pytermor

Release 2.0.0-dev.14

Alexandr Shavykin

Aug 05, 2022

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 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 span

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

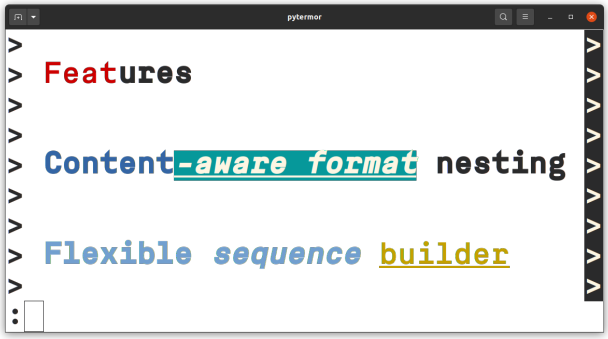
Content-aware format nesting

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* with *build()* method, 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 sequence, build

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

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

256 colors / True Color support

The library supports extended color modes:

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

```
from pytermor import color_indexed, color_rgb, sequence

start_color = 41
for idx, c in enumerate(range(start_color, start_color+(36*6), 36)):
    print(f'{color_indexed(c)}{sequence.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'{color_rgb(r, g, b)}{sequence.COLOR_OFF}', end='')
```

Customizable output formats

@TODO

String and number formatters

@TODO

1.2 High-level abstractions

1.2.1 Colors and Styles

1.2.2 Output format control

1.2.3 Color mode fallbacks

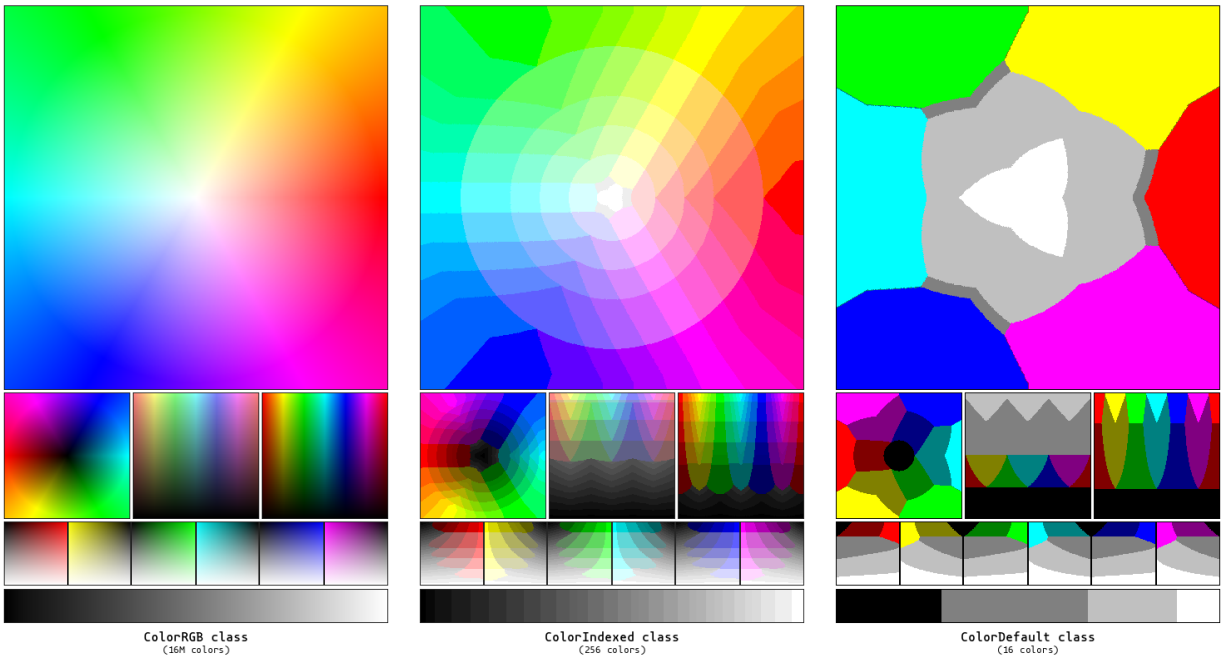


Fig. 1: Color approximations for indexed modes

1.2.4 Core API

@EXAMPLES

1.3 Low-level abstractions

So, what's happening under the hood?

1.3.1 Format soft reset

There are two ways to manage color and attribute termination:

- hard reset (SGR-0 or `e[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 `e[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 sequence, span, Span

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

orig_text = span.BOLD(f'this is {sequence.BG_GRAY}the original{sequence.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 `sequence.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).

1.3.2 Working with Spans

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

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

Each sequence param can be specified as:

- string key (see *Preset list*);
- integer param value;
- existing *SequenceSGR* instance (params will be extracted).

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

1.3.3 Creating and applying SGRs

You can use any of predefined sequences from *sequence* or create your own via standard constructor. Valid argument values as well as preset constants are described in *Preset list* page.

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

- *build()* for non-specific sequences;

  Important: `SequenceSGR` with zero params was specifically implemented to translate into an empty string and not into `e[m`, which would make sense, but also could be very entangling, as terminal emulators interpret that sequence as `e[0m`, which is *hard* reset sequence.

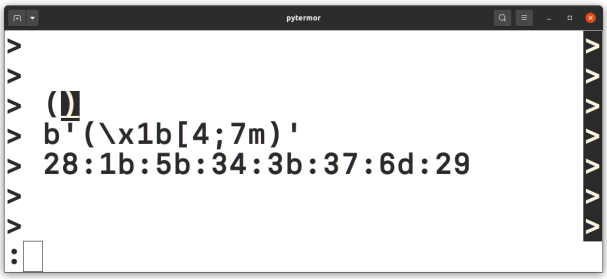
- *color_indexed()* for complex color selection sequences operating in 256-colors mode (for a complete list see *Preset list*);
- *color_rgb()* for setting the colors in True Color 16M mode (however, some terminal emulators doesn't support it).

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

```
from pytermor.sequence import SequenceSGR

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

print(msg + f'{SequenceSGR(0).encode()}')
print(str(msg.encode()))
print(msg.encode().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 hexademical string representation.

1.3.4 SGR sequence structure

1. `\x1b` is ESC *control character*, which opens a control sequence.
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.

1.3.5 Combining SGRs

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

```
from pytermor import sequence
from pytermor.sequence import SequenceSGR

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

1.3.6 Core API

- @TODO
- *Span* constructor
- *Span.from_seq()*

1.4 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 (library references that palette as *default*, see *ColorDefault*), and second one consisting of 256 colors (referenced as *indexed*, e.g. *ColorIndexed*). 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)

1.4.1 Meta, attributes, breakers

	Name	INT	SEQ	SPN	CLR	STY	Description
Meta							
	NOOP		V	V	V	V	No-operation; always encoded 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* color_indexed *and* color_rgb *instead*
	BG_COLOR_EXTENDED	48					Set background color [*indexed/RGB* mode]; *use* color_indexed *and* color_rgb *instead*
	OVERLINED	53	V	V		V	Overline; *not widely supported*
Breakers							
	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.

