Blessed Documentation

Release 1.53.0

Erik Rose, Jeff Quast, Avram Lubkin

Nov 20, 2026

CONTENTS

1 Introduction **3**
- 1.1 Examples . . . 3
- 1.2 Requirements . . . 3
- 1.3 Brief Overview . . . 5
- 1.4 Before And After . . . 5

2 Terminal **7**
- 2.1 Capabilities . . . 7
- 2.2 Capability Resolution Order . . . 8
- 2.3 Compound Formatting . . . 8
- 2.4 Clearing The Screen . . . 8
- 2.5 Hyperlinks . . . 8
- 2.6 Window Title . . . 9
- 2.7 Progress Bar . . . 9
- 2.8 Sixel Graphics Support . . . 10
- 2.9 Font Glyph Coverage . . . 11
- 2.10 OSC 52 Clipboard . . . 12
  - 2.10.1 Detection . . . 12
  - 2.10.2 Copying to Clipboard . . . 12
  - 2.10.3 Reading from Clipboard . . . 12
- 2.11 Styled and Colored Underlines . . . 13
- 2.12 Color Scheme Detection . . . 14
- 2.13 Kitty Query Extensions . . . 14
- 2.14 DECRQSS Support . . . 14
- 2.15 Terminal Software Version . . . 14
- 2.16 Styles . . . 15
- 2.17 Full-Screen Mode . . . 16
- 2.18 Line Wrap Control . . . 17
- 2.19 Pipe Savvy . . . 17
- 2.20 Singleton-Free . . . 17

3 Colors **19**
- 3.1 24-bit Colors . . . 19
- 3.2 Hex Colors . . . 20
- 3.3 Querying Terminal Colors . . . 20
- 3.4 256 Colors . . . 21
- 3.5 16 Colors . . . 21
- 3.6 Monochrome . . . 22
- 3.7 Color chart . . . 22

4 Keyboard Input **39**
4.1 Overview . . . 39
4.2 Getting Started . . . 39
4.3 Keystroke . . . 40
4.3.1 Special Keys . . . 40
4.3.2 Modifiers . . . 41
4.3.3 Event Types . . . 42
4.3.4 Magic Methods . . . 42
4.4 All Names . . . 43
4.5 Async Input . . . 47

5 Kitty Keyboard Protocol **49**
5.1 Overview . . . 49
5.2 Getting Started . . . 49
5.3 Event Types . . . 50
5.4 Protocol Features . . . 51
5.5 Key Name Synthesis . . . 51
5.5.1 Shifted Key Reporting . . . 52
5.5.2 Layout-Independent Key Handling . . . 52
5.6 Compatibility . . . 52
5.7 Timeout . . . 53
5.8 See Also . . . 53

6 Line Editor **55**
6.1 Overview . . . 55
6.2 Example . . . 55
6.3 History . . . 57
6.4 Display State & Styling . . . 57
6.5 Rendering Helpers . . . 57
6.6 Other Methods . . . 58
6.7 Constructor Options . . . 58
6.8 Custom Keybindings . . . 58
6.9 Default Keybindings . . . 59

7 Mouse Input **61**
7.1 Getting Started . . . 61
7.2 Understanding Buttons . . . 62
7.3 Understanding Mouse Coordinates . . . 63
7.4 Checking Support . . . 63
7.5 Using mouse_enabled() . . . 64
7.5.1 report_drag . . . 64
7.5.2 report_motion . . . 65
7.5.3 report_pixels . . . 66
7.6 Legacy Mouse Protocols . . . 66

8 Cursor Shapes **67**
8.1 Available Shapes . . . 67
8.2 Usage . . . 67
8.3 Direct Sequences . . . 68
8.4 Example . . . 68

9 DEC Private Modes **69**
9.1 Overview . . . 69
9.2 Timeout Behavior . . . 69
9.3 Getting Started . . . 70

9.4 Querying Mode Support . . . 70
9.5 Quick Boolean Detection . . . 71
9.6 Context Managers . . . 71
9.7 Synchronized Output . . . 72
9.8 Bracketed Paste . . . 73
9.9 Focus Events . . . 73

10 Location **75**
10.1 Example . . . 76
10.2 Context Manager . . . 76
10.3 Finding The Cursor . . . 77
10.4 CPR_RESPONSE . . . 77
10.5 Scroll Region . . . 78

11 Sizing & Alignment **81**
11.1 Length . . . 81
11.2 Alignment . . . 82
11.3 Terminal Corrections . . . 83
11.4 Text Sizing . . . 83
11.5 Detecting Resize . . . 85
11.6 Using SIGWINCH . . . 85
11.7 Using notify_on_resize . . . 86
11.8 Using both . . . 86

