pytermor

Alexandr Shavykin

Jun 06, 2022

CONTENTS

(yet another) Python library designed for formatting terminal output using ANSI escape codes. Implements automatic "soft" format termination. Provides a registry of ready-to-use SGR sequences and formatting spans (or combined sequences). Also includes a set of number formatters for pretty output.

CHAPTER

ONE

MOTIVATION

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.

CHAPTER

TWO

INSTALLATION

```
pip install pytermor
```

CHAPTER

THREE

WHAT'S NEXT?

3.1 Use cases

```
from pytermor import span

print(span.blue('Use'), span.cyan('cases'))
```

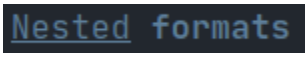
Span is a combination of two control sequences; it wraps specified string with pre-defined leading and trailing SGR definitions.

3.1.1 Nested formats

Preset spans can safely overlap with each other (as long as they require different *breaker* sequences to reset).

```
from pytermor import span

print(span.blue(span.underlined('Nested') + span.bold(' formats')))
```



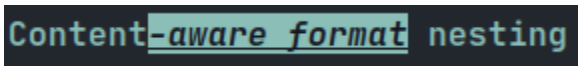
3.1.2 Content-aware nesting

Compose text spans with automatic content-aware format termination.

```
from pytermor import autocomplete

span1 = autocomplete('hi_cyan', 'bold')
span2 = autocomplete('bg_black', 'inversed', 'underlined', 'italic')

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



3.1.3 Flexible sequence builder

Create your own *SGR sequences* with `build()` method, which accepts color/attribute keys, integer codes and even existing *SGR*-s, in any amount and in any order. Key resolving is case-insensitive.

```
from pytermor import sequence, build

seq1 = build('red', 1)  # keys or integer codes
seq2 = build(seq1, sequence.ITALIC)  # existing SGRs as part of a new one
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)
```

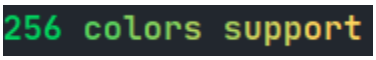
3.1.4 256 colors support

Use `color_indexed()` to set foreground/background color to any of xterm-256 colors.

```
from pytermor import color_indexed, sequence, autocomplete

txt = '256 colors support'
start_color = 41
msg = ''
for idx, c in enumerate(range(start_color, start_color+(36*6), 36)):
    msg += f'{color_indexed(c)}{txt[idx*3:(idx+1)*3]}{sequence.COLOR_OFF}'

print(autocomplete(sequence.BOLD).wrap(msg))
```



3.1.5 True Color support

Support for 16M-color mode (or True Color) — with `color_rgb()` wrapper method.

```
from pytermor import color_rgb, sequence, span

txt = 'True color support'
msg = ''
for idx, c in enumerate(range(0, 256, 256//18)):
    r = max(0, 255-c)
    g = max(0, min(255, 127-(c*2)))
    b = c
    msg += f'{color_rgb(r, g, b)}{txt[idx:(idx+1)]}{sequence.COLOR_OFF}'

print(span.bold(msg))
```

3.2 Core functions and classes

3.2.1 Functions

pytermor.**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 (see *intcode*)
- existing *SequenceSGR* instance (params will be extracted).

pytermor.**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.

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.**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)
```

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.

pytermor.**autocomplete**(**args: str | int | SequenceSGR*) → *Span*

Create new *Span* with specified control sequence(s) as an opening sequence and **automatically compose** 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 sequence param can be specified as:

- string key (see *span*)
- integer param value (see *intcode*)
- existing *SequenceSGR* instance (params will be extracted).

