*)

> 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** (*str*) – input string
> > > - **extra** (*Optional[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**
> > > > str

class pytermor.utilstr.**WhitespaceRemover**

> Bases: *StringLinearizer*
>
> 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** (*str*) – input string
> > > - **extra** (*Optional[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**
> > > > str

class pytermor.utilstr.**OmniMapper**(*override=None*)

> 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|None]; 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*, or None.

> **See**
> *NonPrintsOmniVisualizer*

> **apply**(*inp*, *extra=None*)
> Apply the filter to input *str* or *bytes*.

> > **Parameters**
> > - **inp** (`pytermor.utilstr.IT`) – input string
> > - **extra** (`Optional[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**
> > pytermor.utilstr.IT

class pytermor.utilstr.**StringMapper**(*override=None*)

> Bases: *OmniMapper*[str]

> a

> **apply**(*inp*, *extra=None*)
> Apply the filter to input *str* or *bytes*.

> > **Parameters**
> > - **inp** (`str`) – input string
> > - **extra** (`Optional[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**
> > str

class pytermor.utilstr.**NonPrintsOmniVisualizer**(*override=None*)

> Bases: *OmniMapper*

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

> **apply**(*inp*, *extra=None*)
> Apply the filter to input *str* or *bytes*.

> > **Parameters**
> > - **inp** (`pytermor.utilstr.IT`) – input string
> > - **extra** (`Optional[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**
> pytermor.utilstr.IT

class pytermor.utilstr.**NonPrintsStringVisualizer**(*keep_newlines=True*)

> 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'
>>> apply_filters('1. D'+os.linesep+'2. L ', NonPrintsStringVisualizer())
'1.␣D
```

Block quote ends without a blank line; unexpected unindent.

2.␣L␣'

> > **param keep_newlines**
> > 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** (*str*) – input string
> >
> > - **extra** (*Optional[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**
> > str

class pytermor.utilstr.**OmniSanitizer**(*repl=b'.'*)

> 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** (*pytermor.utilstr.IT*) – input string
> >
> > - **extra** (*Optional[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**
>> pytermor.utilstr.IT

pytermor.utilstr.**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'

>>> 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** (*pytermor.utilstr.IT*) – String/bytes to filter.
>> - **args** (*Union[IFilter, Type[IFilter]]*) – Instance(s) implementing *IFilter* or their type(s).
>
> **Return type**
>> pytermor.utilstr.OT

pytermor.utilstr.**dump**(*data*, *label=None*, *max_len_shift=None*)

Todo:

- format selection
- special handling of one-line input
- squash repeating lines

> **Return type**
>> str | None

pytermor.utilstr.**trace**(*enabled=True*, *level=5*, *label='Dump'*)

> **Parameters**
>> - **enabled** (*bool*) –
>> - **level** (*int*) –
>> - **label** (*str*) –
>
> **Returns**

pytermor.utilstr.**measure**(*level=10*, *template='Done in %s'*)

> **Parameters**
>> - **level** (*int*) –
>> - **template** (*str*) –
>
> **Returns**

3

Changelog

3.1 Releases

This project uses Semantic Versioning – https://semver.org *(starting from v2.0)*

3.1.1 pending

- [DOCS] updated changelog
- [FIX] debug logging
- [FIX] bug in `SimpleTable` renderer when row is wider than a terminal
- [NEW] `lab_to_rgb()`
- [FIX] `measure_char_width` and `get_char_width` internal logic
- [UPDATE] README
- [FIX] `StaticFormatter` padding
- [NEW] `Hightlighter` from static methods to real class
- [REFACTOR] edited highlighter styles
- [NEW] `format_time`, `format_time_ms`, `format_time_ns`
- [REFACTOR] naming:
    - `CustomBaseUnit` -> `DualBaseUnit`
    - `DynamicBaseFormatter` -> `DynamicFormatter`
    - `StaticBaseFormatter` -> `StaticFormatter`
- [NEW] numeric formatters fallback mechanics
- [TESTS] numeric formatters colorizing
- [REFACTOR] `utilnum._TDF_REGISTRY` -> `TDF_REGISTRY`
- [REFACTOR] `TDF_REGISTRY` -> `dual_registry`- ``FORMATTER_` constants from top-level imports
- [FIX] `AbstractTracer` failure on empty input
- `Dockerfile` for repeatable builds
- [FIX] pipelines
- **|DRAFT|** ci debug

- coverage git ignore
- `cli-docker` make command
- [DOCS] small fixes

3.1.2 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*

3.1.3 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

3.1.4 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*

3.1.5 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.

3.1.6 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.

3.1.7 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.

3.1.8 v2.2-dev

Oct 22

- [NEW] *TmuxRenderer*
- [NEW] *wait_key()* input helper.
- [NEW] Color config.
- [NEW] IRenderable` interface.
- [NEW] Named colors list.

3.1.9 v2.1-dev

Aug 22

- [NEW] Color presets.
- [TESTS] More unit tests for formatters.

3.1.10 v2.0-dev

Jul 22

- [REWORK] Complete library rewrite.
- [DOCS] sphinx and readthedocs integraton.
- [NEW] High-level abstractions Color, *Renderer* and *Style*.
- [TESTS] pytest and coverage integration.
- [TESTS] Unit tests for formatters and new modules.

3.1.11 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.

3.1.12 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.

3.1.13 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.

3.1.14 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.

3.1.15 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.

3.1.16 v1.3

- [NEW] Added `span.GRAY` and `span.BG_GRAY` format presets.
- [REFACTOR] Interface revisioning.

3.1.17 v1.2

- [NEW] `EmptySequenceSGR` and `EmptyFormat` classes.
- [NEW] `opening_seq` and `closing_seq` properties for `Span` class.

3.1.18 v1.1

Apr 22

- [NEW] Autoformat feature.

3.1.19 v1.0

- First public version.

3.1.20 v0.90

Mar 22

- First commit.

4

License

```
                    GNU LESSER GENERAL PUBLIC 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.

  This version of the GNU Lesser General Public License incorporates 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.

  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 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 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.
```

Python Module Index

p

Index

L

M

N

O

P

R