12 Sequences **89**
12.1 Splitting Sequences . . . 89
12.2 Stripping Sequences . . . 89

13 Sixel Graphics **91**
13.1 Checking Sixel Support . . . 91
13.2 Window Dimensions . . . 91
13.3 Cell Dimensions . . . 92
13.4 Colors . . . 92
13.5 Caching . . . 92
13.6 Complete Workflow . . . 93

14 Examples **95**
14.1 bounce.py . . . 95
14.2 cellestial.py . . . 95
14.3 cnn.py . . . 95
14.4 detect-multibyte.py . . . 95
14.5 editor.py . . . 96
14.6 keymatrix.py . . . 96
14.7 on_resize.py . . . 96
14.8 plasma.py . . . 96
14.9 progress_bar.py . . . 96
14.10 resize.py . . . 96
14.11 tprint.py . . . 96
14.12 worms.py . . . 97
14.13 x11_colorpicker.py . . . 97
14.14 display-modes.py . . . 97
14.15 display-unicode.py . . . 97
14.16 strip.py . . . 97
14.17 mouse_paint.py . . . 97
14.18 scroll_region.py . . . 97

14.19 text_sizing_protocol.py . . . 98
14.20 detect-tofus.py . . . 98
14.21 scroller.py . . . 98

15 API Documentation **101**
15.1 color.py . . . 101
15.2 colorspace.py . . . 104
15.3 dec_modes.py . . . 108
15.4 formatters.py . . . 116
15.5 keyboard.py . . . 122
15.6 line_editor.py . . . 135
15.7 mouse.py . . . 138
15.8 sequences.py . . . 140
15.9 terminal.py . . . 144

16 Project **189**
16.1 Fork . . . 189
16.2 License . . . 189
16.3 Running Tests . . . 189
16.4 Formatting . . . 189
16.5 Linting . . . 190
16.6 Further Reading . . . 190
16.7 The misnomer of ANSI . . . 191

17 Version History **193**

18 Indexes **205**

Python Module Index **207**

Index **209**

CHAPTER

ONE

INTRODUCTION

Blessed is an easy, practical *library* for making *terminal* apps, by providing an elegant, well-documented interface to Colors, Keyboard input, and screen position and Location capabilities.

```
from blessed import Terminal

term = Terminal()

print(term.home + term.clear + term.move_y(term.height // 2))
print(term.black_on_darkkhaki(term.center('press any key to continue.')))

with term.cbreak(), term.hidden_cursor():
    inp = term.inkey()

print(term.move_down(2) + 'You pressed ' + term.bold(repr(inp)))
```

It's meant to be *fun* and *easy*, to do basic terminal graphics and styling with Python using *blessed*. Terminal is the only class you need to import and the only object you should need for Terminal capabilities.

Whether you want to improve CLI apps with colors, or make fullscreen applications or games, *blessed* should help get you started quickly. Your users will love it because it works on Windows, Mac, and Linux, and you will love it because it has plenty of documentation and examples!

Full documentation at https://blessed.readthedocs.io/en/latest/

1.1 Examples

Fig. 1: x11-colorpicker.py, bounce.py, worms.py, and plasma.py demonstration programs from our repository.

Fig. 2: cellestial.py demonstration program from our repository.

Exemplary 3rd-party examples which use *blessed*,

1.2 Requirements

Blessed works with Windows, Mac, Linux, and BSD's, on Python 3.9+.