1.4.2 Default colors

	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

1.4.3 Indexed colors

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
	IDX_BLACK[3]	0			V		#000000	Black
	IDX_MAROON	1			V		#800000	Maroon
	IDX_GREEN_5	2			V		#008000	**Green**[4]
	IDX_OLIVE	3			V		#808000	Olive
	IDX_NAVY	4			V		#000080	Navy
	IDX_PURPLE	5			V		#800080	Purple
	IDX_TEAL	6			V		#008080	Teal
	IDX_SILVER	7			V		#c0c0c0	Silver
	IDX_GREY	8			V		#808080	Grey
	IDX_RED	9			V		#ff0000	Red
	IDX_LIME	10			V		#00ff00	Lime
	IDX_YELLOW	11			V		#ffff00	Yellow
	IDX_BLUE	12			V		#0000ff	Blue
	IDX_FUCHSIA	13			V		#ff00ff	Fuchsia
	IDX_AQUA	14			V		#00ffff	Aqua
	IDX_WHITE	15			V		#ffffff	White
	IDX_GREY_0	16			V		#000000	Grey0
	IDX_NAVY_BLUE	17			V		#00005f	NavyBlue
	IDX_DARK_BLUE	18			V		#000087	DarkBlue
	IDX_BLUE_3	19			V		#0000af	Blue3
	IDX_BLUE_2	20			V		#0000d7	**Blue3**
	IDX_BLUE_1	21			V		#0000ff	Blue1
	IDX_DARK_GREEN	22			V		#005f00	DarkGreen
	IDX_DEEP_SKY_BLUE_7	23			V		#005f5f	**DeepSkyBlue4**
	IDX_DEEP_SKY_BLUE_6	24			V		#005f87	**DeepSkyBlue4**
	IDX_DEEP_SKY_BLUE_5	25			V		#005faf	**DeepSkyBlue4**
	IDX_DODGER_BLUE_3	26			V		#005fd7	DodgerBlue3
	IDX_DODGER_BLUE_2	27			V		#005fff	DodgerBlue2
	IDX_GREEN_4	28			V		#008700	Green4
	IDX_SPRING_GREEN_4	29			V		#00875f	SpringGreen4
	IDX_TURQUOISE_4	30			V		#008787	Turquoise4
	IDX_DEEP_SKY_BLUE_4	31			V		#0087af	**DeepSkyBlue3**
	IDX_DEEP_SKY_BLUE_3	32			V		#0087d7	DeepSkyBlue3
	IDX_DODGER_BLUE_1	33			V		#0087ff	DodgerBlue1
	IDX_GREEN_3	34			V		#00af00	Green3
	IDX_SPRING_GREEN_3	35			V		#00af5f	SpringGreen3
	IDX_DARK_CYAN	36			V		#00af87	DarkCyan
	IDX_LIGHT_SEA_GREEN	37			V		#00afaf	LightSeaGreen
	IDX_DEEP_SKY_BLUE_2	38			V		#00afd7	DeepSkyBlue2
	IDX_DEEP_SKY_BLUE_1	39			V		#00afff	DeepSkyBlue1
	IDX_GREEN_2	40			V		#00d700	**Green3**
	IDX_SPRING_GREEN_3	41			V		#00d75f	SpringGreen3
	IDX_SPRING_GREEN_2	42			V		#00d787	SpringGreen2
	IDX_CYAN_3	43			V		#00d7af	Cyan3
	IDX_DARK_TURQUOISE	44			V		#00d7d7	DarkTurquoise
	IDX_TURQUOISE_2	45			V		#00d7ff	Turquoise2
	IDX_GREEN_1	46			V		#00ff00	Green1

continues on next page

Table 2 – continued from previous page

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
	IDX_SPRING_GREEN_2	47			V		#00ff5f	SpringGreen2
	IDX_SPRING_GREEN_1	48			V		#00ff87	SpringGreen1
	IDX_MEDIUM_SPRING_GREEN	49			V		#00ffaf	MediumSpringGreen
	IDX_CYAN_2	50			V		#00ffd7	Cyan2
	IDX_CYAN_1	51			V		#00ffff	Cyan1
	IDX_DARK_RED_2	52			V		#5f0000	**DarkRed**
	IDX_DEEP_PINK_8	53			V		#5f005f	**DeepPink4**
	IDX_PURPLE_4	54			V		#5f0087	Purple4
	IDX_PURPLE_4	55			V		#5f00af	Purple4
	IDX_PURPLE_3	56			V		#5f00d7	Purple3
	IDX_BLUE_VIOLET	57			V		#5f00ff	BlueViolet
	IDX_ORANGE_4	58			V		#5f5f00	Orange4
	IDX_GREY_37	59			V		#5f5f5f	Grey37
	IDX_MEDIUM_PURPLE_7	60			V		#5f5f87	**MediumPurple4**
	IDX_SLATE_BLUE_3	61			V		#5f5faf	SlateBlue3
	IDX_SLATE_BLUE_3	62			V		#5f5fd7	SlateBlue3
	IDX_ROYAL_BLUE_1	63			V		#5f5fff	RoyalBlue1
	IDX_CHARTREUSE_6	64			V		#5f8700	**Chartreuse4**
	IDX_DARK_SEA_GREEN_9	65			V		#5f875f	**DarkSeaGreen4**
	IDX_PALE_TURQUOISE_4	66			V		#5f8787	PaleTurquoise4
	IDX_STEEL_BLUE	67			V		#5f87af	SteelBlue
	IDX_STEEL_BLUE_3	68			V		#5f87d7	SteelBlue3
	IDX_CORNFLOWER_BLUE	69			V		#5f87ff	CornflowerBlue
	IDX_CHARTREUSE_5	70			V		#5faf00	**Chartreuse3**
	IDX_DARK_SEA_GREEN_8	71			V		#5faf5f	**DarkSeaGreen4**
	IDX_CADET_BLUE_2	72			V		#5faf87	**CadetBlue**
	IDX_CADET_BLUE_1	73			V		#5fafaf	**CadetBlue**
	IDX_SKY_BLUE_3	74			V		#5fafd7	SkyBlue3
	IDX_STEEL_BLUE_1	75			V		#5fafff	SteelBlue1
	IDX_CHARTREUSE_4	76			V		#5fd700	**Chartreuse3**
	IDX_PALE_GREEN_4	77			V		#5fd75f	**PaleGreen3**
	IDX_SEA_GREEN_3	78			V		#5fd787	SeaGreen3
	IDX_AQUAMARINE_3	79			V		#5fd7af	Aquamarine3
	IDX_MEDIUM_TURQUOISE	80			V		#5fd7d7	MediumTurquoise
	IDX_STEEL_BLUE_1	81			V		#5fd7ff	SteelBlue1
	IDX_CHARTREUSE_2	82			V		#5fff00	Chartreuse2
	IDX_SEA_GREEN_2	83			V		#5fff5f	SeaGreen2
	IDX_SEA_GREEN_1	84			V		#5fff87	SeaGreen1
	IDX_SEA_GREEN_1	85			V		#5fffaf	SeaGreen1
	IDX_AQUAMARINE_2	86			V		#5fffd7	**Aquamarine1**
	IDX_DARK_SLATE_GRAY_2	87			V		#5fffff	DarkSlateGray2
	IDX_DARK_RED_1	88			V		#870000	**DarkRed**
	IDX_DEEP_PINK_7	89			V		#87005f	**DeepPink4**
	IDX_DARK_MAGENTA_2	90			V		#870087	**DarkMagenta**
	IDX_DARK_MAGENTA_1	91			V		#8700af	**DarkMagenta**
	IDX_DARK_VIOLET_2	92			V		#8700d7	**DarkViolet**
	IDX_PURPLE	93			V		#8700ff	Purple
	IDX_ORANGE_3	94			V		#875f00	**Orange4**
	IDX_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
	IDX_PLUM_4	96			V		#875f87	Plum4
	IDX_MEDIUM_PURPLE_6	97			V		#875faf	**MediumPurple3**
	IDX_MEDIUM_PURPLE_5	98			V		#875fd7	**MediumPurple3**
	IDX_SLATE_BLUE_1	99			V		#875fff	SlateBlue1
	IDX_YELLOW_4	100			V		#878700	Yellow4
	IDX_WHEAT_4	101			V		#87875f	Wheat4
	IDX_GREY_53	102			V		#878787	Grey53
	IDX_LIGHT_SLATE_GREY	103			V		#8787af	LightSlateGrey
	IDX_MEDIUM_PURPLE_4	104			V		#8787d7	**MediumPurple**
	IDX_LIGHT_SLATE_BLUE	105			V		#8787ff	LightSlateBlue
	IDX_YELLOW_4	106			V		#87af00	Yellow4
	IDX_DARK_OLIVE_GREEN_6	107			V		#87af5f	**DarkOliveGreen3**
	IDX_DARK_SEA_GREEN_7	108			V		#87af87	**DarkSeaGreen**
	IDX_LIGHT_SKY_BLUE_3	109			V		#87afaf	LightSkyBlue3
	IDX_LIGHT_SKY_BLUE_2	110			V		#87afd7	**LightSkyBlue3**
	IDX_SKY_BLUE_2	111			V		#87afff	SkyBlue2
	IDX_CHARTREUSE_3	112			V		#87d700	**Chartreuse2**
	IDX_DARK_OLIVE_GREEN_4	113			V		#87d75f	**DarkOliveGreen3**
	IDX_PALE_GREEN_3	114			V		#87d787	PaleGreen3
	IDX_DARK_SEA_GREEN_5	115			V		#87d7af	**DarkSeaGreen3**
	IDX_DARK_SLATE_GRAY_3	116			V		#87d7d7	DarkSlateGray3
	IDX_SKY_BLUE_1	117			V		#87d7ff	SkyBlue1
	IDX_CHARTREUSE_1	118			V		#87ff00	Chartreuse1
	IDX_LIGHT_GREEN_2	119			V		#87ff5f	**LightGreen**
	IDX_LIGHT_GREEN_1	120			V		#87ff87	**LightGreen**
	IDX_PALE_GREEN_1	121			V		#87ffaf	PaleGreen1
	IDX_AQUAMARINE_1	122			V		#87ffd7	Aquamarine1
	IDX_DARK_SLATE_GRAY_1	123			V		#87ffff	DarkSlateGray1
	IDX_RED_3	124			V		#af0000	Red3
	IDX_DEEP_PINK_6	125			V		#af005f	**DeepPink4**
	IDX_MEDIUM_VIOLET_RED	126			V		#af0087	MediumVioletRed
	IDX_MAGENTA_6	127			V		#af00af	**Magenta3**
	IDX_DARK_VIOLET_1	128			V		#af00d7	**DarkViolet**
	IDX_PURPLE	129			V		#af00ff	Purple
	IDX_DARK_ORANGE_3	130			V		#af5f00	DarkOrange3
	IDX_INDIAN_RED_4	131			V		#af5f5f	**IndianRed**
	IDX_HOT_PINK_5	132			V		#af5f87	**HotPink3**
	IDX_MEDIUM_ORCHID_4	133			V		#af5faf	**MediumOrchid3**
	IDX_MEDIUM_ORCHID_3	134			V		#af5fd7	**MediumOrchid**
	IDX_MEDIUM_PURPLE_2	135			V		#af5fff	MediumPurple2
	IDX_DARK_GOLDENROD	136			V		#af8700	DarkGoldenrod
	IDX_LIGHT_SALMON_3	137			V		#af875f	LightSalmon3
	IDX_ROSY_BROWN	138			V		#af8787	RosyBrown
	IDX_GREY_63	139			V		#af87af	Grey63
	IDX_MEDIUM_PURPLE_3	140			V		#af87d7	**MediumPurple2**
	IDX_MEDIUM_PURPLE_1	141			V		#af87ff	MediumPurple1
	IDX_GOLD_3	142			V		#afaf00	Gold3
	IDX_DARK_KHAKI	143			V		#afaf5f	DarkKhaki
	IDX_NAVAJO_WHITE_3	144			V		#afaf87	NavajoWhite3

