pytermor

Release 2.36.0-dev1

Alexandr Shavykin

Jan 19, 2023

CONTENTS

(yet another) Python library designed for formatting terminal output using ANSI escape codes. Implements automatic *"soft"* format termination. Provides a registry of low-level SGR (Select Graphic Rendition) `sequences` and formatting `spans` (or combined sequences). Also includes a set of formatters for pretty output.

Key feature of this library is providing necessary abstractions for building complex text sections with lots of formatting, while keeping the application code clear and readable.

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

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

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

CHAPTER

ONE

GUIDE

1.1 Getting started

1.1.1 Installation

```
pip install pytermor
```

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:

```
from pytermor import Spans

print(Spans.RED('Feat') + Spans.BOLD('ures'))
```

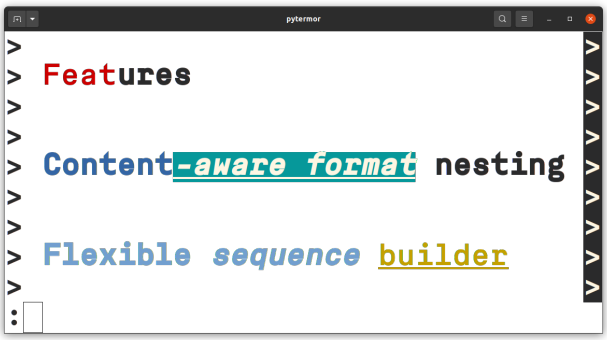
Content-aware format nesting

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).

```
from pytermor import Span

span1 = Span('blue', 'bold')
span2 = Span('cyan', 'inversed', 'underlined', 'italic')

msg = span1(f'Content{span2("-aware format")} nesting')
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.

```
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 `presets`);
- True Color RGB mode (16M colors).

```
from pytermor import SequenceSGR, SeqIndex

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

Customizable output formats

Todo: @TODOTODO

String and number formatters

Todo: @TODOTODO

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.

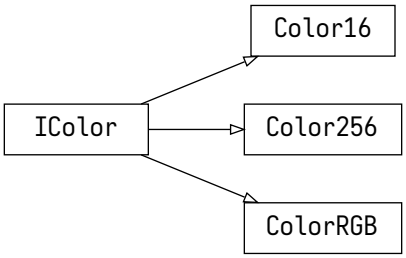


Fig. 1: `IColor` inheritance tree

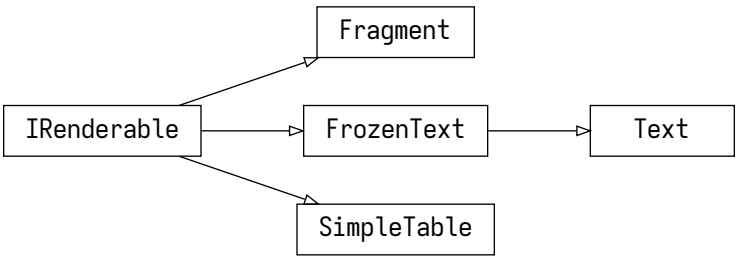


Fig. 2: *IRenderable* inheritance tree

1.2.2 Colors

1.2.3 Styles

1.2.4 Output format control

1.2.5 Color mode fallbacks

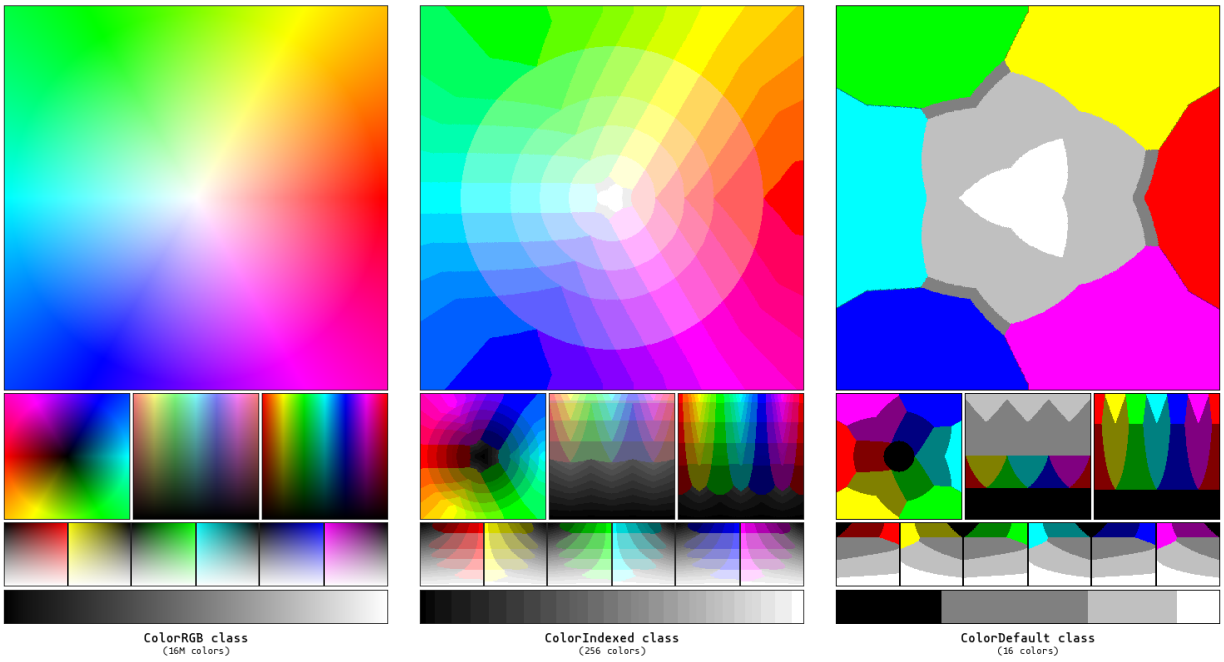


Fig. 3: Color approximations for indexed modes

1.3 Renderers

Todo: Win32Renderer ?

1.4 String filters

1.5 Number formatters

Todo: The library contains @TODO

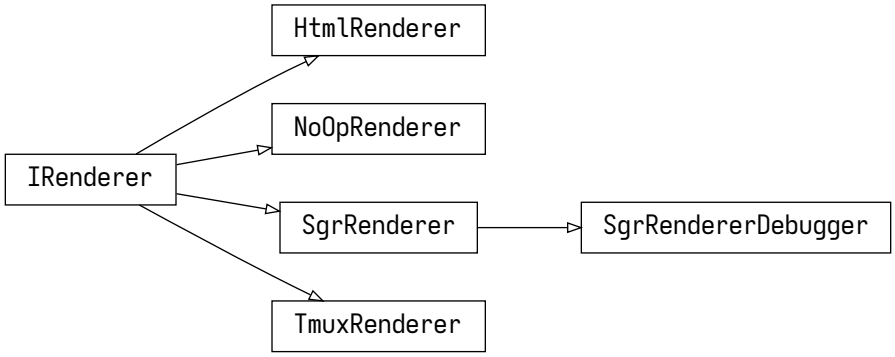


Fig. 4: *IRenderer* inheritance tree

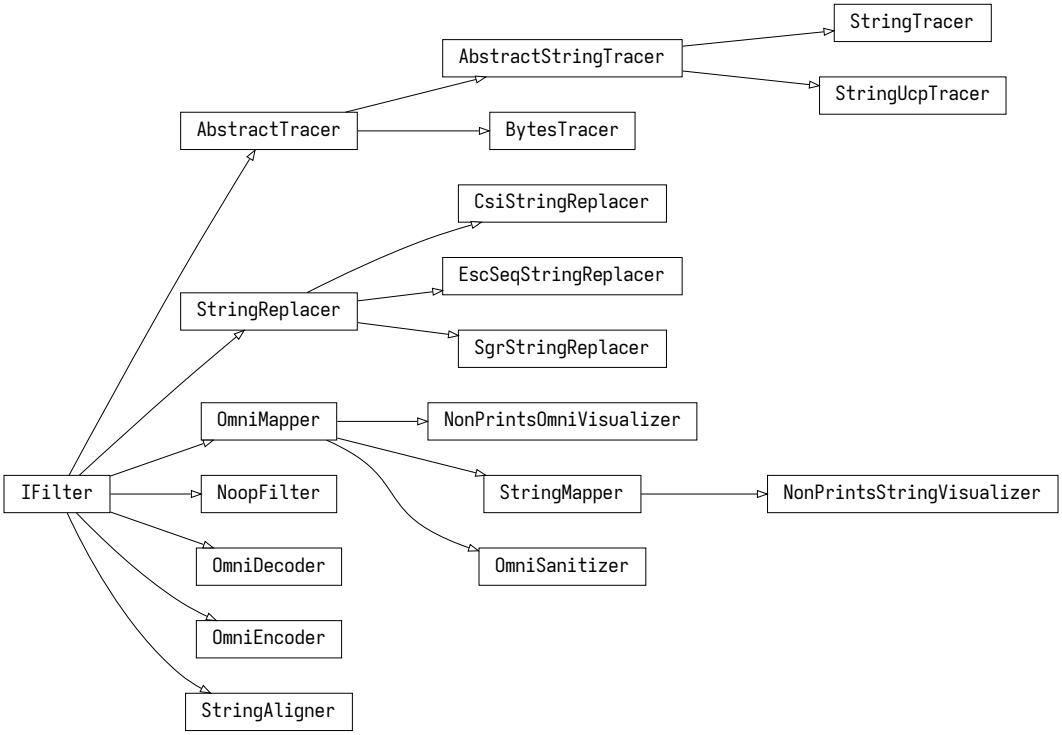


Fig. 5: *IFilter* inheritance tree

1.5.1 Auto-float formatter

1.5.2 Prefixed-unit formatter

1.5.3 Time delta formatter