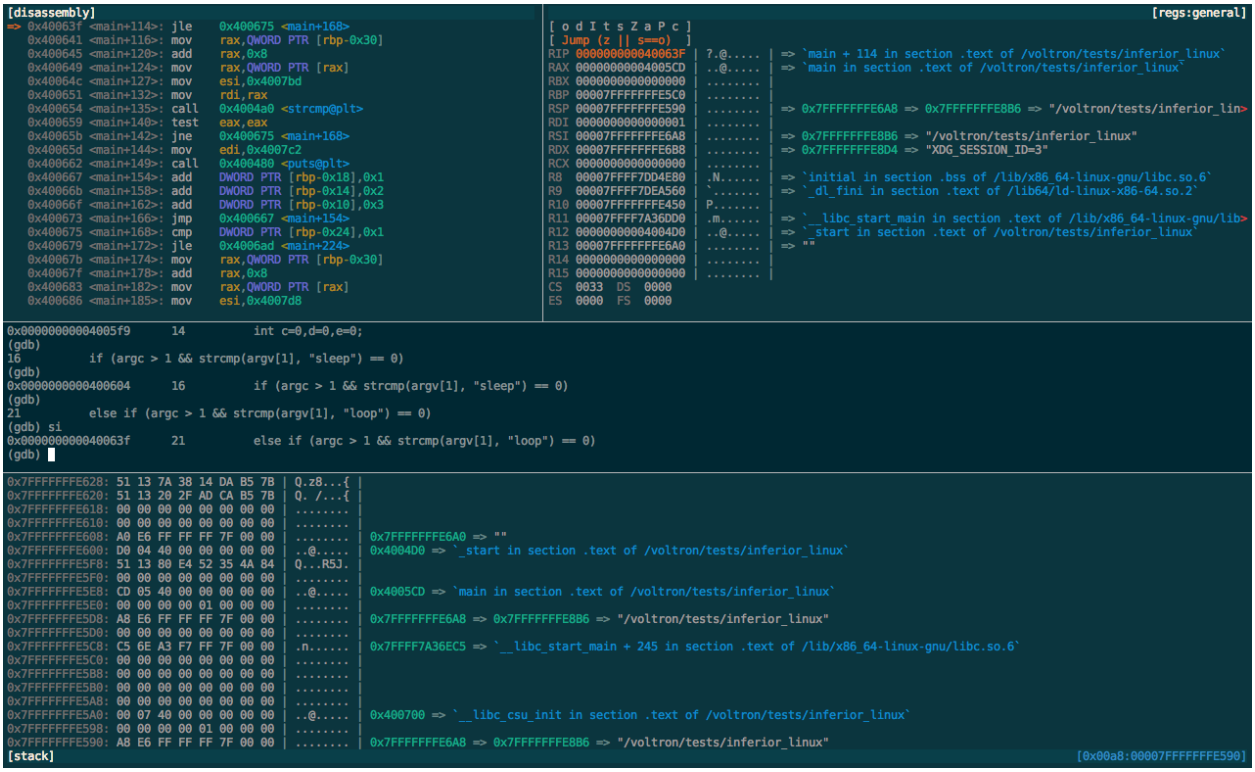


Fig. 3: Voltron is an extensible debugger UI toolkit written in Python

Fig. 4: cursewords is “graphical” command line program for solving crossword puzzles in the terminal.

Fig. 5: Dashing is a library to quickly create terminal-based dashboards.

Fig. 6: Enlighten is a console progress bar library that allows simultaneous output without redirection.

1.3 Brief Overview

Blessed is more than just a Python wrapper around curses:

- Styles, Colors, and maybe a little positioning without necessarily clearing the whole screen first.
- Works great with Python's new f-strings or any other kind of string formatting.
- Provides up-to-the-moment Location and terminal height and width, so you can respond to terminal size changes.
- Avoids making a mess if the output gets piped to a non-terminal, you can output sequences to any file-like object such as *StringIO*, files, pipes or sockets.
- Uses terminfo(5) so it works with any terminal type and capability: No more C-like calls to tigetstr and tparm.
- Non-obtrusive calls to only the capabilities database ensures that you are free to mix and match with calls to any other curses application code or library you like.
- Provides context managers Terminal.fullscreen() and Terminal.hidden_cursor() to safely express terminal modes, curses development will no longer fudge up your shell.
- Act intelligently when somebody redirects your output to a file, omitting all of the special sequences colors, but still containing all of the text.

Blessed is a fork of blessings, which does all of the same above with the same API, as well as following **enhancements**:

- Windows support, new since Dec. 2019!
- Dead-simple keyboard handling: safely decoding unicode input in your system's preferred locale and supports application/arrow keys.
- 24-bit color support, using Terminal.color_rgb() and Terminal.on_color_rgb() and all X11 Colors by name, and not by number.
- Determine cursor location using Terminal.get_location(), enter key-at-a-time input mode using Terminal.cbreak() or Terminal.raw() context managers, and read timed key presses using Terminal.inkey().
- Allows the *printable length* of strings that contain sequences to be determined by Terminal.length(), supporting additional methods Terminal.wrap() and Terminal.center(), terminal-aware variants of the built-in function textwrap.wrap() and method str.center(), respectively.
- Allows sequences to be removed from strings that contain them, using Terminal.strip_seqs() or sequences and whitespace using Terminal.strip().
- Per-character text sizing using the kitty text sizing protocol, via Terminal.text_sized() and Terminal.does_text_sizing().