continues on next page

Table 2 – continued from previous page

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
	IDX_GREY_69	145			V		#afafaf	Grey69
	IDX_LIGHT_STEEL_BLUE_3	146			V		#afafd7	LightSteelBlue3
	IDX_LIGHT_STEEL_BLUE_2	147			V		#afafff	**LightSteelBlue**
	IDX_YELLOW_3	148			V		#afd700	Yellow3
	IDX_DARK_OLIVE_GREEN_5	149			V		#afd75f	**DarkOliveGreen3**
	IDX_DARK_SEA_GREEN_6	150			V		#afd787	**DarkSeaGreen3**
	IDX_DARK_SEA_GREEN_4	151			V		#afd7af	**DarkSeaGreen2**
	IDX_LIGHT_CYAN_3	152			V		#afd7d7	LightCyan3
	IDX_LIGHT_SKY_BLUE_1	153			V		#afd7ff	LightSkyBlue1
	IDX_GREEN_YELLOW	154			V		#afff00	GreenYellow
	IDX_DARK_OLIVE_GREEN_3	155			V		#afff5f	**DarkOliveGreen2**
	IDX_PALE_GREEN_2	156			V		#afff87	**PaleGreen1**
	IDX_DARK_SEA_GREEN_3	157			V		#afffaf	**DarkSeaGreen2**
	IDX_DARK_SEA_GREEN_1	158			V		#afffd7	DarkSeaGreen1
	IDX_PALE_TURQUOISE_1	159			V		#afffff	PaleTurquoise1
	IDX_RED_3	160			V		#d70000	Red3
	IDX_DEEP_PINK_5	161			V		#d7005f	**DeepPink3**
	IDX_DEEP_PINK_3	162			V		#d70087	DeepPink3
	IDX_MAGENTA_3	163			V		#d700af	Magenta3
	IDX_MAGENTA_5	164			V		#d700d7	**Magenta3**
	IDX_MAGENTA_4	165			V		#d700ff	**Magenta2**
	IDX_DARK_ORANGE_2	166			V		#d75f00	**DarkOrange3**
	IDX_INDIAN_RED_3	167			V		#d75f5f	**IndianRed**
	IDX_HOT_PINK_4	168			V		#d75f87	**HotPink3**
	IDX_HOT_PINK_3	169			V		#d75faf	**HotPink2**
	IDX_ORCHID_3	170			V		#d75fd7	**Orchid**
	IDX_MEDIUM_ORCHID_2	171			V		#d75fff	**MediumOrchid1**
	IDX_ORANGE_2	172			V		#d78700	**Orange3**
	IDX_LIGHT_SALMON_2	173			V		#d7875f	**LightSalmon3**
	IDX_LIGHT_PINK_2	174			V		#d78787	**LightPink3**
	IDX_PINK_2	175			V		#d787af	**Pink3**
	IDX_PLUM_3	176			V		#d787d7	Plum3
	IDX_VIOLET	177			V		#d787ff	Violet
	IDX_GOLD_2	178			V		#d7af00	**Gold3**
	IDX_LIGHT_GOLDENROD_5	179			V		#d7af5f	**LightGoldenrod3**
	IDX_TAN	180			V		#d7af87	Tan
	IDX_MISTY_ROSE_3	181			V		#d7afaf	MistyRose3
	IDX_THISTLE_3	182			V		#d7afd7	Thistle3
	IDX_PLUM_2	183			V		#d7afff	Plum2
	IDX_YELLOW_3	184			V		#d7d700	Yellow3
	IDX_KHAKI_3	185			V		#d7d75f	Khaki3
	IDX_LIGHT_GOLDENROD_3	186			V		#d7d787	**LightGoldenrod2**
	IDX_LIGHT_YELLOW_3	187			V		#d7d7af	LightYellow3
	IDX_GREY_84	188			V		#d7d7d7	Grey84
	IDX_LIGHT_STEEL_BLUE_1	189			V		#d7d7ff	LightSteelBlue1
	IDX_YELLOW_2	190			V		#d7ff00	Yellow2
	IDX_DARK_OLIVE_GREEN_2	191			V		#d7ff5f	**DarkOliveGreen1**
	IDX_DARK_OLIVE_GREEN_1	192			V		#d7ff87	DarkOliveGreen1
	IDX_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
	IDX_HONEYDEW_2	194			V		#d7ffd7	Honeydew2
	IDX_LIGHT_CYAN_1	195			V		#d7ffff	LightCyan1
	IDX_RED_1	196			V		#ff0000	Red1
	IDX_DEEP_PINK_4	197			V		#ff005f	**DeepPink2**
	IDX_DEEP_PINK_2	198			V		#ff0087	**DeepPink1**
	IDX_DEEP_PINK_1	199			V		#ff00af	DeepPink1
	IDX_MAGENTA_2	200			V		#ff00d7	Magenta2
	IDX_MAGENTA_1	201			V		#ff00ff	Magenta1
	IDX_ORANGE_RED_1	202			V		#ff5f00	OrangeRed1
	IDX_INDIAN_RED_2	203			V		#ff5f5f	**IndianRed1**
	IDX_INDIAN_RED_1	204			V		#ff5f87	IndianRed1
	IDX_HOT_PINK_2	205			V		#ff5faf	**HotPink**
	IDX_HOT_PINK_1	206			V		#ff5fd7	**HotPink**
	IDX_MEDIUM_ORCHID_1	207			V		#ff5fff	MediumOrchid1
	IDX_DARK_ORANGE_1	208			V		#ff8700	**DarkOrange**
	IDX_SALMON_1	209			V		#ff875f	Salmon1
	IDX_LIGHT_CORAL	210			V		#ff8787	LightCoral
	IDX_PALE_VIOLET_RED_1	211			V		#ff87af	PaleVioletRed1
	IDX_ORCHID_2	212			V		#ff87d7	Orchid2
	IDX_ORCHID_1	213			V		#ff87ff	Orchid1
	IDX_ORANGE_1	214			V		#ffaf00	Orange1
	IDX_SANDY_BROWN	215			V		#ffaf5f	SandyBrown
	IDX_LIGHT_SALMON_1	216			V		#ffaf87	LightSalmon1
	IDX_LIGHT_PINK_1	217			V		#ffafaf	LightPink1
	IDX_PINK_1	218			V		#ffafd7	Pink1
	IDX_PLUM_1	219			V		#ffafff	Plum1
	IDX_GOLD_1	220			V		#ffd700	Gold1
	IDX_LIGHT_GOLDENROD_4	221			V		#ffd75f	**LightGoldenrod2**
	IDX_LIGHT_GOLDENROD_2	222			V		#ffd787	LightGoldenrod2
	IDX_NAVAJO_WHITE_1	223			V		#ffd7af	NavajoWhite1
	IDX_MISTY_ROSE_1	224			V		#ffd7d7	MistyRose1
	IDX_THISTLE_1	225			V		#ffd7ff	Thistle1
	IDX_YELLOW_1	226			V		#ffff00	Yellow1
	IDX_LIGHT_GOLDENROD_1	227			V		#ffff5f	LightGoldenrod1
	IDX_KHAKI_1	228			V		#ffff87	Khaki1
	IDX_WHEAT_1	229			V		#ffffaf	Wheat1
	IDX_CORNSILK_1	230			V		#ffffd7	Cornsilk1
	IDX_GREY_100	231			V		#ffffff	Grey100
	IDX_GREY_3	232			V		#080808	Grey3
	IDX_GREY_7	233			V		#121212	Grey7
	IDX_GREY_11	234			V		#1c1c1c	Grey11
	IDX_GREY_15	235			V		#262626	Grey15
	IDX_GREY_19	236			V		#303030	Grey19
	IDX_GREY_23	237			V		#3a3a3a	Grey23
	IDX_GREY_27	238			V		#444444	Grey27
	IDX_GREY_30	239			V		#4e4e4e	Grey30
	IDX_GREY_35	240			V		#585858	Grey35
	IDX_GREY_39	241			V		#626262	Grey39
	IDX_GREY_42	242			V		#6c6c6c	Grey42