3.2.2 Classes

class pytermor.**SequenceSGR**(**params: int*)

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)
```

which is equivalent to:

```
SequenceSGR(31, 1)
```

print() → str

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

class pytermor.**Span**(*opening_seq: Optional[SequenceSGR] = None*, *closing_seq: Optional[SequenceSGR] = None*, *hard_reset_after: bool = False*)

Wrapper class that contains starter, or *opening* sequence and (optionally) *closing* sequence.

Note that *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** – Set *closing_seq* to a hard reset sequence.

property closing_seq: *SequenceSGR*

Return closing SGR sequence instance.

property closing_str: **str**

Return closing SGR sequence encoded.

property opening_seq: *SequenceSGR*

Return opening SGR sequence instance.

property opening_str: **str**

Return opening SGR sequence encoded.

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

Wrap given *text* with *Span*'s *SGRs* – *opening_seq* to the left, *closing_seq* to the right. `str(text)` will be invoked for all argument types with the exception of *None*, which will be replaced with empty string.

Parameters
text – String to wrap.

Returns
Resulting string; input argument enclosed to *Span*'s *SGRs*, if any.

3.3 Working with SGRs

3.3.1 Format soft reset

3.3.2 Creating and applying *SGRs*

3.3.3 SGR structure

3.3.4 Combining SGRs

3.3.5 Creating and applying *Spans*

3.4 Formatters and Filters

3.4.1 Auto-float formatter

3.4.2 Prefixed-unit formatter

3.4.3 Time delta formatter

3.4.4 *StringFilters*

3.4.5 Standard Library extensions

3.5 API docs

3.5.1 formatters

auto_float

pytermor.formatters.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.

Examples:

- `format_auto_float( 0.0167, 5)` => '0.017'
- `format_auto_float( 0.167, 5)` => '0.167'
- `format_auto_float( 1.567, 5)` => '1.567'
- `format_auto_float( 12.56, 5)` => '12.56'
- `format_auto_float( 123.56, 5)` => '123.6'
- `format_auto_float(1234.56, 5)` => ' 1235'
- `format_auto_float(12345.6, 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 is 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.

prefixed_unit

pytermor.formatters.prefixed_unit.**PRESET_SI_BINARY = PrefixedUnitPreset(max_value_len=5, integer_input=True, unit='b', unit_separator=' ', mcoef=1024.0, prefixes=['y', 'z', 'a', 'f', 'p', 'n', '', 'm', None, 'k', 'M', 'G', 'T', 'P', 'E', 'Z', 'Y'], prefix_zero_idx=8)**

Similar to PRESET_SI_METRIC, but this preset differs in one aspect. Given a variable with default value = *995*, printing it's value out using this preset results in "995 b". After increasing it by *20* we'll have *1015*, but it's still not enough to become a kilobyte – so displayed value will be "1015 b". Only after one more increasing (at *1024* and more) the value will be in a form of "1.00 kb".

So, in this case max_value_len must be at least **5** (not 4), because it's a minimum requirement for displaying values from *1023* to *-1023*.

Total maximum length is max_value_len + 3 = **8** (+3 is from separator, unit and prefix, assuming all of them have 1-char width).

pytermor.formatters.prefixed_unit.**PRESET_SI_METRIC = PrefixedUnitPreset(max_value_len=4, integer_input=False, unit='', unit_separator=' ', mcoef=1000.0, prefixes=['y', 'z', 'a', 'f', 'p', 'n', '', 'm', None, 'k', 'M', 'G', 'T', 'P', 'E', 'Z', 'Y'], prefix_zero_idx=8)**

Suitable for formatting any SI unit with values from approximately *10^-27* to *10^27*.

max_value_len must be at least **4**, because it's a minimum requirement for displaying values from *999* to *-999*. Next number to *999* is *1000*, which will be displayed as 1k.