```
import pytermor.utilnum
from pytermor import RendererManager, SgrRenderer
from pytermor.util import time_delta

seconds_list = [2, 10, 60, 2700, 32340, 273600, 4752000, 864000000]
max_len_list = [3, 6, 10]

for max_len in max_len_list:
    formatter = pytermor.utilnum.tdf_registry.find_matching(max_len)

RendererManager.set_default(SgrRenderer)
for seconds in seconds_list:
    for max_len in max_len_list:
        formatter = pytermor.utilnum.tdf_registry.get_by_max_len(max_len)
        print(formatter.format(seconds), end=' ')
    print()
```

1.6 es7s color collection

1.6.1 lisr

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::.

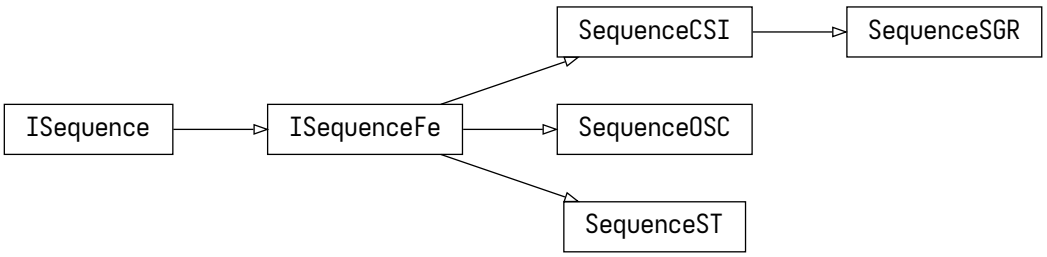


Fig. 6: *ISequence* inheritance tree

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:

```
from pytermor import Span, Spans, SeqIndex

# implicitly:
span_warn = Span(93, 4)
# or explicitly:
span_warn = Span.init_explicit(
    SeqIndex.HI_YELLOW + SeqIndex.UNDERLINED,  # sequences can be summed up, remember?
    SeqIndex.COLOR_OFF + SeqIndex.UNDERLINED_OFF,  # "counteractive" sequences
    hard_reset_after=False
)

orig_text = Spans.BOLD(f'this is {SeqIndex.BG_GRAY}the original{SeqIndex.RESET} string')
updated_text = orig_text.replace('original', span_warn('updated'), 1)
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 `presets`);
- integer param value;
- existing *SequenceSGR* instance (params will be extracted).

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

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 `presets` 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:

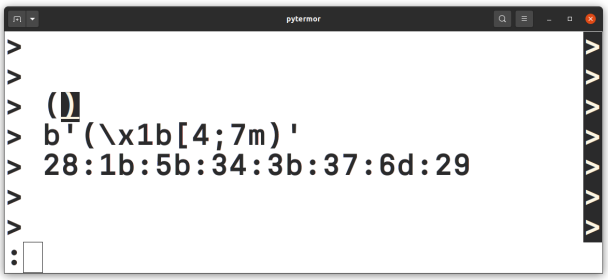
- *make_color_256()* will produce sequence operating in 256-colors mode (for a complete list see `presets`);
- *make_color_rgb()* will create a sequence capable of setting the colors in True Color 16M mode (however, some terminal emulators doesn't support it).

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

```
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())
```

1.7.7 Sources

1. XTerm Control Sequences
2. ECMA-48 specification

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 (not including `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)
- SEQ (sequence module)
- SPN (span module)
- CLR (color module)
- STY (style module)

1.8.1 Meta, attributes, resetters

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

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

1.8.2 Color16 presets

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
Foreground *default* colors								
	BLACK	30	V	V	V		#000000	Black
	RED	31	V	V	V		#800000	Maroon
	GREEN	32	V	V	V		#008000	Green
	YELLOW	33	V	V	V		#808000	Olive
	BLUE	34	V	V	V		#000080	Navy
	MAGENTA	35	V	V	V		#800080	Purple
	CYAN	36	V	V	V		#008080	Teal
	WHITE	37	V	V	V		#c0c0c0	Silver
Background *default* colors								
	BG_BLACK	40	V	V	V		#000000	Black
	BG_RED	41	V	V	V		#800000	Maroon
	BG_GREEN	42	V	V	V		#008000	Green
	BG_YELLOW	43	V	V	V		#808000	Olive
	BG_BLUE	44	V	V	V		#000080	Navy
	BG_MAGENTA	45	V	V	V		#800080	Purple
	BG_CYAN	46	V	V	V		#008080	Teal
	BG_WHITE	47	V	V	V		#c0c0c0	Silver
High-intensity foreground *default* colors								
	GRAY	90	V	V	V		#808080	Grey
	HI_RED	91	V	V	V		#ff0000	Red
	HI_GREEN	92	V	V	V		#00ff00	Lime
	HI_YELLOW	93	V	V	V		#ffff00	Yellow
	HI_BLUE	94	V	V	V		#0000ff	Blue
	HI_MAGENTA	95	V	V	V		#ff00ff	Fuchsia
	HI_CYAN	96	V	V	V		#00ffff	Aqua
	HI_WHITE	97	V	V	V		#ffffff	White
High-intensity background *default* colors								
	BG_GRAY	100	V	V	V		#808080	Grey
	BG_HI_RED	101	V	V	V		#ff0000	Red
	BG_HI_GREEN	102	V	V	V		#00ff00	Lime
	BG_HI_YELLOW	103	V	V	V		#ffff00	Yellow
	BG_HI_BLUE	104	V	V	V		#0000ff	Blue
	BG_HI_MAGENTA	105	V	V	V		#ff00ff	Fuchsia
	BG_HI_CYAN	106	V	V	V		#00ffff	Aqua
	BG_HI_WHITE	107	V	V	V		#ffffff	White