1.4 Before And After

With the built-in curses module, this is how you would typically print some underlined text at the bottom of the screen:

```
from curses import tigetstr, setupterm, tparm
from fcntl import ioctl
from os import isatty
import struct
import sys
from termios import TIOCGWINSZ

# If we want to tolerate having our output piped to other commands or
# files without crashing, we need to do all this branching:
if hasattr(sys.stdout, 'fileno') and isatty(sys.stdout.fileno()):
```

(continues on next page)

(continued from previous page)

```
    setupterm()
    sc = tigetstr('sc')
    cup = tigetstr('cup')
    rc = tigetstr('rc')
    underline = tigetstr('smul')
    normal = tigetstr('sgr0')
else:
    sc = cup = rc = underline = normal = ''

# Save cursor position.
print(sc)

if cup:
    # tigetnum('lines') doesn't always update promptly, hence this:
    height = struct.unpack('hhhh', ioctl(0, TIOCGWINSZ, '\000' * 8))[0]

    # Move cursor to bottom.
    print(tparm(cup, height - 1, 0))

print(f'This is {underline}underlined{normal}!')

# Restore cursor position.
print(rc)
```

The same program with *Blessed* is simply:

```
from blessed import Terminal

term = Terminal()
with term.location(0, term.height - 1):
    print('This is ' + term.underline('underlined') + '!', end='')
```

CHAPTER

TWO

TERMINAL

Blessed provides just **one** top-level object: *Terminal*. Instantiating a *Terminal* figures out whether you're on a terminal at all and, if so, does any necessary setup:

```
>>> import blessed
>>> term = blessed.Terminal()
```

This is the only object, named `term` here, that you should need from blessed, for all of the remaining examples in our documentation.

You can proceed to ask it all sorts of things about the terminal, such as its size:

```
>>> term.height, term.width
(34, 102)
```

Support for *Colors*:

```
>>> term.number_of_colors
256
```

And create printable strings containing sequences for *Colors*:

```
>>> term.green_reverse('ALL SYSTEMS GO')
'\x1b[32m\x1b[7mALL SYSTEMS GO\x1b[m'
```

When printed, these codes make your terminal go to work:

```
>>> print(term.white_on_firebrick3('SYSTEM OFFLINE'))
```

And thanks to f-strings since python 3.6, it's very easy to mix attributes and strings together:

```
>>> print(f"{term.yellow}Yellow is brown, {term.bright_yellow}"
          f"Bright yellow is actually yellow!{term.normal}")
```

2.1 Capabilities

Any capability in the terminfo(5) manual, under column **Cap-name** can be an attribute of the *Terminal* class, such as 'smul' for 'begin underline mode'.

There are **a lot** of interesting capabilities in the terminfo(5) manual page, but many of these will return an empty string, as they are not supported by your terminal. They can still be used, but have no effect. For example, `blink` only works on a few terminals, does yours?

```
>>> print(term.blink("Insert System disk into drive A:"))
```

2.2 Capability Resolution Order

At time of `Terminal()` initialization, Blessed tries terminal the terminal kind automatically.

1. **'kind'** parameter: an explicit `kind` to *Terminal* is used first, defaults to None.
2. **XTGETTCAP** The remote terminal's automatic response to `TN` (Terminal Name) queried by the `DCS +q` protocol, supported by most terminals.
3. **TERM** environment variable, or legacy ConHost.exe windows terminal descriptor.
4. **'kind_fallback'** (defaults to `'xterm-256color'`). This terminal capability is used when all other methods are exhausted or not found in the jinxed virtual terminal capability database.

2.3 Compound Formatting

If you want to do lots of crazy formatting all at once, you can just mash it all together:

```
>>> print(term.underline_bold_green_on_yellow('They live! In sewers!'))
```

This compound notation comes in handy for users & configuration to customize your app, too!

2.4 Clearing The Screen

Blessed provides syntactic sugar over some screen-clearing capabilities:

`clear`
: Clear the whole screen.

`clear_eol`
: Clear to the end of the line.

`clear_bol`
: Clear backward to the beginning of the line.

`clear_eos`
: Clear to the end of screen.

Suggest to always combine `home` and `clear`, and, in almost all emulators, clearing the screen after setting the background color will repaint the background of the screen:

```
>>> print(term.home + term.on_blue + term.clear)
```

2.5 Hyperlinks

Maybe you haven't noticed, because it's a recent addition to terminal emulators, is that they can now support hyperlinks, like to HTML, or even `file://` URLs, which allows creating clickable links of text.

```
>>> print(f"blessed {term.link('https://blessed.readthedocs.org', 'documentation')}")
blessed documentation
```

Hover your cursor over `documentation`, and it should highlight as a clickable URL.