continues on next page

Table 2 – continued from previous page

	Name	INT	SEQ	SPN	CLR	STY	RGB code	XTerm name
	IDX_GREY_46	243			V		#767676	Grey46
	IDX_GREY_50	244			V		#808080	Grey50
	IDX_GREY_54	245			V		#8a8a8a	Grey54
	IDX_GREY_58	246			V		#949494	Grey58
	IDX_GREY_62	247			V		#9e9e9e	Grey62
	IDX_GREY_66	248			V		#a8a8a8	Grey66
	IDX_GREY_70	249			V		#b2b2b2	Grey70
	IDX_GREY_74	250			V		#bcbcbc	Grey74
	IDX_GREY_78	251			V		#c6c6c6	Grey78
	IDX_GREY_82	252			V		#d0d0d0	Grey82
	IDX_GREY_85	253			V		#dadada	Grey85
	IDX_GREY_89	254			V		#e4e4e4	Grey89
	IDX_GREY_93	255			V		#eeeeee	Grey93

Sources

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

1.5 Color palette

Actual colors of *default* palette depend on user's terminal settings, i.e. the result color of `ColorDefault` 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).

Important: `@TODO` 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 `ColorDefault` 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.

Approximation algorithm is as simple as iterating through all colors in the *lookup table* (which contains all possible ... `@TODO` finish

[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.

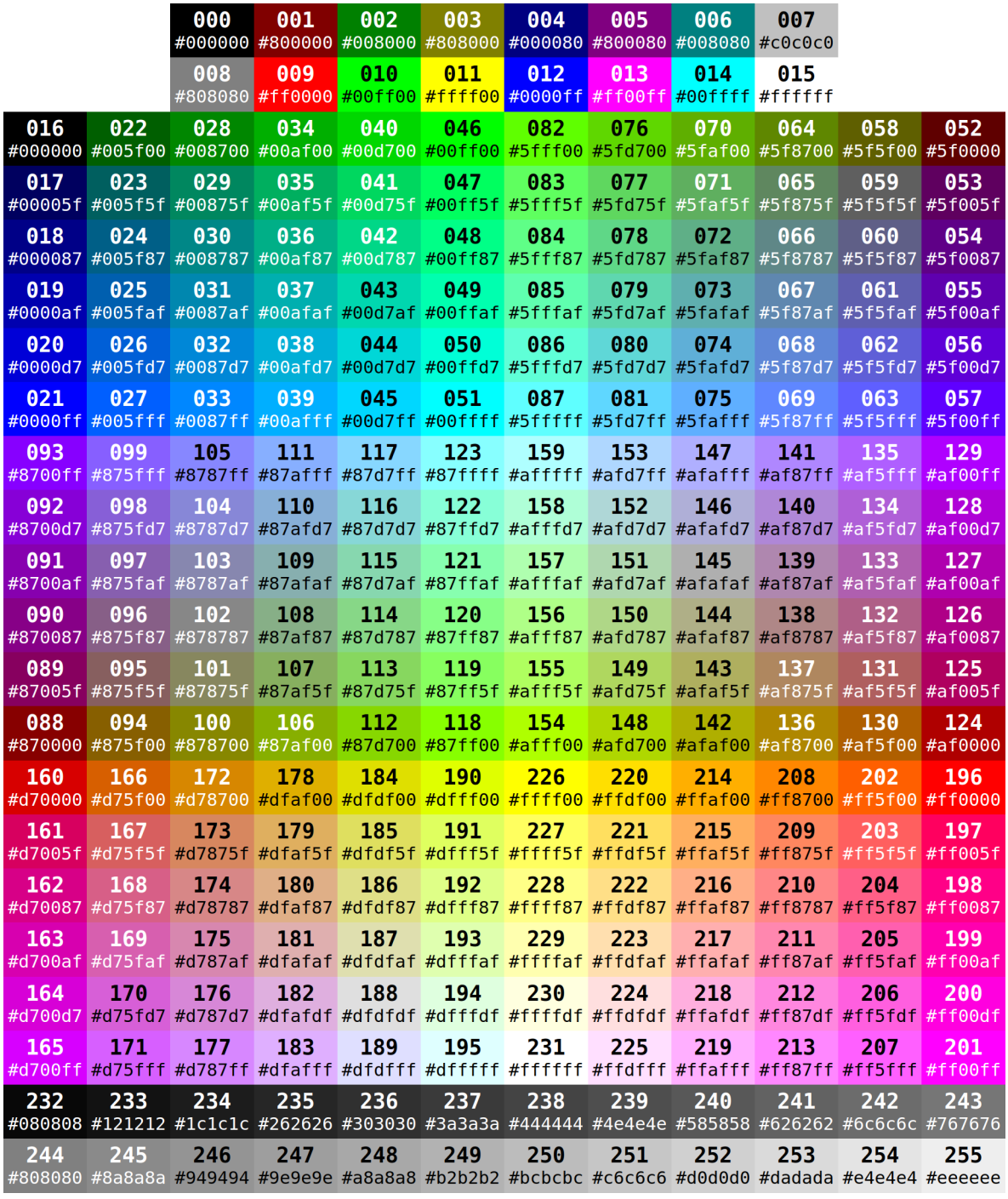


Fig. 2: *Indexed* mode palette

Sources

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

1.6 Formatters and Filters

The library contains @TODO

1.6.1 Auto-float formatter

1.6.2 Prefixed-unit formatter

1.6.3 Time delta formatter

```
from pytermor import Style, color, ColorRGB
from pytermor import renderer
from pytermor.renderer 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]

custom_stylesheet = time_delta.TimeDeltaStylesheet(
    default=Style(bg=0x202028),
    digit=Style(0x3333000, 'yellow'),
    unit=Style(fg='green', bg=0x202028, underlined=True),
    overflow=Style(fg=color.BLACK, bg='hi_red', bold=True),
)
for max_len in max_len_list:
    formatter = time_delta.registry.find_matching(max_len)
    formatter.stylesheet = custom_stylesheet

RendererManager.set_up(SGRRenderer)
for seconds in seconds_list:
    for max_len in max_len_list:
        formatter = time_delta.registry.get_by_max_len(max_len)
        print(formatter.format(seconds, True), end=' ')
    print()
```

1.6.4 String filters

1.6.5 Standard Library extensions

@TODO

CHAPTER

TWO

API REFERENCE

2.1 color

@TODO

@TODO black/white text selection depending on bg

class pytermor.color.**Color**

Abstract superclass for other `Colors`.

__init__(*hex_value: Optional[int] = None*)

static hex_value_to_hsv_channels(*hex_value: int*) → Tuple[int, float, float]

Transforms `hex_value` in `0xffffff` format into tuple of three numbers corresponding to *hue*, *saturation* and *value* channel values respectively. *Hue* is within [0, 359] range, *saturation* and *value* are within [0; 1] range.

static hex_value_to_rgb_channels(*hex_value: int*) → Tuple[int, int, int]

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

```
>>> Color.hex_value_to_rgb_channels(0x80ff80)
(128, 255, 128)
>>> Color.hex_value_to_rgb_channels(0x000001)
(0, 0, 1)
```

abstract classmethod get_default() → *Color*

Returns
Fallback instance of *Color* inheritor (if registries are empty).

abstract classmethod find_closest(*hex_value: int*) → *Color*

Wrapper for *Approximator.find_closest()*.