1.8.3 Color256 presets

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
	XTERM_BLACK[3]	0			V		#000000	
	XTERM_MAROON	1			V		#800000	
	XTERM_GREEN	2			V		#008000	
	XTERM_OLIVE	3			V		#808000	
	XTERM_NAVY	4			V		#000080	
	XTERM_PURPLE_5	5			V		#800080	Purple[4]
	XTERM_TEAL	6			V		#008080	
	XTERM_SILVER	7			V		#c0c0c0	
	XTERM_GREY	8			V		#808080	
	XTERM_RED	9			V		#ff0000	
	XTERM_LIME	10			V		#00ff00	
	XTERM_YELLOW	11			V		#ffff00	
	XTERM_BLUE	12			V		#0000ff	
	XTERM_FUCHSIA	13			V		#ff00ff	
	XTERM_AQUA	14			V		#00ffff	
	XTERM_WHITE	15			V		#ffffff	
	XTERM_GREY_0	16			V		#000000	
	XTERM_NAVY_BLUE	17			V		#00005f	
	XTERM_DARK_BLUE	18			V		#000087	
	XTERM_BLUE_3	19			V		#0000af	
	XTERM_BLUE_2	20			V		#0000d7	Blue3
	XTERM_BLUE_1	21			V		#0000ff	
	XTERM_DARK_GREEN	22			V		#005f00	
	XTERM_DEEP_SKY_BLUE_7	23			V		#005f5f	DeepSkyBlue4
	XTERM_DEEP_SKY_BLUE_6	24			V		#005f87	DeepSkyBlue4
	XTERM_DEEP_SKY_BLUE_5	25			V		#005faf	DeepSkyBlue4
	XTERM_DODGER_BLUE_3	26			V		#005fd7	
	XTERM_DODGER_BLUE_2	27			V		#005fff	
	XTERM_GREEN_5	28			V		#008700	Green4
	XTERM_SPRING_GREEN_4	29			V		#00875f	
	XTERM_TURQUOISE_4	30			V		#008787	
	XTERM_DEEP_SKY_BLUE_4	31			V		#0087af	DeepSkyBlue3
	XTERM_DEEP_SKY_BLUE_3	32			V		#0087d7	
	XTERM_DODGER_BLUE_1	33			V		#0087ff	
	XTERM_GREEN_4	34			V		#00af00	Green3
	XTERM_SPRING_GREEN_5	35			V		#00af5f	SpringGreen3
	XTERM_DARK_CYAN	36			V		#00af87	
	XTERM_LIGHT_SEA_GREEN	37			V		#00afaf	
	XTERM_DEEP_SKY_BLUE_2	38			V		#00afd7	
	XTERM_DEEP_SKY_BLUE_1	39			V		#00afff	
	XTERM_GREEN_3	40			V		#00d700	
	XTERM_SPRING_GREEN_3	41			V		#00d75f	
	XTERM_SPRING_GREEN_6	42			V		#00d787	SpringGreen2
	XTERM_CYAN_3	43			V		#00d7af	
	XTERM_DARK_TURQUOISE	44			V		#00d7d7	
	XTERM_TURQUOISE_2	45			V		#00d7ff	
	XTERM_GREEN_2	46			V		#00ff00	Green1

continues on next page

Table 2 – continued from previous page

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
	XTERM_SPRING_GREEN_2	47			V		#00ff5f	
	XTERM_SPRING_GREEN_1	48			V		#00ff87	
	XTERM_MEDIUM_SPRING_GREEN	49			V		#00ffaf	
	XTERM_CYAN_2	50			V		#00ffd7	
	XTERM_CYAN_1	51			V		#00ffff	
	XTERM_DARK_RED_2	52			V		#5f0000	DarkRed
	XTERM_DEEP_PINK_8	53			V		#5f005f	DeepPink4
	XTERM_PURPLE_6	54			V		#5f0087	Purple4
	XTERM_PURPLE_4	55			V		#5f00af	
	XTERM_PURPLE_3	56			V		#5f00d7	
	XTERM_BLUE_VIOLET	57			V		#5f00ff	
	XTERM_ORANGE_4	58			V		#5f5f00	
	XTERM_GREY_37	59			V		#5f5f5f	
	XTERM_MEDIUM_PURPLE_7	60			V		#5f5f87	MediumPurple4
	XTERM_SLATE_BLUE_3	61			V		#5f5faf	
	XTERM_SLATE_BLUE_2	62			V		#5f5fd7	SlateBlue3
	XTERM_ROYAL_BLUE_1	63			V		#5f5fff	
	XTERM_CHARTREUSE_6	64			V		#5f8700	Chartreuse4
	XTERM_DARK_SEA_GREEN_9	65			V		#5f875f	DarkSeaGreen4
	XTERM_PALE_TURQUOISE_4	66			V		#5f8787	
	XTERM_STEEL_BLUE	67			V		#5f87af	
	XTERM_STEEL_BLUE_3	68			V		#5f87d7	
	XTERM_CORNFLOWER_BLUE	69			V		#5f87ff	
	XTERM_CHARTREUSE_5	70			V		#5faf00	Chartreuse3
	XTERM_DARK_SEA_GREEN_8	71			V		#5faf5f	DarkSeaGreen4
	XTERM_CADET_BLUE_2	72			V		#5faf87	CadetBlue
	XTERM_CADET_BLUE	73			V		#5fafaf	
	XTERM_SKY_BLUE_3	74			V		#5fafd7	
	XTERM_STEEL_BLUE_2	75			V		#5fafff	SteelBlue1
	XTERM_CHARTREUSE_4	76			V		#5fd700	Chartreuse3
	XTERM_PALE_GREEN_4	77			V		#5fd75f	PaleGreen3
	XTERM_SEA_GREEN_3	78			V		#5fd787	
	XTERM_AQUAMARINE_3	79			V		#5fd7af	
	XTERM_MEDIUM_TURQUOISE	80			V		#5fd7d7	
	XTERM_STEEL_BLUE_1	81			V		#5fd7ff	
	XTERM_CHARTREUSE_2	82			V		#5fff00	
	XTERM_SEA_GREEN_4	83			V		#5fff5f	SeaGreen2
	XTERM_SEA_GREEN_2	84			V		#5fff87	SeaGreen1
	XTERM_SEA_GREEN_1	85			V		#5fffaf	
	XTERM_AQUAMARINE_2	86			V		#5fffd7	Aquamarine1
	XTERM_DARK_SLATE_GRAY_2	87			V		#5fffff	
	XTERM_DARK_RED	88			V		#870000	
	XTERM_DEEP_PINK_7	89			V		#87005f	DeepPink4
	XTERM_DARK_MAGENTA_2	90			V		#870087	DarkMagenta
	XTERM_DARK_MAGENTA	91			V		#8700af	
	XTERM_DARK_VIOLET_2	92			V		#8700d7	DarkViolet
	XTERM_PURPLE_2	93			V		#8700ff	Purple
	XTERM_ORANGE_3	94			V		#875f00	Orange4
	XTERM_LIGHT_PINK_3	95			V		#875f5f	LightPink4

continues on next page

Table 2 – continued from previous page

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
	XTERM_PLUM_4	96			V		#875f87	
	XTERM_MEDIUM_PURPLE_6	97			V		#875faf	MediumPurple3
	XTERM_MEDIUM_PURPLE_5	98			V		#875fd7	MediumPurple3
	XTERM_SLATE_BLUE_1	99			V		#875fff	
	XTERM_YELLOW_6	100			V		#878700	Yellow4
	XTERM_WHEAT_4	101			V		#87875f	
	XTERM_GREY_53	102			V		#878787	
	XTERM_LIGHT_SLATE_GREY	103			V		#8787af	
	XTERM_MEDIUM_PURPLE_4	104			V		#8787d7	MediumPurple
	XTERM_LIGHT_SLATE_BLUE	105			V		#8787ff	
	XTERM_YELLOW_4	106			V		#87af00	
	XTERM_DARK_OLIVE_GREEN_6	107			V		#87af5f	DarkOliveGreen3
	XTERM_DARK_SEA_GREEN_7	108			V		#87af87	DarkSeaGreen
	XTERM_LIGHT_SKY_BLUE_3	109			V		#87afaf	
	XTERM_LIGHT_SKY_BLUE_2	110			V		#87afd7	LightSkyBlue3
	XTERM_SKY_BLUE_2	111			V		#87afff	
	XTERM_CHARTREUSE_3	112			V		#87d700	Chartreuse2
	XTERM_DARK_OLIVE_GREEN_4	113			V		#87d75f	DarkOliveGreen3
	XTERM_PALE_GREEN_3	114			V		#87d787	
	XTERM_DARK_SEA_GREEN_5	115			V		#87d7af	DarkSeaGreen3
	XTERM_DARK_SLATE_GRAY_3	116			V		#87d7d7	
	XTERM_SKY_BLUE_1	117			V		#87d7ff	
	XTERM_CHARTREUSE_1	118			V		#87ff00	
	XTERM_LIGHT_GREEN_2	119			V		#87ff5f	LightGreen
	XTERM_LIGHT_GREEN	120			V		#87ff87	
	XTERM_PALE_GREEN_1	121			V		#87ffaf	
	XTERM_AQUAMARINE_1	122			V		#87ffd7	
	XTERM_DARK_SLATE_GRAY_1	123			V		#87ffff	
	XTERM_RED_4	124			V		#af0000	Red3
	XTERM_DEEP_PINK_6	125			V		#af005f	DeepPink4
	XTERM_MEDIUM_VIOLET_RED	126			V		#af0087	
	XTERM_MAGENTA_6	127			V		#af00af	Magenta3
	XTERM_DARK_VIOLET	128			V		#af00d7	
	XTERM_PURPLE	129			V		#af00ff	
	XTERM_DARK_ORANGE_3	130			V		#af5f00	
	XTERM_INDIAN_RED_4	131			V		#af5f5f	IndianRed
	XTERM_HOT_PINK_5	132			V		#af5f87	HotPink3
	XTERM_MEDIUM_ORCHID_4	133			V		#af5faf	MediumOrchid3
	XTERM_MEDIUM_ORCHID_3	134			V		#af5fd7	MediumOrchid
	XTERM_MEDIUM_PURPLE_2	135			V		#af5fff	
	XTERM_DARK_GOLDENROD	136			V		#af8700	
	XTERM_LIGHT_SALMON_3	137			V		#af875f	
	XTERM_ROSY_BROWN	138			V		#af8787	
	XTERM_GREY_63	139			V		#af87af	
	XTERM_MEDIUM_PURPLE_3	140			V		#af87d7	MediumPurple2
	XTERM_MEDIUM_PURPLE_1	141			V		#af87ff	
	XTERM_GOLD_3	142			V		#afaf00	
	XTERM_DARK_KHAKI	143			V		#afaf5f	
	XTERM_NAVAJO_WHITE_3	144			V		#afaf87	