Total maximum length is max_value_len + 3, which is **7** (+3 is from separator, unit and prefix, assuming all of them have 1-char width). Without unit (default) it's **6**.

class pytermor.formatters.prefixed_unit.**PrefixedUnitPreset**(*max_value_len: int*, *integer_input: bool*, *unit: str | None*, *unit_separator: str | None*, *mcoef: float*, *prefixes: List[str | None] | None*, *prefix_zero_idx: int | None*)

New in version 1.7.

integer_input: bool

property max_len: int

max_value_len: int

mcoef: float

prefix_zero_idx: int | None

prefixes: List[str | None] | None

unit: str | None

unit_separator: str | None

pytermor.formatters.prefixed_unit.**format_prefixed_unit**(*value: float*, *preset:* PrefixedUnitPreset) → str

Format `value` using `preset` settings. The main idea of this method is to fit into specified string length as much significant digits as it's theoretically possible, using multipliers and unit prefixes to indicate them.

Parameters

- **value** – Input value
- **preset** – Formatter settings

Returns
Formatted value

New in version 1.7.

pytermor.formatters.prefixed_unit.**format_si_binary**(*value: float*) → str

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

Examples:

- `format_si_binary(631)` => '631 b'
- `format_si_binary(1080)` => '1.05 kb'
- `format_si_binary(45200)` => '44.14 kb'
- `format_si_binary(1257800)` => '1.20 Mb'

Parameters
value – Input value in bytes.

Returns
Formatted string with SI-prefix if necessary.

New in version 2.0.

pytermor.formatters.prefixed_unit.**format_si_metric**(*value: float*) → str

Format `value` as unitless value with SI-prefixes, max result length is *6* chars. Base is *1000*.

Examples:

- `format_si_metric(123.456)` => '123'
- `format_si_metric(1080)` => '1.08 k'
- `format_si_metric(45200)` => '45.2 k'

- `format_si_metric(1257800)` => '1.26 M'

Parameters
value – Input value (unitless).

Returns
Formatted string with SI-prefix if necessary.

New in version 2.0.

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.

class pytermor.formatters.time_delta.**TimeDeltaFormatter**(*max_len: int*, *units: List[TimeUnit]*, *allow_negative: bool*, *unit_separator: str | None = None*, *plural_suffix: str | None = None*, *overflow_msg: str | None = 'OVERFLOW'*)

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

Example output:
"10 secs", "5 mins", "4h 15min", "5d 22h"

allow_negative: bool

format(*seconds: float*) → str

max_len: int

overflow_msg: str | None = 'OVERFLOW'

plural_suffix: str | None = None

unit_separator: str | None = None

units: List[*TimeUnit*]

class pytermor.formatters.time_delta.**TimeDeltaFormatterRegistry**

Simple registry for storing formatters and selecting the suitable one by max output length.

find_matching(*max_len: int*) → *TimeDeltaFormatter* | None

get_by_max_len(*max_len: int*) → *TimeDeltaFormatter* | None

get_longest() → *TimeDeltaFormatterRegistry* | None

get_shortest() → *TimeDeltaFormatterRegistry* | None

register(**formatters:* TimeDeltaFormatter)

class pytermor.formatters.time_delta.**TimeUnit**(*name: 'str'*, *in_next: 'int' = None*, *custom_short: 'str' = None*, *collapsible_after: 'int' = None*, *overflow_afer: 'int' = None*)

collapsible_after: int = None

custom_short: str = None

in_next: int = None

name: str

overflow_afer: int = None

pytermor.formatters.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. print "3h 48min" instead of "228 mins" or "3 hours",

Formatters are defined for max_len= 3, 4, 6 and 10. Therefore, you can pass in any value from 3 incl. and it's guarenteed that result's length will be less or equal to required length. If omitted longest registred will be used.

Example output:

- max_len=3: "10s", "5m", "4h", "5d"
- max_len=4: "10 s", "5 m", "4 h", "5 d"
- max_len=6: "10 sec", "5 min", "4h 15m", "5d 22h"
- max_len=10: "10 secs", "5 mins", "4h 15min", "5d 22h"

Parameters

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

Returns

Formatted string

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

A :param value: :param separator: :return:

3.5.2 intcode

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

Suitable for *autocomplete()* and *build()* library methods.

pytermor.intcode.**RESET = 0**

Hard reset code.

pytermor.intcode.**BOLD = 1**

pytermor.intcode.**DIM = 2**

pytermor.intcode.**ITALIC = 3**

pytermor.intcode.**UNDERLINED = 4**

pytermor.intcode.**BLINK_SLOW = 5**

pytermor.intcode.**BLINK_FAST = 6**

pytermor.intcode.**INVERSED = 7**

pytermor.intcode.**HIDDEN = 8**

pytermor.intcode.**CROSSLINED = 9**

pytermor.intcode.**DOUBLE_UNDERLINED = 21**

pytermor.intcode.**OVERLINED = 53**

pytermor.intcode.**NO_BOLD_DIM = 22**

> **Note:** There is no separate sequence for disabling either *BOLD* or *DIM* while keeping the other.

pytermor.intcode.**ITALIC_OFF = 23**

pytermor.intcode.**UNDERLINED_OFF = 24**

pytermor.intcode.**BLINK_OFF = 25**

pytermor.intcode.**INVERSED_OFF = 27**

pytermor.intcode.**HIDDEN_OFF = 28**

pytermor.intcode.**CROSSLINED_OFF = 29**

pytermor.intcode.**COLOR_OFF = 39**

pytermor.intcode.**BG_COLOR_OFF = 49**

pytermor.intcode.**OVERLINED_OFF = 55**

pytermor.intcode.**BLACK = 30**

pytermor.intcode.**RED = 31**

pytermor.intcode.**GREEN = 32**

pytermor.intcode.**YELLOW = 33**

pytermor.intcode.**BLUE = 34**

pytermor.intcode.**MAGENTA = 35**

pytermor.intcode.**CYAN = 36**

pytermor.intcode.**WHITE = 37**

pytermor.intcode.**BG_BLACK = 40**

pytermor.intcode.**BG_RED = 41**

pytermor.intcode.**BG_GREEN = 42**

pytermor.intcode.**BG_YELLOW = 43**

pytermor.intcode.**BG_BLUE = 44**

pytermor.intcode.**BG_MAGENTA = 45**

pytermor.intcode.**BG_CYAN = 46**

pytermor.intcode.**BG_WHITE = 47**

pytermor.intcode.**GRAY = 90**

pytermor.intcode.**HI_RED = 91**

pytermor.intcode.**HI_GREEN = 92**

pytermor.intcode.**HI_YELLOW = 93**

pytermor.intcode.**HI_BLUE = 94**

pytermor.intcode.**HI_MAGENTA = 95**

pytermor.intcode.**HI_CYAN = 96**

pytermor.intcode.**HI_WHITE = 97**

pytermor.intcode.**BG_GRAY = 100**

pytermor.intcode.**BG_HI_RED = 101**

pytermor.intcode.**BG_HI_GREEN = 102**

pytermor.intcode.**BG_HI_YELLOW = 103**

pytermor.intcode.**BG_HI_BLUE = 104**

pytermor.intcode.**BG_HI_MAGENTA = 105**

pytermor.intcode.**BG_HI_CYAN = 106**

pytermor.intcode.**BG_HI_WHITE = 107**

pytermor.intcode.**LIST_COLORS = [30, 31, 32, 33, 34, 35, 36, 37, 38]**

pytermor.intcode.**LIST_BG_COLORS = [40, 41, 42, 43, 44, 45, 46, 47, 48]**

pytermor.intcode.**LIST_HI_COLORS = [90, 91, 92, 93, 94, 95, 96, 97]**

pytermor.intcode.**LIST_BG_HI_COLORS = [100, 101, 102, 103, 104, 105, 106, 107]**

pytermor.intcode.**LIST_ALL_COLORS = [30, 31, 32, 33, 34, 35, 36, 37, 38, 40, 41, 42, 43, 44, 45, 46, 47, 48, 90, 91, 92, 93, 94, 95, 96, 97, 100, 101, 102, 103, 104, 105, 106, 107]**

3.5.3 registry

class pytermor.registry.**Registry**

get_closing_seq(*opening_seq:* SequenceSGR) → *SequenceSGR*

register_complex(*starter_codes: Tuple[int, ...]*, *param_len: int*, *breaker_code: int*)

register_single(*starter_code: int | Tuple[int, ...]*, *breaker_code: int*)

3.5.4 sequence

Module contains definitions for working with ANSI escape sequences as classes and instances.

Each preset defined below is a valid argument for *autocomplete()* and *build()* methods.

class pytermor.sequence.**SequenceSGR**(**params: int*)

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)
```