Parameters
hex_value – Integer color value in `0xffffff` format.

Returns
Nearest found color of specified type.

abstract to_sgr(*bg: bool = False*) → *SequenceSGR*

property hex_value: int | None

format_value(*prefix: str | None = '0x'*, *noop_str: str = '^'*) → str

class pytermor.color.**ColorDefault**

Bases: *Color*

__init__(*hex_value: int*, *code_fg: int*, *code_bg: int*)

classmethod get_default() → *ColorDefault*

Returns
Fallback instance of *Color* inheritor (if registries are empty).

classmethod find_closest(*hex_value: int*) → *ColorDefault*

Wrapper for *Approximator.find_closest()*.

Attention: Use this method only if you know what you are doing. *Default* mode colors may vary in a huge range depending on user terminal setup (colors even can have exactly the opposite value of what's listed in preset list). Much more reliable and predictable approach is to use *ColorIndexed.find_closest* instead.

Parameters
hex_value – Integer color value in `0xffffff` format.

Returns
Nearest found *ColorDefault* instance.

```
>>> ColorDefault.find_closest(0x660000)
ColorDefault[fg=31, bg=41, 0x800000]
```

to_sgr(*bg: bool = False*) → *SequenceSGR*

format_value(*prefix: str | None = '0x'*, *noop_str: str = '^'*) → str

property hex_value: int | None

static hex_value_to_hsv_channels(*hex_value: int*) → Tuple[int, float, float]

Transforms `hex_value` in `0xffffff` format into tuple of three numbers corresponding to *hue*, *saturation* and *value* channel values respectively. *Hue* is within [0, 359] range, *saturation* and *value* are within [0; 1] range.

static hex_value_to_rgb_channels(*hex_value: int*) → Tuple[int, int, int]

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

```
>>> Color.hex_value_to_rgb_channels(0x80ff80)
(128, 255, 128)
>>> Color.hex_value_to_rgb_channels(0x000001)
(0, 0, 1)
```

class pytermor.color.**ColorIndexed**

Bases: *Color*

__init__(*hex_value: int*, *code: int*)

classmethod get_default() → *ColorIndexed*

Returns
Fallback instance of *Color* inheritor (if registries are empty).

classmethod find_closest(*hex_value: int*) → *ColorIndexed*

Wrapper for *Approximator.find_closest()*.

Note: Approximation algorithm ignores colors 000-015 from the *indexed* palette and will return colors with int codes in 016-255 range only. The reason for this is the same as for discouraging the usage of *ColorDefault* method version – because aforementioned colors actually depend on end-user terminal settings and the final result can be differ drastically from what's the developer imagined.

Parameters
hex_value – Integer color value in `0xffffff` format.

Returns
Nearest found *ColorIndexed* instance.

```
>>> ColorIndexed.find_closest(0xd9dbdb)
ColorIndexed[code=253, 0xdadada]
```

to_sgr(*bg: bool = False*) → *SequenceSGR*

format_value(*prefix: str | None = '0x'*, *noop_str: str = '^'*) → str

property hex_value: int | None

static hex_value_to_hsv_channels(*hex_value: int*) → Tuple[int, float, float]

Transforms `hex_value` in `0xffffff` format into tuple of three numbers corresponding to *hue*, *saturation* and *value* channel values respectively. *Hue* is within [0, 359] range, *saturation* and *value* are within [0; 1] range.

static hex_value_to_rgb_channels(*hex_value: int*) → Tuple[int, int, int]

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

```
>>> Color.hex_value_to_rgb_channels(0x80ff80)
(128, 255, 128)
>>> Color.hex_value_to_rgb_channels(0x000001)
(0, 0, 1)
```

class pytermor.color.**ColorRGB**

Bases: *Color*

classmethod get_default() → *ColorRGB*

Returns
Fallback instance of *Color* inheritor (if registries are empty).

classmethod find_closest(*hex_value: int*) → *ColorRGB*

In case of *ColorRGB* we suppose that user's terminal is not limited to a palette, therefore RGB-type color map works by simplified algorithm – it just checks if instance with same hex value was already created and returns it if that's the case, or returns a brand new instance with specified color value otherwise.

Parameters
hex_value – Integer color value in `0xffffff` format.

Returns
Existing *ColorRGB* instance or newly created one.

```
>>> existing_color1 = ColorRGB(0x660000)
>>> existing_color2 = ColorRGB(0x660000)
>>> existing_color1 == existing_color2
True
>>> existing_color1 is existing_color2  # different instances
False
>>> existing_color1 == ColorRGB.find_closest(0x660000)
True
>>> existing_color1 is ColorRGB.find_closest(0x660000)  # same instance
True
```

to_sgr(*bg: bool = False*) → *SequenceSGR*

__init__(*hex_value: Optional[int] = None*)

format_value(*prefix: str | None = '0x'*, *noop_str: str = '^'*) → str

property hex_value: int | None

static hex_value_to_hsv_channels(*hex_value: int*) → Tuple[int, float, float]

Transforms `hex_value` in `0xffffff` format into tuple of three numbers corresponding to *hue*, *saturation* and *value* channel values respectively. *Hue* is within [0, 359] range, *saturation* and *value* are within [0; 1] range.

static hex_value_to_rgb_channels(*hex_value: int*) → Tuple[int, int, int]

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

```
>>> Color.hex_value_to_rgb_channels(0x80ff80)
(128, 255, 128)
>>> Color.hex_value_to_rgb_channels(0x000001)
(0, 0, 1)
```

pytermor.color.**TypeColor**

Any non-abstract *Color* type.

alias of TypeVar('TypeColor', ColorDefault, ColorIndexed, ColorRGB)

class pytermor.color.**Approximator**

Bases: `Generic`[*TypeColor*]

Class contains a dictionary of registred *Colors* indexed by hex code along with cached nearest color search results to avoid unnecessary instance copies and search repeating.

__init__(*parent_type: TypeColor*)

Called in *Color*-type class constructors. Each *Color* type should have class variable with instance of *Approximator* and create it by itself if it's not present.

Parameters
parent_type – Parent *Color* type.

add_to_map(*color: TypeColor*)

Called in *Color*-type class constructors. Add a new element in color lookup table if it wasn't there, and then drop cached search results as they are most probably useless after registering a new color (i.e. now there will be better result for at least one cached value).

Parameters
color – *Color* instance being created.

get_exact(*hex_value: int*) → Optional[TypeColor]

Public interface for searching exact values in the *lookup table*, or global registry of created instances of specified `Color` class.

Parameters
hex_value – Color value in RGB format.

Returns
`Color` with specified value. Type is equal to type of the parent of selected color map.

find_closest(*hex_value: int*) → TypeColor

Search for nearest to `hex_value` registered color. Is used by *SGRRenderer* to find supported color alternatives in case user's terminal is incapable of operating in better mode.

Parameters
hex_value – Color value in RGB format.

Returns
Nearest to `hex_value` registered `Color`. Type is equal to type of the parent of selected color map. If no colors of required type were created (table and cache are empty), invokes *get_default()* `Color` method.

approximate(*hex_value: int*, *max_results: int = 1*) → List[TypeColor]

Core color approximation method. Iterate the registred SGRs table, or *lookup table*, containing parents' instances, and compute the euclidean distance from argument to each color of the palette. Sort the results and return the first <`max_results`> of them.

Note: It's not guaranteed that this method will **always** succeed in searching (the result list can be empty). Consider using *find_closest* instead, if you really want to be sure that at least some color will be returned. Another option is to use special "color" named *NOOP*.

Parameters

- **hex_value** – Color RGB value.
- **max_results** – Maximum amount of values to return.

Returns
Closest *Color* instances found, sorted by color distance descending (i.e. 0th element is always the closest to the input value).

static __new__(*cls*, **args*, ***kwds*)

pytermor.color.**NOOP = ColorRGB[^]**

Special instance of *Color* class always rendering into empty string.

Attention: Preset constants are omitted from API doc pages to improve readability and avoid duplication. Summary list of all presets can be found in *Preset list* section of the guide.

2.2 intcode

Module with SGR param integer codes, contains a complete or almost complete list of reliably working ones.

Suitable for *Span* and *SequenceSGR* default constructors.

Attention: Preset constants are omitted from API doc pages to improve readability and avoid duplication. Summary list of all presets can be found in *Preset list* section of the guide.

2.3 renderer

Module with output formatters. By default *SGRRenderer* is used. It also contains compatibility settings, see *SGRRenderer.set_up()*.