continues on next page

Table 2 – continued from previous page

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
	XTERM_GREY_69	145			V		#afafaf	
	XTERM_LIGHT_STEEL_BLUE_3	146			V		#afafd7	
	XTERM_LIGHT_STEEL_BLUE_2	147			V		#afafff	LightSteelBlue
	XTERM_YELLOW_5	148			V		#afd700	Yellow3
	XTERM_DARK_OLIVE_GREEN_5	149			V		#afd75f	DarkOliveGreen3
	XTERM_DARK_SEA_GREEN_6	150			V		#afd787	DarkSeaGreen3
	XTERM_DARK_SEA_GREEN_4	151			V		#afd7af	DarkSeaGreen2
	XTERM_LIGHT_CYAN_3	152			V		#afd7d7	
	XTERM_LIGHT_SKY_BLUE_1	153			V		#afd7ff	
	XTERM_GREEN_YELLOW	154			V		#afff00	
	XTERM_DARK_OLIVE_GREEN_3	155			V		#afff5f	DarkOliveGreen2
	XTERM_PALE_GREEN_2	156			V		#afff87	PaleGreen1
	XTERM_DARK_SEA_GREEN_3	157			V		#afffaf	DarkSeaGreen2
	XTERM_DARK_SEA_GREEN_1	158			V		#afffd7	
	XTERM_PALE_TURQUOISE_1	159			V		#afffff	
	XTERM_RED_3	160			V		#d70000	
	XTERM_DEEP_PINK_5	161			V		#d7005f	DeepPink3
	XTERM_DEEP_PINK_3	162			V		#d70087	
	XTERM_MAGENTA_3	163			V		#d700af	
	XTERM_MAGENTA_5	164			V		#d700d7	Magenta3
	XTERM_MAGENTA_4	165			V		#d700ff	Magenta2
	XTERM_DARK_ORANGE_2	166			V		#d75f00	DarkOrange3
	XTERM_INDIAN_RED_3	167			V		#d75f5f	IndianRed
	XTERM_HOT_PINK_4	168			V		#d75f87	HotPink3
	XTERM_HOT_PINK_3	169			V		#d75faf	HotPink2
	XTERM_ORCHID_3	170			V		#d75fd7	Orchid
	XTERM_MEDIUM_ORCHID_2	171			V		#d75fff	MediumOrchid1
	XTERM_ORANGE_2	172			V		#d78700	Orange3
	XTERM_LIGHT_SALMON_2	173			V		#d7875f	LightSalmon3
	XTERM_LIGHT_PINK_2	174			V		#d78787	LightPink3
	XTERM_PINK_3	175			V		#d787af	
	XTERM_PLUM_3	176			V		#d787d7	
	XTERM_VIOLET	177			V		#d787ff	
	XTERM_GOLD_2	178			V		#d7af00	Gold3
	XTERM_LIGHT_GOLDENROD_5	179			V		#d7af5f	LightGoldenrod3
	XTERM_TAN	180			V		#d7af87	
	XTERM_MISTY_ROSE_3	181			V		#d7afaf	
	XTERM_THISTLE_3	182			V		#d7afd7	
	XTERM_PLUM_2	183			V		#d7afff	
	XTERM_YELLOW_3	184			V		#d7d700	
	XTERM_KHAKI_3	185			V		#d7d75f	
	XTERM_LIGHT_GOLDENROD_3	186			V		#d7d787	LightGoldenrod2
	XTERM_LIGHT_YELLOW_3	187			V		#d7d7af	
	XTERM_GREY_84	188			V		#d7d7d7	
	XTERM_LIGHT_STEEL_BLUE_1	189			V		#d7d7ff	
	XTERM_YELLOW_2	190			V		#d7ff00	
	XTERM_DARK_OLIVE_GREEN_2	191			V		#d7ff5f	DarkOliveGreen1
	XTERM_DARK_OLIVE_GREEN_1	192			V		#d7ff87	
	XTERM_DARK_SEA_GREEN_2	193			V		#d7ffaf	DarkSeaGreen1

continues on next page

Table 2 – continued from previous page

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
	XTERM_HONEYDEW_2	194			V		#d7ffd7	
	XTERM_LIGHT_CYAN_1	195			V		#d7ffff	
	XTERM_RED_1	196			V		#ff0000	
	XTERM_DEEP_PINK_4	197			V		#ff005f	**DeepPink2**
	XTERM_DEEP_PINK_2	198			V		#ff0087	**DeepPink1**
	XTERM_DEEP_PINK_1	199			V		#ff00af	
	XTERM_MAGENTA_2	200			V		#ff00d7	
	XTERM_MAGENTA_1	201			V		#ff00ff	
	XTERM_ORANGE_RED_1	202			V		#ff5f00	
	XTERM_INDIAN_RED_1	203			V		#ff5f5f	
	XTERM_INDIAN_RED_2	204			V		#ff5f87	**IndianRed1**
	XTERM_HOT_PINK_2	205			V		#ff5faf	**HotPink**
	XTERM_HOT_PINK	206			V		#ff5fd7	
	XTERM_MEDIUM_ORCHID_1	207			V		#ff5fff	
	XTERM_DARK_ORANGE	208			V		#ff8700	
	XTERM_SALMON_1	209			V		#ff875f	
	XTERM_LIGHT_CORAL	210			V		#ff8787	
	XTERM_PALE_VIOLET_RED_1	211			V		#ff87af	
	XTERM_ORCHID_2	212			V		#ff87d7	
	XTERM_ORCHID_1	213			V		#ff87ff	
	XTERM_ORANGE_1	214			V		#ffaf00	
	XTERM_SANDY_BROWN	215			V		#ffaf5f	
	XTERM_LIGHT_SALMON_1	216			V		#ffaf87	
	XTERM_LIGHT_PINK_1	217			V		#ffafaf	
	XTERM_PINK_1	218			V		#ffafd7	
	XTERM_PLUM_1	219			V		#ffafff	
	XTERM_GOLD_1	220			V		#ffd700	
	XTERM_LIGHT_GOLDENROD_4	221			V		#ffd75f	**LightGoldenrod2**
	XTERM_LIGHT_GOLDENROD_2	222			V		#ffd787	
	XTERM_NAVAJO_WHITE_1	223			V		#ffd7af	
	XTERM_MISTY_ROSE_1	224			V		#ffd7d7	
	XTERM_THISTLE_1	225			V		#ffd7ff	
	XTERM_YELLOW_1	226			V		#ffff00	
	XTERM_LIGHT_GOLDENROD_1	227			V		#ffff5f	
	XTERM_KHAKI_1	228			V		#ffff87	
	XTERM_WHEAT_1	229			V		#ffffaf	
	XTERM_CORNSILK_1	230			V		#ffffd7	
	XTERM_GREY_100	231			V		#ffffff	
	XTERM_GREY_3	232			V		#080808	
	XTERM_GREY_7	233			V		#121212	
	XTERM_GREY_11	234			V		#1c1c1c	
	XTERM_GREY_15	235			V		#262626	
	XTERM_GREY_19	236			V		#303030	
	XTERM_GREY_23	237			V		#3a3a3a	
	XTERM_GREY_27	238			V		#444444	
	XTERM_GREY_30	239			V		#4e4e4e	
	XTERM_GREY_35	240			V		#585858	
	XTERM_GREY_39	241			V		#626262	
	XTERM_GREY_42	242			V		#6c6c6c	

continues on next page

Table 2 – continued from previous page

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
	XTERM_GREY_46	243			V		#767676	
	XTERM_GREY_50	244			V		#808080	
	XTERM_GREY_54	245			V		#8a8a8a	
	XTERM_GREY_58	246			V		#949494	
	XTERM_GREY_62	247			V		#9e9e9e	
	XTERM_GREY_66	248			V		#a8a8a8	
	XTERM_GREY_70	249			V		#b2b2b2	
	XTERM_GREY_74	250			V		#bcbcbc	
	XTERM_GREY_78	251			V		#c6c6c6	
	XTERM_GREY_82	252			V		#d0d0d0	
	XTERM_GREY_85	253			V		#dadada	
	XTERM_GREY_89	254			V		#e4e4e4	
	XTERM_GREY_93	255			V		#eeeeee	

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 …

[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.