2.6 Window Title

You can set the terminal window title using *set_window_title()*, which returns the appropriate xterm OSC escape sequence:

```
>>> print(term.set_window_title('My Application'))
```

The `mode` parameter controls what is set: 0 (default) sets both icon name and window title, 1 sets icon name only, 2 sets window title only.

For temporary title changes, use the *window_title()* context manager, which pushes the current title onto the xterm title stack and restores it on exit:

```
with term.window_title('Working...'):
    do_long_task()
# previous title is restored
```

2.7 Progress Bar

Use *progress_bar()* to produce the ConEmu OSC 9;4 protocol escape sequence to display a graphical progress indicator in the terminal's taskbar and/or status area. The `state` parameter accepts a 'state' and 'value' argument. The *value* parameter is required only for `'normal'` state and ignored for all others:

```
# Normal progress at 60%
print(term.progress_bar('normal', 60))

# Enter error state
print(term.progress_bar('error'))

# Indeterminate
print(term.progress_bar('indeterminate'))

# Paused
print(term.progress_bar('pause'))

# Remove the indicator
print(term.progress_bar('clear'))
```

Example program, *progress_bar.py*:

```
from blessed import Terminal


def make_bar(term, state, value=0):
    """Return a labeled ASCII progress-bar string with OSC 9;4 sequence."""
    width = max(10, term.width - 25)
    filled = int(width * value / 100)
    bar = '[' + '=' * filled + '>' + ' ' * (width - filled - 1) + ']'
    ascii_bar = f'{state.title():>13s}: {bar} {value:3d}%'
    return term.move_x(0) + ascii_bar + term.progress_bar(state, value) + term.clear_eol
```

(continues on next page)

(continued from previous page)

```
def dexit(term, delay):
    if term.inkey(delay) == 'q':
        quit()


def main():
    """Program entry point."""
    term = Terminal()

    print(term.reverse(term.center('Progress Bar Demo')))
    print('Progress bar demo')

    def echo(*args):
        print(*args, sep='', end='', flush=True)

    with term.cbreak(), term.hidden_cursor():
        # Normal progress 0 -> 100
        for val_pct in range(0, 101):
            echo(make_bar(term, 'normal', val_pct))
            dexit(term, 0.03)
        echo(make_bar(term, 'clear'))

        # Static states
        for state in ('error', 'paused', 'indeterminate'):
            echo(make_bar(term, state))
            dexit(term, 2)
            echo(make_bar(term, 'clear'))

        # Reverse progress 100 -> 0
        for val_pct in range(100, -1, -1):
            echo(make_bar(term, 'normal', val_pct))
            dexit(term, 0.03)

        echo(make_bar(term, 'clear'))

    print('\nDemo complete.')


if __name__ == '__main__':
    main()
```

2.8 Sixel Graphics Support

Sixel is a bitmap graphics format supported by some modern terminal emulators (including xterm with `-ti vt340`, mlterm, WezTerm, and others), allowing applications to display inline images directly in the terminal.

You can check whether your terminal supports sixel graphics using the *does_sixel()* method:

```
if term.does_sixel():
    display_sixel_image()
```

(continues on next page)

(continued from previous page)

```
else:
    display_text_fallback()
```

Default `timeout` argument of 1 second is used to avoid blocking indefinitely when the terminal fails to respond to DA1 queries.

2.9 Font Glyph Coverage

When a terminal is asked to draw a character its font has no glyph for, it draws a placeholder in its place. These are called "tofu", after U+25A1 WHITE SQUARE () and its resemblance to the grocery food. It may also contain a large 'X', or the hexadecimal value of the codepoint.

The *get_font_coverage()* method queries the terminal support for font glyphs, returning `FontCoverage` that reports on each codepoint of the given text, and whether the font has a glyph for it.

In this example, the terminal's font is tested for an Octant character, new in Unicode 16 (2024):

```
octant = '\U0001CDE4' # "
font_result = term.get_font_coverage(octant)
if not font_result.supported:
    print('This terminal cannot report about tofus, status of octants is unknown.')
elif not font_result.covers(octant):
    print(f'This terminal does not support octants ({octant}).')
else:
    print(f'This terminal does support octants ({octant}).')
```

Two protocols are supported and tested in this order: mintty's Font Glyph Coverage Enquiry (`OSC 7771`), and the Glyph Protocol from Rio terminal.