Working with non-default renderer can be achieved in two ways:

a. Method *RendererManager.set_up()* sets the default renderer globally.

b. Alternatively, you can use renderer's own class method *Renderer.render()* directly and avoid calling *Style.render()* method whatsoever.

class pytermor.renderer.**RendererManager**

classmethod set_up(*default_renderer: Type[*Renderer*] | None = None*)

Set up renderer preferences. Affects all renderer types.

Parameters

default_renderer – Default renderer to use globally. Passing None will result in library default setting restored (*SGRRenderer*).

```
>>> RendererManager.set_up(DebugRenderer)
>>> Style('red').render('text')
'|31|text|39|'
```

```
>>> NoOpRenderer.render(Style('red'), 'text')
'text'
```

classmethod get_default() → Type[*Renderer*]

Get global default renderer type.

class pytermor.renderer.**Renderer**

Abstract ancestor of all renderers.

abstract classmethod render(*style:* Style, *text: str*) → str

Apply colors and attributes described in `style` argument to `text` and return the result. Output format depends on renderer's class (which defines the implementation).

class pytermor.renderer.**SGRRenderer**

Bases: *Renderer*

Default renderer that *Style.render()* invokes. Transforms *Color* instances defined in `style` argument into ANSI control sequence characters and merges them with input string.

classmethod set_up(*force_styles: bool | None = False*, *compatibility_indexed: bool = False*, *compatibility_default: bool = False*)

Set up renderer preferences. Affects all renderer types.

Parameters

- **force_styles** –
  - If set to *None*, all renderers will pass input text through themselves without any changes (i.e. no colors and attributes will be applied).
  - If set to *True*, renderers will always apply the formatting regardless of other internal rules and algorithms.
  - If set to *False* [default], the final decision will be made by every renderer independently, based on their own algorithms.
- **compatibility_indexed** – Disable *RGB* (or True Color) output mode. 256-color (*indexed*) sequences will be printed out instead of disabled ones. Useful when combined with `curses` – that way you can check the terminal capabilities from the inside of that terminal and switch to different output mode at once.
- **compatibility_default** – Disable *indexed* output mode and use *default* 16-color sequences instead. If this setting is set to *True*, the value of `compatibility_indexed` will be ignored completely. Useful when combined with `curses` – that way you can check the terminal capabilities from the inside of that terminal and switch to different output mode at once.

classmethod render(*style: Style*, *text: str*)

Render `text` with `style` applied as ANSI control sequences.

Respects compatibility preferences (see *RendererManager.set_up()*) and maps RGB colors to closest *indexed* colors if terminal doesn't support RGB output. In case terminal doesn't support even 256 colors, falls back to *default* colors, searching for closest counterparts in 16-color table.

Type of output `SequenceSGR` depends on type of *Color* variables in `style` argument. Keeping all that in mind, let's summarize:

1. *ColorRGB* can be rendered as True Color sequence, indexed sequence or default (16-color) sequence depending on terminal capabilities.
2. *ColorIndexed* can be rendered as indexed sequence or default sequence.
3. *ColorDefault* will be rendered as default-color sequence.

```
>>> SGRRenderer.render(Style('red', bold=True), 'text')
'\x1b[1;31mtext\x1b[22;39m'
```

Parameters

- **style** – Style to apply.
- **text** – Input string.

Returns

Input string enclosed in SGR sequences.

class pytermor.renderer.**TmuxRenderer**

Bases: *Renderer*

abstract classmethod render(*style:* Style, *text: str*) → str

Apply colors and attributes described in `style` argument to `text` and return the result. Output format depends on renderer's class (which defines the implementation).

class pytermor.renderer.**NoOpRenderer**

Bases: *Renderer*

classmethod render(*style:* Style, *text: str*) → str

Special renderer type that does nothing with the input string and just returns it as is.

```
>>> NoOpRenderer.render(Style('red', bold=True), 'text')
'text'
```

Parameters

- **style** – Style to ignore.
- **text** – Input string.

Returns

Input string without changes.

class pytermor.renderer.**HtmlRenderer**

Bases: *Renderer*

classmethod render(*style:* Style, *text: str*) → str

```
>>> HtmlRenderer.render(Style('red', bold=True), 'text')
'<span style="color: #800000; font-weight: bold">text</span>'
```

class pytermor.renderer.**DebugRenderer**

Bases: *Renderer*

classmethod render(*style:* Style, *text: str*) → str

```
>>> DebugRenderer.render(Style('red', bold=True), 'text')
'|1;31|text|22;39|'
```

2.4 sequence

Module contains definitions for low-level ANSI escape sequences handling.

Each preset defined below is a valid argument for *Span* and *SequenceSGR* default constructors (case-insensitive).

```
>>> Span(sequence.BG_GREEN, sequence.UNDERLINED)
Span[SGR[42;4], SGR[49;24]]
```

pytermor.sequence.**build**(**args: str | int |* SequenceSGR) → *SequenceSGR*

Create new *SequenceSGR* with specified `args` as params.

Resulting sequence param order is same as an argument order.

Each sequence param can be specified as:

- string key (see *span*)

- integer param value (from *intcode*)
- existing SequenceSGR instance (params will be extracted).

Deprecated since version 2.0: @TODO ,

```
>>> build('yellow', 'bold')
SGR[33;1]
>>> build(91, 7)
SGR[91;7]
>>> build(HI_CYAN, UNDERLINED)
SGR[96;4]
```

pytermor.sequence.**color_indexed**(*color: int*, *bg: bool = False*) → *SequenceSGR*

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

Deprecated since version 2.0: @TODO

Parameters

- **color** – Index of the color in the pallete, 0 – 255.
- **bg** – Set to *True* to change the background color (default is foreground).

Returns

SequenceSGR with required params.

pytermor.sequence.**color_rgb**(*r: int*, *g: int*, *b: int*, *bg: bool = False*) → *SequenceSGR*

Wrapper for creation of *SequenceSGR* operating in True Color mode (16M). Valid values for *r*, *g* and *b* are in range [0; 255]. This range linearly translates into [0x00; 0xFF] for each channel. The result value is composed as #RRGGBB. For example, sequence with color of #FF3300 can be created with:

```
color_rgb(255, 51, 0)
```

Deprecated since version 2.0: @TODO

Parameters

- **r** – Red channel value, 0 – 255.
- **g** – Blue channel value, 0 – 255.
- **b** – Green channel value, 0 – 255.
- **bg** – Set to *True* to change the background color (default is foreground).

Returns

SequenceSGR with required params.

class pytermor.sequence.**Sequence**

Abstract ancestor of all escape sequenes.

__init__(**params: int*)

abstract encode() → str

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

property params: List[int]

Return internal params as array.

class pytermor.sequence.**SequenceCSI**

Bases: *Sequence*

Abstract class representing CSI-type ANSI escape sequence. All subtypes of this sequence start with `e[`.

__init__(**params: int*)

classmethod regexp() → str

abstract encode() → str

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

property params: List[int]

Return internal params as array.

class pytermor.sequence.**SequenceSGR**

Bases: *SequenceCSI*

Class representing SGR-type escape sequence with varying amount of parameters.

SequenceSGR with zero params was specifically implemented to translate into empty string and not into `e[m`, which would have made sense, but also would be very entangling, as this sequence is equivalent of `e[0m` – hard reset sequence. The empty-string-sequence is predefined as *NOOP*.

It's possible to add of one SGR sequence to another:

```
>>> SequenceSGR(31) + SequenceSGR(1) == SequenceSGR(31, 1)
True
```

encode() → str

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

__init__(**params: int*)

property params: List[int]

Return internal params as array.

classmethod regexp() → str

pytermor.sequence.**NOOP = SGR[^]**

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

- `NOOP.encode()` returns empty string.
- `NOOP.params` returns empty list.

```
>>> NOOP.encode()
''
>>> NOOP.params
[]
```

New in version 1.8.

pytermor.sequence.**RESET = SGR[0]**

Resets all attributes and colors.

Attention: Preset constants are omitted from API doc pages to improve readability and avoid duplication. Summary list of all presets can be found in *Preset list* section of the guide.

2.5 span

Module introducing *Span* low-level abstractions. The key difference beetween them and Sequences is that sequence can *open* text style and also *close*, or terminate it. As for Spans – they always do both; typical use-case of *Span* is to wrap some text in opening SGR and closing one.

class pytermor.span.**Span**

__init__(**opening_params: str | int | SequenceSGR*)

Create a *Span* with specified control sequence(s) as an opening sequence and **automatically compose** second (closing) sequence that will terminate attributes defined in the first one while keeping the others (*soft* reset).