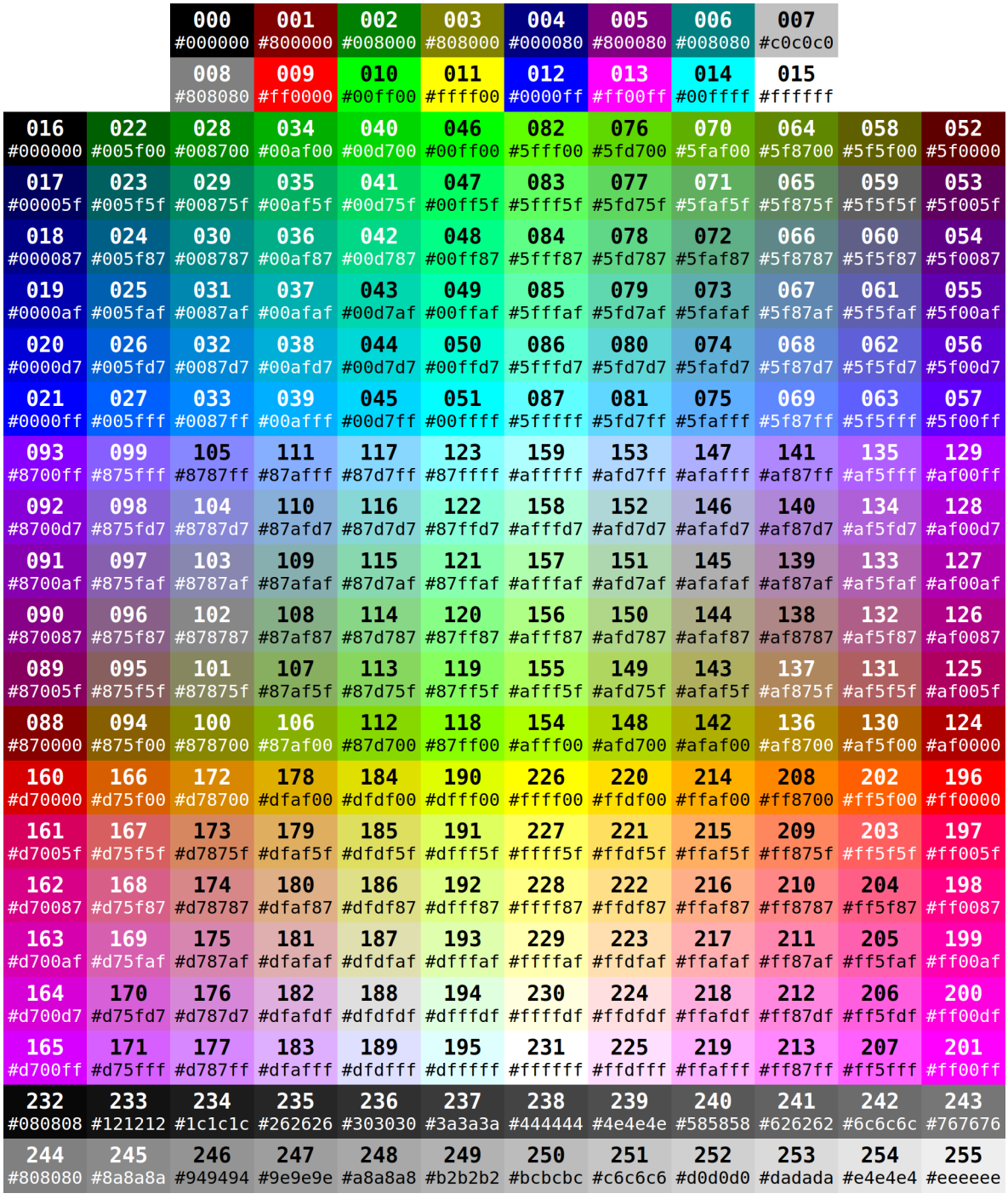


Fig. 7: *Indexed* mode palette

Sources

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

1.10 Configuration

See also:

Config – class containing configuration variables.

PYTERMOR_TRACE_RENDERS
yare-yare-daze

PYTERMOR_PREFER_RGB
YES

1.11 Documentation guidelines

(mostly as a reminder for myself)

- 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):

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

CHAPTER

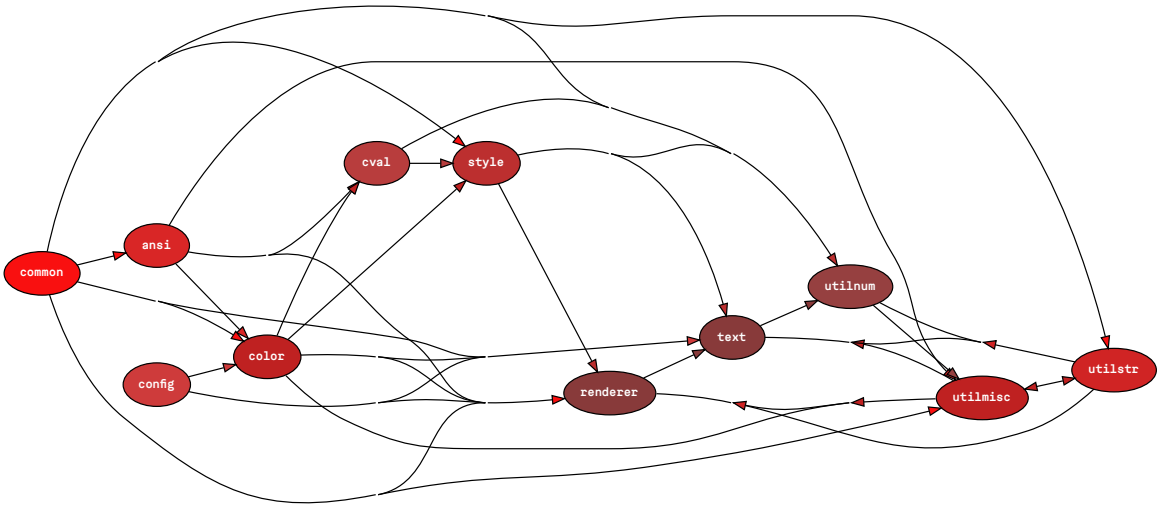
TWO

API REFERENCE

A AA

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

cv :noindex:



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** – Sequence internal parameters, existnce and valid amount depends on sequence type.

`assemble()`

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

Return type
str

`property params: t.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** – Sequence internal parameters, existnce and valid amount depends on sequence type.

`assemble()`

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

Return type
str

`property params: t.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** – Sequence internal parameters, existnce and valid amount depends on sequence type.

`assemble()`

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

Return type
str

`property params: t.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** – Sequence internal parameters, existnce and valid amount depends on sequence type.

assemble()

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

Return type
str

property params: t.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** –
- **short_name** –
- **params** –

assemble()

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

Return type
str

property params: t.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.

When cast to *str*, as all other sequences, invokes *assemble()* method and transforms into encoded control sequence string. It is possible to add of one SGR sequence to another, resulting in a new one with merged params (see examples).

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.

```
>>> SequenceSGR(IntCode.HI_CYAN, 'underlined', 1)
<SGR[96,4,1]>
>>> SequenceSGR(31) + SequenceSGR(1) == SequenceSGR(31, 1)
True
```

Parameters

- **args** – 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
  - *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
[]
```

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 "next 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
t.Tuple[int, int] | None

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.

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)
```

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.