Answers are cached, so repeated calls do not incur further delay. Very long strings are broken down and queried in batches. Use argument `force=True` to forcefully re-query terminal for new result, such as if a font change is detected. A codepoint the terminal declines to answer for, whether by an error reply or timeout, is reported through `unknown` together with the reason.

⚠ Warning

Coverage is reported by each individual codepoint, no protocol allows for query of a *sequence* of codepoints, graphemes are not supported, meaning a great many kinds of font glyphs that may "tofu" cannot be queried about, which is a limitation of both terminal protocols.

A grapheme cluster may render correctly even though the protocol may report "no support" of its individual codepoints. For example, *U+231A WATCH* followed by *U+FE0E VARIATION SELECTOR-15*: fonts do not holds a glyph for a variation selector 15, so the terminal truthfully reports it as "uncovered" or "tofu", while otherwise drawing the cluster perfectly fine:

```
>>> watch = '\u231a\ufe0e'  # "
>>> font_result = term.get_font_coverage(watch)
>>> font_result
FontCoverage(protocol='glyph', covered=1, uncovered=1, unknown=0)
>>> font_result.covers(watch)
False
>>> list(map(hex, font_result.covered))
['0x231a']
>>> list(map(hex, font_result.uncovered))
['0xfe0e']
```

See example program, *detect-tofus.py*, which detects and reports "tofus" for the attached terminal, and performs NFC normalization and discards "default ignorable codepoints" from reporting, such as the variation selector, above.

2.10 OSC 52 Clipboard

The OSC 52 protocol allows terminal applications to read from and write to the system clipboard without requiring platform-specific clipboard tools. Many modern terminals support this protocol, including xterm, kitty, WezTerm, and iTerm2.

2.10.1 Detection

Use *does_osc52_clipboard()* to check whether your terminal advertises OSC 52 support:

```
>>> if term.does_osc52_clipboard():
...     print("Terminal supports clipboard access")
```

Detection uses DA1 extension 52 and XTGETTCAP `Ms` queries, which do not trigger user-facing clipboard permission dialogs. Detection indicates the terminal *understands* OSC 52, not that any particular read will succeed.

2.10.2 Copying to Clipboard

Use *clipboard_copy()* to copy text to the system clipboard:

```
>>> term.clipboard_copy('Hello from blessed!')
```

Most modern terminals accept clipboard writes without any user prompt, even when clipboard reads are restricted. No detection query is needed for write-only use – the sequence is silently ignored by terminals that do not support it.

2.10.3 Reading from Clipboard

Use *clipboard_paste()* to read the clipboard contents:

```
>>> text = term.clipboard_paste()
>>> if text is not None:
...     print(f"Clipboard: {text}")
```

Warning

Many modern terminals display a permission dialog when an application reads the clipboard. The user must approve the dialog before the terminal sends a response. The default timeout of 10 seconds allows time for this interaction. If the user denies the dialog or the terminal does not support clipboard reads, `None` is returned.

Example program demonstrating clipboard copy and paste:

```
#!/usr/bin/env python
"""Demonstrate OSC 52 clipboard copy and paste."""

from blessed import Terminal

term = Terminal()
```

(continues on next page)

(continued from previous page)

```
print('Checking OSC 52 clipboard support ...', end='', flush=True)

if not term.does_osc52_clipboard():
    print()
    print(term.bright_red('OSC 52 clipboard not detected.'))
    print('Your terminal may still support clipboard writes --')
    print('many terminals accept OSC 52 set without advertising it.')
    raise SystemExit(1)

print(term.bright_green(' supported!'))
print()

# Copy text to clipboard
message = 'Hello from blessed!'
term.clipboard_copy(message)
print(f'Copied to clipboard: {term.bold(repr(message))}')
print()

# Read clipboard back (may trigger a permission prompt)
print('Reading clipboard (your terminal may ask for permission) ...',
      end='', flush=True)
result = term.clipboard_paste()

if result is None:
    print()
    print(term.bright_yellow('No response -- clipboard read may be '
                             'disabled or was denied.'))
else:
    print(term.bright_green(' OK'))
    print(f'Clipboard contains: {term.bold(repr(result))}')
```

2.11 Styled and Colored Underlines

Modern terminals can render underline styles beyond the standard single underline, such as curly (`CSI 4:3 m`), dotted, and dashed underlines. Some also support colored underlines (`CSI 58;2;r;g;b m`), where the underline color differs from the text color.