Resulting sequence param order is same as an argument order.

Each argument can be specified as:

- string key (name of any constant defined in *intcode*, case-insensitive)
- integer param value (defined in *intcode*)
- existing *SequenceSGR* instance (params will be extracted).

```
>>> Span('red', 'bold')
Span[SGR[31;1], SGR[39;22]]
>>> Span(intcode.GREEN)
Span[SGR[32], SGR[39]]
>>> Span(93, 4)
Span[SGR[93;4], SGR[39;24]]
>>> Span(sequence.BG_BLACK + sequence.RED)
Span[SGR[40;31], SGR[49;39]]
```

Parameters

opening_params – string keys, integer codes or existing SequenceSGR instances to build Span from.

classmethod from_seq(*opening_seq: Optional[SequenceSGR] = None*, *closing_seq: Optional[SequenceSGR] = None*, *hard_reset_after: bool = False*) → *Span*

Create new *Span* with explicitly specified opening and closing sequences.

Note: *closing_seq* gets overwritten with *sequence.RESET* if hard_reset_after is *True*.

Parameters

- **opening_seq** – Starter sequence, in general determening how *Span* will actually look like.
- **closing_seq** – Finisher SGR sequence.
- **hard_reset_after** – Terminate *all* formatting after this span.

wrap(*text: Optional[Any] = None*) → str

Wrap given text string with SGRs defined on initialization – *opening_seq* on the left, *closing_seq* on the right. str(text) will be invoked for all argument types with the exception of *None*, which will be replaced with an empty string.

Parameters
text – String to wrap.

Returns
`text` enclosed in instance's `SGRs`, if any.

property opening_str: str

Return opening SGR sequence encoded.

property opening_seq: *SequenceSGR*

Return opening SGR sequence instance.

property closing_str: str

Return closing SGR sequence encoded.

property closing_seq: *SequenceSGR*

Return closing SGR sequence instance.

__call__(*text: Optional[Any] = None*) → str

Can be used instead of *wrap()* method.

```
>>> RED('text') == RED.wrap('text')
True
```

pytermor.span.**NOOP = Span[SGR[^], SGR[^]]**

Special *Span* in cases where you *have to* select one or another *Span*, but do not want any control sequence to be actually included.

- `NOOP(string)` or `NOOP.wrap(string)` returns `string` without any modifications;
- `NOOP.opening_str` and `NOOP.closing_str` are empty strings;
- `NOOP.opening_seq` and `NOOP.closing_seq` both returns *sequence.NOOP*.

```
>>> NOOP('text')
'text'
>>> NOOP.opening_str
''
>>> NOOP.opening_seq
SGR[^]
```

Attention: Preset constants are omitted from API doc pages to improve readability and avoid duplication. Summary list of all presets can be found in *Preset list* section of the guide.

2.6 style

High-level abstraction defining text colors and attributes.

class pytermor.style.**Style**

Create a new `Style()`.

Key difference between `Styles` and `Spans` or `SGRs` is that `Styles` describe colors in RGB format and therefore support output rendering in several different formats (see *renderer*).

Both `fg` and `bg_color` can be specified as:

1. *Color* instance or library preset;
2. key code – name of any of aforementioned presets, case-insensitive;
3. integer color value in hexademical RGB format.

```
>>> Style('green', bold=True)
Style[fg=008000, no bg, bold]
>>> Style(bg=0x0000ff)
Style[no fg, bg=0000ff]
>>> Style(color.IDX_DEEP_SKY_BLUE_1, color.IDX_GREY_93)
Style[fg=00afff, bg=eeeeee]
```

Parameters

- **fg** – Foreground (i.e., text) color.
- **bg** – Background color.
- **auto_fg** – Automatically select `fg` based on `bg_color` (black or white depending on background brightness, if `bg_color` is defined).
- **blink** – Blinking effect; *supported by limited amount of Renderers*.
- **bold** – Bold or increased intensity.
- **crosslined** – Strikethrough.
- **dim** – Faint, decreased intensity.
- **double_underlined** – Faint, decreased intensity.
- **inversed** – Swap foreground and background colors.
- **italic** – Italic.
- **overlined** – Overline.
- **underlined** – Underline.

__init__(*fg: str | int | Color = None*, *bg: str | int | Color = None*, *auto_fg: bool = False*, *blink: bool = False*, *bold: bool = False*, *crosslined: bool = False*, *dim: bool = False*, *double_underlined: bool = False*, *inversed: bool = False*, *italic: bool = False*, *overlined: bool = False*, *underlined: bool = False*)

render(*text: Optional[Any] = None*) → str

Returns `text` with all attributes and colors applied.

By default uses *SequenceSGR* renderer, that means that output will contain ANSI escape sequences.

property fg: *Color*

property bg: *Color*

property blink: bool

property bold: bool

property crosslined: bool

property dim: bool

property double_underlined: bool

property inversed: bool

property italic: bool

property overlined

property underlined

class pytermor.style.**Stylesheet**

__init__(*default: Optional[Style] = None*)

2.7 util

Package containing a set of formatters for prettier output, as well as utility classes for removing some of the boilerplate code when dealing with escape sequences.

pytermor.util.**format_thousand_sep**(*value: int | float*, *separator=' '*)

Returns input `value` with integer part splitted 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'
```

2.7.1 auto_float

pytermor.util.auto_float.**format_auto_float**(*value: float*, *req_len: int*, *allow_exponent_notation: bool = True*) → str

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_auto_float(0.016789, 5)
'0.017'
>>> format_auto_float(0.167891, 5)
'0.168'
>>> format_auto_float(1.567891, 5)
'1.568'
>>> format_auto_float(12.56789, 5)
'12.57'
>>> format_auto_float(123.5678, 5)
'123.6'
>>> format_auto_float(1234.567, 5)
' 1235'
>>> format_auto_float(12345.67, 5)
'12346'
```

For cases when it's impossible to fit a number in the required length and rounding doesn't help (e.g. 12 500 000 and 5 chars) algorithm switches to scientific notation and the result looks like '1.2e7'.

When exponent form is disabled, there are two options for value that cannot fit into required length:

1) if absolute value is less than 1, zeros will be displayed ('0.0000');

2) in case of big numbers (like 10^9) ValueError will be raised instead.

Parameters

- **value** – Value to format
- **req_len** – Required output string length
- **allow_exponent_notation** – Enable/disable exponent form.

Returns
Formatted string of required length

Raises
ValueError –

New in version 1.7.

2.7.2 prefixed_unit

pytermor.util.prefixed_unit.**format_si_metric**(*value: float*, *unit: str = 'm'*) → str

Format `value` as meters with SI-prefixes, max result length is *6* chars. Base is *1000*. Unit can be customized.

```
>>> format_si_metric(123.456)
'1213 m'
>>> format_si_metric(0.331, 'g')
'331 mg'
>>> format_si_metric(45200, 'V')
'45.2 kV'
>>> format_si_metric(1.26e-9, 'm²')
'1.26 nm²'
```

Parameters

- **value** – Input value (unitless).
- **unit** – Value unit, printed right after the prefix.

Returns
Formatted string with SI-prefix if necessary.

New in version 2.0.

pytermor.util.prefixed_unit.**format_si_binary**(*value: float*, *unit: str = 'b'*) → str

Format `value` as binary size (bytes, kbytes, Mbytes), max result length is *8* chars. Base is *1024*. Unit can be customized.

```
>>> format_si_binary(631)
'631 b'
>>> format_si_binary(1080)
'1.055 kb'
>>> format_si_binary(45200)
'44.14 kb'
>>> format_si_binary(1.258 * pow(10, 6), 'bps')
'1.200 Mbps'
```

Parameters

- **value** – Input value in bytes.
- **unit** – Value unit, printed right after the prefix.

Returns
Formatted string with SI-prefix if necessary.

New in version 2.0.

class pytermor.util.prefixed_unit.**PrefixedUnitFormatter**

Formats `value` using settings passed to constructor. The main idea 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 to indicate them.

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

@TODO params desc

Parameters
prefix_zero_idx – Index of prefix which will be used as default, i.e. without multiplying coefficients.

New in version 1.7.

__init__(*max_value_len: int*, *integer_input: bool = False*, *unit: str = None*, *unit_separator: str = None*, *mcoef: float = 1000.0*, *prefixes: List[str | None] = None*, *prefix_zero_idx: int = None*)

property max_len: int

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

format(*value: float*, *unit: Optional[str] = None*) → str

Parameters

- **value** – Input value
- **unit** – Unit override

Returns
Formatted value

pytermor.util.prefixed_unit.**PREFIXES_SI = ['y', 'z', 'a', 'f', 'p', 'n', '', 'm', None, 'k', 'M', 'G', 'T', 'P', 'E', 'Z', 'Y']**

Prefix presets used by default module formatters. Can be useful if you are building your own formatter.

pytermor.util.prefixed_unit.**PREFIX_ZERO_SI = 8**

Index of prefix which will be used as default, i.e. without multiplying coefficients.