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.
`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: pytermor.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 "0xFFFFFF".

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** – Color RGB value, e.g. 0x5f0000.
- **code** – Int code for a color setup, e.g. 52.
- **name** – Name of the color, e.g. "dark-red".
- **register** – If *True*, add color to registry for resolving by name.
- **index** – If *True*, add color to approximation index.
- **aliases** – Alternative color names (used in *resolve_color()*).
- **color16_equiv** – *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 "0xFFFFFF".

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** – Color RGB value, e.g. 0x73a9c2.
- **name** – Name of the color, e.g. "moonstone-blue".
- **register** – If *True*, add color to registry for resolving by name.
- **index** – If *True*, add color to approximation index.
- **aliases** – Alternative color names (used in *resolve_color()*).
- **variation_map** – 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 "0xFFFFFF".

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

Case-insensitive search through registry contents. Search is performed for `IColor` instance named as specified 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 hexadecimal value "#RRGGBB" (or in short form, as "#RGB"), as well as just use an *int* in [0x0; 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

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).

Classes

Align(value)	Align type.
ExtendedEnum(value)	An enumeration.

Exceptions

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

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

Bases: Enum

An enumeration.

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).

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 *init_config()*;
2) or preliminarily set the corresponding environment variables to intended values, and the default config instance will catch up them.

See also:

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 `logging.WARNING` level, so in order to see the tracing attaching the 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.
OutputMode(value)	Determines what types of SGR sequences are allowed to use in the output.
RendererManager()	Class for global rendering mode setup.
SgrRenderer([output_mode, io])	Default renderer invoked by *Text.render()*.
SgrRendererDebugger([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.
TmuxRenderer()	Translates *Styles* attributes into tmux-compatible 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` (preferrable; 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).

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 (IRenderer | *t.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 markup. tmux 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. 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.

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 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.**SgrRendererDebugger**(*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.

```
>>> SgrRendererDebugger(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,bold]>
>>> Style(bg=0x0000ff)
<Style[bg=0000FF]>
>>> Style(fg='DeepSkyBlue1', bg='gray3')
<Style[X39[00AFFF],bg=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`).

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`).

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.

class pytermor.style.**Styles**

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

WARNING = <Style[yellow]>

WARNING_LABEL = <Style[yellow,bold]>

WARNING_ACCENT = <Style[hi-yellow]>

ERROR = <Style[red]>

ERROR_LABEL = <Style[red,bold]>

ERROR_ACCENT = <Style[hi-red]>

CRITICAL = <Style[hi-white,bg=X160[D70000]]>

CRITICAL_LABEL = <Style[hi-white,bg=X160[D70000],bold]>

CRITICAL_ACCENT = <Style[hi-white,bg=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*.

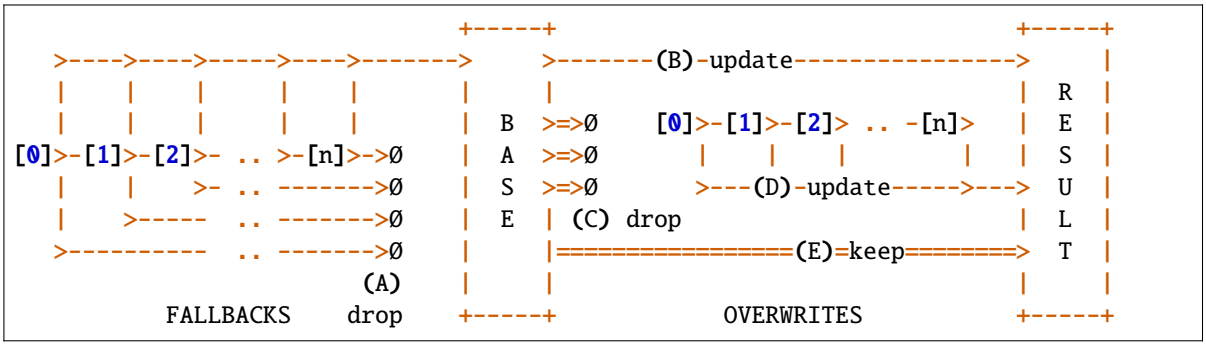
Parameters
fmt (*FT*) – See *FT*.

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



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

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

In the example above:

- the first fallback will be ignored, as `fg` is already set;
- the second fallback will be applied (`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.

(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 5: Overwrite merge algorithm example

```
>>> base = Style(fg='red')
>>> overwrites = [Style(fg='blue'), Style(bold=True), Style(bold=False)]
```

(continues on next page)

(continued from previous page)

```
>>> merge_styles(base, overwrites=overwrites)
<Style[blue]>
```

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

distribute_padded()	**param max_len**
echo([string, fmt, renderer, ...])	.
echoi([string, fmt, renderer, ...])	echo inline
render([string, fmt, renderer, ...])	.

Classes

Fragment([string, fmt, close_this, close_prev])	<Immutable>
FrozenText()	T
IRenderable(*args, **kwds)	I
SimpleTable(*rows[, width, sep, border_st])	Table class with dynamic (not bound to each other) rows.
TemplateEngine([custom_styles])	
Text()	

pytermor.text.**RT**

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

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

class pytermor.text.**IRenderable**(**args*, ***kwds*)

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: pytermor.style.FT = NOOP_STYLE*, ***, *width: int = None*, *align: str | Align = None*, *fill: str = ' '*, *overflow: str = ''*, *pad: int = 0*)

class pytermor.text.**FrozenText**(**fragments: Fragment*, *width: int = None*, *align: str | Align = None*, *fill: str = ' '*, *overflow: str = ''*, *pad: int = 0*)

Bases: *IRenderable*

T

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: pytermor.style.FT = NOOP_STYLE*, ***, *width: int = None*, *align: str | Align = None*, *fill: str = ' '*, *overflow: str = ''*, *pad: int = 0*)

class pytermor.text.**Text**(**fragments: Fragment*, *width: int = None*, *align: str | Align = None*, *fill: str = ' '*, *overflow: str = ''*, *pad: int = 0*)

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.**SimpleTable**(**rows*, *width=None*, *sep=' '*, *border_st=<_NoOpStyle[NOP]>*)

Bases: *IRenderable*

Table class with dynamic (not bound to each other) rows.

Allows 0 or 1 dynamic-width cell in each row, while all the others should be static, i.e., be instances of `FixedString`.

```
>>> 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("biiiiiiiiiiiiiiiiiiiiiiiiiiiig 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** –
- **width** –
- **sep** –
- **border_st** –

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** (*RT | t.Iterable[RT]*) – 2
- **fmt** (*FT*) – 2
- **renderer** (IRenderer) – 2
- **parse_template** (*bool*) – 2
- **no_log** (*bool*) – 2

Returns

Return type
str | t.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** (*RT | t.Iterable[RT]*) –
- **fmt** (*FT*) –
- **renderer** (IRenderer) –
- **parse_template** (*bool*) –
- **nl** (*bool*) –
- **file** (*t.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** (*RT | t.Iterable[RT]*) –
- **fmt** (*FT*) –
- **renderer** (IRenderer) –
- **parse_template** (*bool*) –
- **file** (*t.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: pytermor.text.RT*, *pad_left: int = 0*, *pad_right: int = 0*) → *Text*

Parameters

- **max_len** –
- **values** –
- **pad_left** –
- **pad_right** –

Returns

2.9 pytermor.utilmisc

A

Functions

chunk(items, size)	Split item list into chunks of size `size` and return these chunks as *tuples*.
confirm([attempts, default, keymap, prompt, ...])	Ensure the next action is manually confirmed by user.
flatten(items)	
flatten1(items)	Take a list of nested lists and unpack all nested elements one level up.
get_char_width(char, wait)	General-purpose method for getting width of a character in terminal columns.
get_preferable_wrap_width([force_width])	Return preferable terminal width for comfort reading of wrapped text (max=120).
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.
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 a tuple of three numbers 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]).
measure([level, template])	**param level**
measure_char_width(char[, clear_after, legacy])	Low-level function that returns the exact character width in terminal columns.
median(N[, key])	Find the median of a list of values.
percentile(N, percent[, key])	Find the percentile of a list of values.
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.
trace([enabled, level, label])	**param enabled**
wait_key()	Wait for a key press on the console and return it.

pytermor.utilmisc.**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(make_query_cursor_position())
'SequenceCSI'
>>> get_qname(threading.Thread)
'Thread'
```

Return type
str

pytermor.utilmisc.**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.utilmisc.**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.utilmisc.**flatten**(*items*)

Todo: recursrive

Return type
List[*T*]

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

Find the percentile of a list of values.

Origin
https://code.activestate.com/recipes/511478/

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.utilmisc.**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

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

Parameters

- **enabled** (*bool*) –
- **level** (*int*) –
- **label** (*str*) –

Returns

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

Parameters

- **level** (*int*) –
- **template** (*str*) –

Returns

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
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)
(240.0, 1.0, 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 a tuple of three numbers 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)
(0, 0.0, 0.6)
```

Parameters
hex_value (*int*) – RGB value.