You can detect these capabilities via XTGETTCAP:

```
>>> if term.does_styled_underlines():
...     print("Curly, dotted, and dashed underlines supported")
```

```
>>> if term.does_colored_underlines():
...     print("Colored underlines supported")
```

These methods query the terminal's `Smulx` and `Setulc` terminfo capabilities, respectively. Default `timeout` argument of 1 second is used.

2.12 Color Scheme Detection

Some terminals can report whether they are in dark or light mode via the color-scheme DSR query (`CSI ? 996 n`). This is supported by Contour, Ghostty, Kitty (0.38.1+), and VTE (0.82.0+).

Use *get_color_scheme()* to query the current preference:

```
>>> scheme = term.get_color_scheme()
>>> if scheme == 'dark':
...     use_dark_palette()
... elif scheme == 'light':
...     use_light_palette()
... else:
...     use_default_palette()
```

Returns `'dark'`, `'light'`, or `None` if the terminal does not support the query. Default `timeout` argument of 1 second is used.

Unlike most detection methods, the result value is not cached – only whether the terminal *supports* the query is remembered. This means repeated calls always return the current scheme, while terminals that do not respond only incur the timeout delay once.

To receive unsolicited notifications when the color scheme changes, enable DEC private mode 2031 (`COLOR_PALETTE_UPDATES`) separately.

2.13 Kitty Query Extensions

Kitty extends the standard XTGETTCAP (`DCS +q`) mechanism with `kitty-query-*` keys that expose runtime metadata such as the terminal name, version, font family, DPI, and clipboard control policy.

You can detect whether these extensions are available using *does_kitty_query()*:

```
>>> if term.does_kitty_query():
...     print("Kitty query extensions available")
```

2.14 DECRQSS Support

DECRQSS (Request Status String) allows applications to query the current state of terminal attributes such as SGR (Select Graphic Rendition), cursor style (DECSCUSR), and conformance level (DECSCL). This is supported by xterm, Contour, kitty, VTE, and others.

You can detect DECRQSS support using *does_decrqss()*:

```
>>> if term.does_decrqss():
...     print("DECRQSS queries supported")
```

2.15 Terminal Software Version

Many modern terminal emulators support the XTVERSION query (CSI > q), which allows applications to identify the terminal software name and version.

You can query the terminal's software version using the *get_software_version()* method, which returns a *SoftwareVersion* object with `name` and `version` attributes, or, `None` if neither method succeeds.

When the TERM_PROGRAM and TERM_PROGRAM_VERSION environment variables are defined, which many modern terminal emulators set (iTerm2, Apple Terminal.app, VS Code, WezTerm, Hyper, mintty, and others), they take precedence and are returned without inquiry. Blank TERM_PROGRAM to have the XTVERSION query made instead. These variables are not forwarded over protocols like ssh, so they are less reliable than XTVERSION.

Example program to display terminal version information:

```
#!/usr/bin/env python

from blessed import Terminal

term = Terminal()

print('Checking software version (XTVERSION) ...', end='', flush=True)

sv = term.get_software_version()

if sv is None:
    print('No response.')
    print(term.bright_red('This terminal does NOT support XTVERSION.'))
else:
    print()
    maybe_version = f', version {sv.version}' if sv.version else ''
    print(f'Terminal: {sv.name}{maybe_version}')
```

2.16 Styles

In addition to *Colors*, blessed also supports the limited amount of *styles* that terminals can do. These are:

bold
: Turn on 'extra bright' mode.

dim
: Enable dim (half-bright) mode.

reverse
: Switch fore and background attributes.

strikethrough
: Enable strikethrough mode.

no_strikethrough
: Disable strikethrough mode.

underline
: Enable underline mode.

no_underline
: Disable underline mode.

normal
: Reset attributes to default.

Note

While the inverse of *underline* is *no_underline*, the only way to turn off *bold* or *reverse* is *normal*, which also cancels any custom colors.

Example program to display supported styles:

```
from blessed import Terminal

term = Terminal()
print(term.home + term.clear)

styles = [
    # common
    ('bold', term.bold),
    ('dim', term.dim),
    ('italic', term.italic),  # often doesn't work
    ('underline', term.underline),
    ('blink', term.blink),
    ('reverse', term.reverse),
    ('standout', term.standout),
    ('strikethrough', term.strikethrough),
    # not common
    ('overline', term.overline),
    ('shadow', term.shadow),
    ('subscript', term.subscript),
    ('superscript', term.superscript),
]

for name, cap in styles:
    if cap:
        txt = cap('The quick brown fox') + '   ' + repr(str(cap))
    else:
        txt = term.webgray('(unsupported)')
    print(f'{name:>14s}   {txt}')
```

2.17 Full-Screen Mode

If you've ever noticed how a program like vim(1) restores you to your unix shell history after exiting, it's actually a pretty basic trick that all terminal emulators support, that *blessed* provides using the `fullscreen()` context manager over these two basic capabilities:

`enter_fullscreen`
: Switch to alternate screen, previous screen is stored by terminal driver.