2.7.3 time_delta

Module for time difference formatting (e.g. “4 days 15 hours”, “8h 59m”).

Supports several output lengths and can be customized even more.

pytermor.util.time_delta.**format_time_delta**(*seconds: float*, *max_len: Optional[int] = None*) → str

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

Parameters

- **seconds** – Value to format
- **max_len** – Maximum output string length (total)

Returns

Formatted string

class pytermor.util.time_delta.**TimeDeltaFormatter**

Formatter 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.

You can create your own formatters if you need fine tuning of the output and customization. If that’s not the case, there is a facade method *format_time_delta()* which will select appropriate formatter automatically.

Example output:

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

__init__(*units: List[TimeUnit]*, *allow_negative: bool*, *unit_separator: Optional[str] = None*, *plural_suffix: Optional[str] = None*, *stylesheet: Optional[TimeDeltaStylesheet] = None*, *overflow_msg: str = 'OVERFLOW'*)

property stylesheet: *TimeDeltaStylesheet*

property max_len: int

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

Returns

Maximum possible output string length.

format(*seconds: float*, *always_max_len: bool = False*) → str

Format the requested amount of seconds and apply styles to the result as defined in current formatter's *stylesheet*. Default `stylesheet` contains no-op spans only and thus no styles will be applied.

Parameters

- **seconds** – Input value.
- **always_max_len** – If result string is less than *max_len* it will be returned as is, unless this flag is set to *True*. In that case output string will be paded with spaces on the left side so that resulting length would be always equal to maximum length.

Returns

Formatted string with applied styles if they are defined, raw string otherwise.

format_raw(*seconds: float*) → str | None

Format the requested amount of seconds as raw string without styling.

Parameters

seconds – Input value.

Returns

Formatted string or *None* on overflow (if input value is too big for the current formatter to handle).

class pytermor.util.time_delta.**TimeUnit**

TimeUnit(name: 'str', in_next: 'int' = None, custom_short: 'str' = None, collapsible_after: 'int' = None, overflow_afer: 'int' = None)

name: str

in_next: int = None

custom_short: str = None

collapsible_after: int = None

overflow_afer: int = None

__init__(*name: str*, *in_next: Optional[int] = None*, *custom_short: Optional[str] = None*, *collapsible_after: Optional[int] = None*, *overflow_afer: Optional[int] = None*) → None

class pytermor.util.time_delta.**TimeDeltaStylesheet**

Bases: *Stylesheet*

__init__(*default: Optional[Style] = None*, *digit: Optional[Style] = None*, *unit: Optional[Style] = None*, *overflow: Optional[Style] = None*)

2.7.4 stdlib_ext

Some of the Python Standard Library methods rewritten for correct work with strings containing control sequences.

pytermor.util.stdlib_ext.**ljust_sgr**(*s: str*, *width: int*, *fillchar: str = ' '*) → str

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).

pytermor.util.stdlib_ext.**rjust_sgr**(*s: str*, *width: int*, *fillchar: str = ' '*) → str

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).

pytermor.util.stdlib_ext.**center_sgr**(*s: str*, *width: int*, *fillchar: str = ' '*) → str

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-

2.7.5 string_filter

String filtering module.

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.

pytermor.util.string_filter.**apply_filters**(*s: AnyStr*, **args:* StringFilter*[AnyStr]* | *Type[*StringFilter*[AnyStr]]*) → AnyStr

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):

```
>>> apply_filters(span.RED('test'), ReplaceSGR(r'E\2\3\5'))
'E[31mtestE[39m'
```

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

Parameters

- **s** (*AnyStr*) – String to filter.
- **args** – *StringFilter* instance(s) or `StringFilter` class(es).

Returns

Filtered `s`.

class pytermor.util.string_filter.**StringFilter**

Bases: `Generic`

Common string modifier interface.

__init__(*pattern: AnyStr*, *repl: AnyStr | Callable[[AnyStr | Match], AnyStr]*)

__call__(*s: AnyStr*) → AnyStr

Can be used instead of *apply()*

apply(*s: AnyStr*) → AnyStr

Apply filter to `s` string (or bytes).

static __new__(*cls*, **args*, ***kwds*)

class pytermor.util.string_filter.**VisualuzeWhitespace**

Bases: *StringFilter*[str]

Replace every invisible character with `repl` (default is `·`), except newlines. Newlines are kept and get prepneded with same string.

```
>>> VisualuzeWhitespace().apply('A  B  C')
'A··B··C'
>>> apply_filters('1. D\n2. L ', VisualuzeWhitespace)
'1.·D·\n2.·L·'
```

__init__(*repl: AnyStr = '·'*)

__call__(*s: AnyStr*) → AnyStr

Can be used instead of *apply()*

static __new__(*cls*, **args*, ***kwds*)

apply(*s: AnyStr*) → AnyStr

Apply filter to `s` string (or bytes).

class pytermor.util.string_filter.**ReplaceSGR**

Bases: *StringFilter*[str]

Find all SGR seqs (e.g. `ESC[1;4m`) and replace with given string. More specific version of *ReplaceCSI*.

Parameters

repl – Replacement, can contain regexp groups (see *apply_filters()*).

__init__(*repl: AnyStr = ''*)

__call__(*s: AnyStr*) → AnyStr

Can be used instead of *apply()*

static __new__(*cls*, **args*, ***kwds*)

apply(*s: AnyStr*) → AnyStr

Apply filter to `s` string (or bytes).

class pytermor.util.string_filter.**ReplaceCSI**

Bases: *StringFilter*[str]

Find all CSI seqs (i.e. `ESC[*`) and replace with given string. Less specific version of *ReplaceSGR*, as CSI consists of SGR and many other sequence subtypes.

Parameters

repl – Replacement, can contain regexp groups (see *apply_filters()*).

__init__(*repl: AnyStr = ''*)

__call__(*s: AnyStr*) → AnyStr

Can be used instead of *apply()*

static __new__(*cls*, **args*, ***kwds*)

apply(*s: AnyStr*) → AnyStr

Apply filter to `s` string (or bytes).

class pytermor.util.string_filter.**ReplaceNonAsciiBytes**

Bases: *StringFilter*[bytes]

Keep 7-bit ASCII bytes [`0x00` - `0x7f`], replace other to `?`.

Parameters

repl – Replacement byte-string.

__init__(*repl: AnyStr = b'?'*)

__call__(*s: AnyStr*) → AnyStr

Can be used instead of *apply()*

static __new__(*cls*, **args*, ***kwds*)

apply(*s: AnyStr*) → AnyStr

Apply filter to `s` string (or bytes).

CHAPTER

THREE

CHANGELOG

3.1 v2.0.0

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

3.2 v1.8.0

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

3.3 v1.7.4

- Added 3 formatters: `format_prefixed_unit`, *format_time_delta*, *format_auto_float*.

3.4 v1.7.3

- Added `span.BG_BLACK` format.

3.5 v1.7.2

- Added *ljust_sgr*, *rjust_sgr*, *center_sgr* util functions to align strings with SGRs correctly.

3.6 v1.7.1

- Print reset sequence as `\e[m` instead of `\e[0m`.

3.7 v1.7.0

- *Span* constructor can be called without arguments.
- Added SGR code lists.

3.8 v1.6.2

- Excluded `tests` dir from distribution package.

3.9 v1.6.1

- Ridded of `EmptyFormat` and `AbstractFormat` classes.
- Renamed `code` module to `sgr` because of conflicts in PyCharm debugger (`pydevd_console_integration.py`).

3.10 v1.5.0

- Removed excessive `EmptySequenceSGR` – default `SGR` class was specifically implemented to print out as empty string instead of `\e[m` if constructed without params.

3.11 v1.4.0

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

3.12 v1.3.2

- Added `span.GRAY` and `span.BG_GRAY` format presets.

3.13 v1.3.1

- Interface revisioning.

3.14 v1.2.1

- *opening_seq* and *closing_seq* properties for *Span* class.

3.15 v1.2.0

- `EmptySequenceSGR` and `EmptyFormat` classes.

3.16 v1.1.0

- Autoformat feature.

3.17 v1.0.0

- First public version.

This project uses Semantic Versioning – https://semver.org *(starting from 2.0.0)*

CHAPTER

FOUR

LICENSE

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W