Returns
H, S, V channel values correspondingly.

Return type
Tuple[float, float, float]

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.**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.utilmisc.**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.utilmisc.**wait_key**()

Wait for a key press on the console and return it.

Raises
EOFError –

Return type
t.AnyStr | None

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*, *wait*)

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.
- **wait** (*bool*) – Set to *True* if you prefer slow, but 100% accurate *measuring* (which **blocks** and requires an output tty), or *False* to invoke device-independent, deterministic and non-blocking *guessing*, which works most of the time, although there could be rare cases when it is not accurate.

Return type
int

pytermor.utilmisc.**measure_char_width**(*char*, *clear_after=True*, *legacy=False*)

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.
- **legacy** (*bool*) – For some terminal and interpreter configurations the method can put the application into an endless wait cycle, unless a newline character appears in stdin (for example, when the python debugger is attached). For these cases it is recommended to set this parameter to *True* to switch the internal input reading mode, which helps to avoid this.

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.

Origin
_pytest._io.wcwidth

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}
```

Origin
https://code.activestate.com/recipes/577504/

Return type
int

2.10 pytermor.utilnum

utilnum

Module Attributes

formatter_si	Decimal SI formatter, formats `value` as a unitless value with SI-prefixes; a unit can be provided as an argument of *format()* method.
formatter_si_binary	Binary SI formatter, formats `value` as binary size ("KiB", "MiB") with base = 1024.

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_si(val[, unit, color])	Wrapper for *formatter_si.format()*.
format_si_binary(val[, unit, color])	Wrapper for *formatter_si_binary.format()*.
format_thousand_sep(val[, separator])	Returns input `val` with integer part split into groups of three digits, joined then with `separator` string.
format_time_delta(val_sec[, max_len, color_ov])	Format time delta using suitable format (which depends on `max_len` argument).

Classes

CustomBaseUnit(name[, in_next, ...])	TU
DynamicBaseFormatter(units, *[, color, ...])	Formatter designed for time intervals.
NumHighlighter()	
StaticBaseFormatter([fallback, ...])	Format `value` using settings passed to constructor.

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'
```

Parameters

- **val** (*int | float*) –
- **separator** (*str*) –

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', unless this feature is explicitly disabled with `allow_exp_form` = *False*; in that case:

1) if absolute value is less than 1, zeros will be displayed ('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)
```

(continues on next page)

(continued from previous page)

```
'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'
```

Parameters

- **val** (*float*) – Value to format.
- **req_len** (*int*) – Required output string length.
- **allow_exp_form** (*bool*) – Allow scientific notation usage when no other way of fitting the value into a string of required length.

Raises

ValueError – If value is too big to fit into `req_len` digits and `allow_exp_form` is set to False.

Return type

str

class pytermor.utilnum.**StaticBaseFormatter**(*fallback=None*, ***, *max_value_len=None*, *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*)

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.

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()* and *format_si_binary()*, which will invoke predefined formatters and doesn't require setting up.

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

Parameters

- **fallback** (StaticBaseFormatter) – 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".

- **color** (*bool*) – [default: *False*]
- **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.
- **allow_fractional** (*bool*) – [default: *True*] Allows the usage of fractional values in the output. If set to *False*, the results will be rounded.
- **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

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

Prefix preset used by *format_si()*. Covers values from 10^{-30} to 10^{32}.

PREFIXES_SI_BIN = [None, 'Ki', 'Mi', 'Gi', 'Ti', 'Pi', 'Ei', 'Zi', 'Yi', 'Ri', 'Qi']

Prefix preset used by *format_si_binary()*. Covers values from 0 to 10^{32}.

get_max_len(*unit_ov=None*)

Parameters
unit_ov (*Optional[str]*) – Unit override. Set to *None* to use formatter's own unit.

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_ov=None*, *color_ov=None*)

Parameters

- **val** (*float*) – Input value.
- **unit_ov** (*Optional[str]*) – Unit override. Set to *None* to use formatter's own unit.
- **color_ov** (*Optional[bool]*) – Color mode override, *bool* to enable/disable colorizing, *None* to use formatters' setting value.

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

Return type
pytermor.text.RT

pytermor.utilnum.**formatter_si = StaticBaseFormatter**

Decimal SI formatter, formats `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}.

Usage

```
# either of:
formatter_si.format(<value>, ...)
format_si(<value>, ...)
```

Max len
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.

See
format_si()

pytermor.utilnum.**formatter_si_binary = StaticBaseFormatter**

Binary SI formatter, formats `value` as binary size ("KiB", "MiB") with base = 1024. Unit can be customized.

While being similar to *formatter_si*, this formatter differs in one aspect. Given a variable with default value = 995, formatting its value results in "995 B". After increasing it by 20 we'll have 1015, but it's 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 be in a form of "1.00 KiB".

Usage

```
# either of:
formatter_si_binary.format(<value>, ...)
format_si_binary(<value>, ...)
```

Max len

First things first, the initial `max_value_len` must be at least 5 (not 4), because it is a minimum requirement for formatting values from 1023 to -1023.

The negative values for this formatter are disabled by default and thus will be rounded to 0, which decreases the `max_value_len` minimum value by 1 (to 4).

Total maximum length is `max_value_len + 4` = **8** (base + 1 from separator, 1 from unit and 2 from prefix, assuming all of them have default values defined in *formatter_si_binary*).

See

format_si_binary()

pytermor.utilnum.**format_si**(*val*, *unit=None*, *color=None*)

Wrapper for *formatter_si.format()*.

```
>>> 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'
```

Parameters

- **val** (*float*) – Input value (unitless).
- **unit** (*Optional[str]*) – A unit override [default unit is an empty string].
- **color** (*Optional[bool]*) – If *True*, the result will be colorized depending on prefix type.

Returns

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

Return type

pytermor.text.RT

pytermor.utilnum.**format_si_binary**(*val*, *unit=None*, *color=False*)

Wrapper for *formatter_si_binary.format()*.

```
>>> 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'
```

Parameters

- **val** (*float*) – Input value in bytes.
- **unit** (*Optional[str]*) – A unit override [default unit is "B"].
- **color** (*bool*) – If *True*, the result will be colorized depending on prefix type.

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

Return type
pytermor.text.RT

class pytermor.utilnum.**DynamicBaseFormatter**(*units*, *, *color=False*, *allow_negative=False*, *allow_fractional=True*, *unit_separator=None*, *pad=False*, *plural_suffix=None*, *overflow_msg='OVERFLOW'*)

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

- **units** (*t.List[CustomBaseUnit]*) –
- **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*) –

property **max_len**: int

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

Returns
Maximum possible output string length.

format(*val*, *color_ov=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** (*float*) – Input value.
- **color_ov** (*Optional[bool]*) – Color mode override, *bool* to enable/disable colorizing, *None* to use formatters' setting value.

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

Return type
pytermor.text.RT

format_base(*val*, *color_ov=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** (*float*) – Input value.
- **color_ov** (*bool*) – Color mode override, *bool* to enable/disable colorizing, *None* to use formatters' setting value.

Returns
Formatted value as *Text* if colorizing is on; as *str* otherwise. Returns *None* on overflow.

Return type
RT | None

class pytermor.utilnum.**CustomBaseUnit**(*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.

pytermor.utilnum.**format_time_delta**(*val_sec*, *max_len=None*, *color_ov=None*)

Format time delta using suitable format (which depends on `max_len` argument). 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",

There are predefined formatters with output length of **3, 4, 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.