which is equivalent to:

```
SequenceSGR(31, 1)
```

print() → str

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

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 (see *intcode*)
- existing *SequenceSGR* instance (params will be extracted).

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.

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)
```

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.

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

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

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

New in version 1.8.

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

Reset all attributes and colors.

pytermor.sequence.**BOLD = SGR[1]**

pytermor.sequence.**DIM = SGR[2]**

pytermor.sequence.**ITALIC = SGR[3]**

pytermor.sequence.**UNDERLINED = SGR[4]**

pytermor.sequence.**BLINK_SLOW = SGR[5]**

pytermor.sequence.**BLINK_FAST = SGR[6]**

pytermor.sequence.**INVERSED = SGR[7]**

pytermor.sequence.**HIDDEN = SGR[8]**

pytermor.sequence.**CROSSLINED = SGR[9]**

pytermor.sequence.**DOUBLE_UNDERLINED = SGR[21]**

pytermor.sequence.**OVERLINED = SGR[53]**

pytermor.sequence.**NO_BOLD_DIM = SGR[22]**

pytermor.sequence.**ITALIC_OFF = SGR[23]**

pytermor.sequence.**UNDERLINED_OFF = SGR[24]**

pytermor.sequence.**BLINK_OFF = SGR[25]**

pytermor.sequence.**INVERSED_OFF = SGR[27]**

pytermor.sequence.**HIDDEN_OFF = SGR[28]**

pytermor.sequence.**CROSSLINED_OFF = SGR[29]**

pytermor.sequence.**OVERLINED_OFF = SGR[55]**

pytermor.sequence.**BLACK = SGR[30]**

pytermor.sequence.**RED = SGR[31]**

pytermor.sequence.**GREEN = SGR[32]**

pytermor.sequence.**YELLOW = SGR[33]**

pytermor.sequence.**BLUE = SGR[34]**

pytermor.sequence.**MAGENTA = SGR[35]**

pytermor.sequence.**CYAN = SGR[36]**

pytermor.sequence.**WHITE = SGR[37]**

pytermor.sequence.**COLOR_OFF = SGR[39]**

pytermor.sequence.**BG_BLACK = SGR[40]**

pytermor.sequence.**BG_RED = SGR[41]**

pytermor.sequence.**BG_GREEN = SGR[42]**

pytermor.sequence.**BG_YELLOW = SGR[43]**

pytermor.sequence.**BG_BLUE = SGR[44]**

pytermor.sequence.**BG_MAGENTA = SGR[45]**

pytermor.sequence.**BG_CYAN = SGR[46]**

pytermor.sequence.**BG_WHITE = SGR[47]**

pytermor.sequence.**BG_COLOR_OFF = SGR[49]**

pytermor.sequence.**GRAY = SGR[90]**

pytermor.sequence.**HI_RED = SGR[91]**

pytermor.sequence.**HI_GREEN = SGR[92]**

pytermor.sequence.**HI_YELLOW = SGR[93]**

pytermor.sequence.**HI_BLUE = SGR[94]**

pytermor.sequence.**HI_MAGENTA = SGR[95]**

pytermor.sequence.**HI_CYAN = SGR[96]**

pytermor.sequence.**HI_WHITE = SGR[97]**

pytermor.sequence.**BG_GRAY = SGR[100]**

pytermor.sequence.**BG_HI_RED = SGR[101]**

pytermor.sequence.**BG_HI_GREEN = SGR[102]**

pytermor.sequence.**BG_HI_YELLOW = SGR[103]**

pytermor.sequence.**BG_HI_BLUE = SGR[104]**

pytermor.sequence.**BG_HI_MAGENTA = SGR[105]**

pytermor.sequence.**BG_HI_CYAN = SGR[106]**

pytermor.sequence.**BG_HI_WHITE = SGR[107]**

3.5.5 span

Module introducing *Span* 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.

Name of any format preset in this module can be used as a string argument in *build()* and *autocomplete()* methods:

```
autocomplete('red', 'bold')
```

class pytermor.span.**Span**(*opening_seq: Optional[SequenceSGR] = None*, *closing_seq: Optional[SequenceSGR] = None*, *hard_reset_after: bool = False*)

Wrapper class that contains starter, or *opening* sequence and (optionally) *closing* sequence.

Note that *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** – Set *closing_seq* to a hard reset sequence.

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

Wrap given *text* with *Span*'s *SGRs* – *opening_seq* to the left, *closing_seq* to the right. `str(text)` will be invoked for all argument types with the exception of *None*, which will be replaced with empty string.

Parameters
text – String to wrap.

Returns
Resulting string; input argument enclosed to *Span*'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.

pytermor.span.**autocomplete**(**args: str | int | SequenceSGR*) → *Span*

Create new *Span* with specified control sequence(s) as an opening sequence and **automatically compose** 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 sequence param can be specified as:

- string key (see *span*)
- integer param value (see *intcode*)
- existing *SequenceSGR* instance (params will be extracted).

pytermor.span.**noop = Span[SGR[], SGR[]]**

Special *Span* in cases where you *have to* select one or another *Span*, but do not want anything to be actually printed.

- `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`.

pytermor.span.**bold = Span[SGR[1], SGR[22]]**

pytermor.span.**dim = Span[SGR[2], SGR[22]]**

pytermor.span.**italic = Span[SGR[3], SGR[23]]**

pytermor.span.**underlined = Span[SGR[4], SGR[24]]**

pytermor.span.**inversed = Span[SGR[7], SGR[27]]**

pytermor.span.**overlined = Span[SGR[53], SGR[55]]**

pytermor.span.**red = Span[SGR[31], SGR[39]]**

pytermor.span.**green = Span[SGR[32], SGR[39]]**

pytermor.span.**yellow = Span[SGR[33], SGR[39]]**

pytermor.span.**blue = Span[SGR[34], SGR[39]]**

pytermor.span.**magenta = Span[SGR[35], SGR[39]]**

pytermor.span.**cyan = Span[SGR[36], SGR[39]]**

pytermor.span.**gray = Span[SGR[90], SGR[39]]**

pytermor.span.**bg_black = Span[SGR[40], SGR[49]]**

pytermor.span.**bg_red = Span[SGR[41], SGR[49]]**

pytermor.span.**bg_green = Span[SGR[42], SGR[49]]**

pytermor.span.**bg_yellow = Span[SGR[43], SGR[49]]**

pytermor.span.**bg_blue = Span[SGR[44], SGR[49]]**

pytermor.span.**bg_magenta = Span[SGR[45], SGR[49]]**

pytermor.span.**bg_cyan = Span[SGR[46], SGR[49]]**

pytermor.span.**bg_gray = Span[SGR[100], SGR[49]]**

3.5.6 util

stdlib_ext

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

pytermor.util.stdlib_ext.**center_aware**(*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).

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_aware**(*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).

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**(*string: AnyStr*, **args:* StringFilter*[AnyStr]* | *Type[*StringFilter*[AnyStr]]*) → AnyStr

Method for applying dynamic filter list to a target str/bytes. Example (will replace all *\x1b* control characters to *E* and make SGR params visible):

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

Note that type of `string` argument must correspond to *StringFilter's* types, i.e. `ReplaceNonAsciiBytes` is *StringFilter[bytes]* type, so you can apply it only to bytes-type strings.

Parameters

- **string** (*AnyStr*) – String for filter application (str or bytes-type).
- **args** – *StringFilter* instances or *StringFilter* types.

Returns

String with applied filters.

class pytermor.util.string_filter.**StringFilter**(*repl: AnyStr*)

Common string modifier interface.

abstract apply(*s: AnyStr*) → AnyStr

class pytermor.util.string_filter.**ReplaceSGR**(*repl: AnyStr = ''*)

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*).

apply(*s: str*) → str

class pytermor.util.string_filter.**ReplaceCSI**(*repl: AnyStr = ''*)

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*).

apply(*s: str*) → str

class pytermor.util.string_filter.**ReplaceNonAsciiBytes**(*repl: AnyStr = b''*)

Keep 7-bit ASCII bytes *[0x00 - 0x7f]*, replace other to '?' (by default).

Parameters
repl – Replacement bytes. To delete non-ASCII bytes define it as b".

apply(*s: bytes*) → bytes

Utility package for removing some of the boilerplate code when dealing with escape sequences.

modindex | genindex

PYTHON MODULE INDEX

p

INDEX

D

F

G

H

I

L

M

N

O

P

R

S

T

U

W

Y