```
>>> format_time_delta(10, 3)
'10s'
>>> format_time_delta(10, 6)
'10.0s'
```

(continues on next page)

(continued from previous page)

```
>>> format_time_delta(15350, 4)
'4 h'
>>> format_time_delta(15350)
'4h 15min'
```

Parameters

- **val_sec** (*float*) – Value to format.
- **max_len** (*Optional[int]*) – Maximum output string length (total).
- **color_ov** (*Optional[bool]*) – Color mode override, *bool* to enable/disable colorizing depending on unit type, *None* to use formatters' setting value.

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
PT	pattern type
RT	replacer type
MT	# map
AT	alias of Union[OmniFilter, Type[OmniFilter]]

Functions

apply_filters(string, *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`.
pad(n)	
padv(n)	
rjust_sgr(s, width[, fillchar, actual_len])	SGR-formatting-aware implementation of `str.rjust`.
wrap_sgr(raw_input, width[, indent_first, ...])	A workaround to make standard library `textwrap.wrap()` more friendly to an SGR-formatted strings.

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(*args, **kwds)	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*.
NoopFilter(*args, **kwds)	
OmniDecoder(*args, **kwds)	
OmniEncoder(*args, **kwds)	
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])	
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)	

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.**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*) –

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.**PT**

pattern type

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

pytermor.utilstr.**RT**

replacer type

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

pytermor.utilstr.**MT**

map

alias of Dict[int, *IT*]

pytermor.utilstr.**AT**

alias of Union[OmniFilter, Type[OmniFilter]]

class pytermor.utilstr.**IFilter**(**args*, ***kwds*)

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** –

- **width** –
- **sgr_aware** –

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 6: 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 7: 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 8: 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 – 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 – 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.**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 – 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 prepended with same char by default, but this behaviour can be disabled with `keep_newlines` = *False*.

```
>>> NonPrintsStringVisualizer().apply('A  B  C')
'A␣␣B␣␣C'
>>> apply_filters('1. D'+os.linesep+'2. L ', NonPrintsStringVisualizer(keep_
↪newlines=False))
'1.␣D2.␣L␣'
```

Parameters
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 – 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**(*string*, **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
>>> apply_filters(f'{SeqIndex.RED}test{SeqIndex.COLOR_OFF}', SgrStringReplacer(r'E}
↪'))
'E[31mtestE[39m'
```

Note that type of `s` argument must be same as `StringFilter` parameterized type, i.e. `ReplaceNonAsciiBytes` is `StringFilter` type, so you can apply it only to bytes-type strings.

Parameters

- **string** (*pytermor.utilstr.IT*) – String to filter.

- **args** (*Union[OmniFilter, Type[OmniFilter]]*) – OmniFilter instance(s) or OmniFilter type(s).

Returns
Filtered 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

CHAPTER

THREE

CHANGELOG

3.1 Releases

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

3.1.1 pending

till `b22ccc`

- [REFACTOR] *`distribute_padded`* overloads
- [DOCS] *`utilnum`* module
- [DOCS] `Changelog` update
- [FIX] critical *`Styles`* color
- [NEW] *`echoi`*, *`flatten`*, *`flatten1`* methods; *`SimpleTable`* class
- 2023 copytight update
- [NEW] methods *`percentile`* and *`median`* ; `render_benchmark` example
- [REFACTOR] *`IRenderable`* rewrite
- [FIX] `NumHighlighter`
- [REFACTOR] moved color transformations and type vars from `_commons`

3.1.2 v2.32-dev

Jan 23

- [FIX] *`TmuxRenderer`* RGB output
- [NEW] `PrefixedUnitFormatter` inheritance
- [NEW] `StyledString`
- [NEW] `pad`, `padv` helpers
- [NEW] `String` and `FixedString` base renderables
- [TESTS] integrated in-code doctests into pytest
- [NEW] *`IRenderable`* result caching
- [NEW] `Renderable` __eq__ methods

- [FIX] `format_prefixed` and *format_auto_float* inaccuracies
- [DOCS] *utilnum* update
- [TESTS] *utilnum* update
- [NEW] subsecond delta support for `TimeDeltaFormatter`
- [NEW] *utilmisc* get_char_width(), guess_char_width(), measure_char_width()
- [NEW] *Color256* aliases "colorNN"
- [NEW] `prefix_refpoint_shift` argument of PrefixedUnitFormatter
- [NEW] `NumHighlighter` from `es7s`, colorizing options of *utilnum* helpers
- [FIX] `Text.prepend` typing
- [DOCS] docstrings, typing
- [NEW] *style.merge_styles()*
- [NEW] style merging strategies: *merge_fallback()*, *merge_overwrite*
- [DOCS] *utilnum* module

3.1.3 v2.23-dev

- [FIX] `OmniHexPrinter` missed out newlines
- [NEW] split *Text* to *Text* and *FrozenText*
- [NEW] *SgrRenderer* now supports non-default IO stream specifying
- [NEW] `utilstr.StringHexPrinter` and `utilstr.StringUcpPrinter`
- [NEW] `Printers` and `Mappers`
- [NEW] *dump* printer caching
- [NEW] extracted *resolve*, *approximate*, *find_closest* from `Color` class to module level, as well as color transform functions
- [NEW] add missing *hsv_to_rgb* function

3.1.4 v2.18-dev

- [NEW] *cval* autobuild.
- [NEW] `ArgCountError` migrated from `es7s/core`.
- [NEW] `black` code style.
- [NEW] Add `OmniHexPrinter` and *chunk()* helper.
- [NEW] Typehinting.
- [FIX] Disabled automatic rendering of *echo()* and *render()*.

3.1.5 v2.14-dev

Dec 22

- [NEW] *confirm()* helper command.
- [NEW] EscapeSequenceStringReplacer filter.
- [NEW] examples/terminal_benchmark script.
- [NEW] StringFilter and OmniFilter classes.
- [DOCS] Docs design fixes.
- [NEW] Minor core improvements.
- [TESTS] Tests for *color* module.
- [NEW] RGB and variations full support.

3.1.6 v2.6-dev

Nov 22

- [REFACTOR] Got rid of Span class.
- [REFACTOR] Rewrite of *color* module.
- [REFACTOR] Changes in ConfigurableRenderer.force_styles logic.
- [NEW] *Text* nesting.
- [NEW] TemplateEngine implementation.
- [REFACTOR] Package reorganizing.

3.1.7 v2.2-dev

Oct 22

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

3.1.8 v2.1-dev

Aug 22

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

3.1.9 v2.0-dev

Jul 22

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

3.1.10 v1.8

Jun 22

- [NEW] Added 3 formatters: `format_prefixed_unit`, *`format_time_delta`*, *`format_auto_float`*.
- [NEW] `format_prefixed_unit` extended for working with decimal and binary metric prefixes.
- [NEW] *`format_time_delta`* extended with new settings.
- [REFACTOR] Value rounding transferred from *`format_auto_float`* to `format_prefixed_unit`.
- [REFACTOR] Utility classes reorganization.
- [TESTS] Unit tests output formatting.
- [NEW] `sequence.NOOP` SGR sequence and `span.NOOP` format.
- [NEW] Max decimal points for `auto_float` extended from (2) to (max-2).

3.1.11 v1.7

May 22

- [NEW] `Span` constructor can be called without arguments.
- [NEW] Added SGR code lists.
- [FIX] Print reset sequence as `\e[m` instead of `\e[0m`.
- [NEW] Added *`ljust_sgr`*, *`rjust_sgr`*, *`center_sgr`* util functions to align strings with SGRs correctly.
- [NEW] Added `span.BG_BLACK` format.

3.1.12 v1.6

- [REFACTOR] Ridded of `EmptyFormat` and `AbstractFormat` classes.
- [REFACTOR] Renamed `code` module to `sgr` because of conflicts in PyCharm debugger (`pydevd_console_integration.py`).
- [TESTS] Excluded `tests` dir from distribution package.

3.1.13 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.

3.1.14 v1.4

- [NEW] `Span.wrap()` now accepts any type of argument, not only *str*.
- [REFACTOR] Rebuilt `Sequence` inheritance tree.
- [NEW] Added equality methods for *SequenceSGR* and `Span` classes/subclasses.
- [TESTS] Added some tests for `fmt.*` and `seq.*` classes.

3.1.15 v1.3

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

3.1.16 v1.2

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

3.1.17 v1.1

Apr 22

- [NEW] Autoformat feature.

3.1.18 v1.0

- First public version.

3.1.19 v0.90

Mar 22

- First commit.

CHAPTER

FOUR

LICENSE

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

N

O

P

R

S

T

U

V

W

X

Y