`exit_fullscreen`
: Switch back to standard screen, restoring the same terminal screen.

```
with term.fullscreen(), term.cbreak():
    print(term.move_y(term.height // 2) +
          term.center('press any key').rstrip())
    term.inkey()
```

2.18 Line Wrap Control

Terminals normally wrap text when reaching the right edge of the window. This context manager temporarily disables it.

In the following example, the letter 'X' is displayed 3 times the width of the page, which would otherwise wrap and fill 3 lines. With line-wrapping disabled, the cursor stays at the last column and continuously "repaints" over the previous one until a newline is received, causing it to fill only 1 line.

```
with term.no_line_wrap():
    print(term.move_x(0) + 'X' * (term.width * 3))
```

You can use these sequences directly as attributes `enable_line_wrap` and `disable_line_wrap` of *Terminal*.

```
print(term.enable_line_wrap)
```

2.19 Pipe Savvy

If your program isn't attached to a terminal, such as piped to a program like less(1) or redirected to a file, all the capability attributes on *Terminal* will return empty strings for any *Colors*, *Location*, or other sequences. You'll get a nice-looking file without any formatting codes gumming up the works.

If you want to override this, such as when piping output to `less -R`, pass argument value *True* to the *force_styling* parameter.

In any case, there is a *does_styling* attribute that lets you see whether the terminal attached to the output stream is capable of formatting. If it is *False*, you may refrain from drawing progress bars and other frippery and just stick to content:

```
if term.does_styling:
    with term.location(x=0, y=term.height - 1):
        print('Progress: [=======>   ]')
print(term.bold("60%"))
```

2.20 Singleton-Free

Beginning 1.40, blessed now uses jinxed instead of `curses`, and is no longer subject to the per-process terminal-kind restriction that curses imposes, "Singleton-free".

Multiple *Terminal* instances with different `kind` values can coexist within the same process. This might also be considered asyncio or thread-safe.

CHAPTER

THREE

COLORS

Doing colors with blessed is easy, pick a color name from the *All Terminal colors, by name* below, any of these named are also attributes of the *Terminal*!

These attributes can be printed directly, causing the terminal to switch into the given color. Or, as a callable, which terminates the string with the `normal` attribute. The following three statements are equivalent:

```
>>> print(term.orangered + 'All systems are offline' + term.normal)
>>> print(f'{term.orangered}All systems are offline{term.normal}')
>>> print(term.orangered('All systems are offline'))
```

To use a background color, prefix any color with `on_`:

```
>>> print(term.on_darkolivegreen('welcome to the army'))
```

And combine two colors using "`_on_`", as in "`foreground_on_background`":

```
>>> print(term.peru_on_seagreen('All systems functioning within defined parameters.'))
```

If you have an exact color in mind, you can use direct `(r, g, b)` values of 0-255 with *`color_rgb()`* and *`on_color_rgb()`* for foreground and background colors, and, *`color_hex()`* and *`on_color_hex()`* with popular html color codes:

```
>>> print(term.color_rgb(255, 0, 255)("Bright Magenta"))
>>> print(term.color_hex("#cafe00")("green goo"))
```

3.1 24-bit Colors

Most Terminal emulators, even Windows, has supported 24-bit colors since roughly 2016. To test or whether the terminal emulator supports 24-bit colors, check terminal attribute `number_of_colors()`:

```
>>> print(term.number_of_colors == 1 << 24)
True
```

This value is automatically determined 24-bit when environment `COLORTERM` is set to `truecolor` or `24bit`. The attribute can be modified directly for the desired color depth.

Even if the terminal only supports `256`, or worse, `16` colors, the nearest color supported by the terminal is automatically mapped:

```
>>> term.number_of_colors = 1 << 24
>>> term.darkolivegreen
'\x1b[38;2;85;107;47m'
```

```
>>> term.number_of_colors = 256
>>> term.darkolivegreen
'\x1b[38;5;58m'
```

```
>>> term.number_of_colors = 16
>>> term.darkolivegreen
'\x1b[90m'
```

3.2 Hex Colors

For convenience, colors can be specified in standard web hex format using *color_hex()* and *on_color_hex()*:

```
>>> print(term.color_hex('#ff5733')('Warning!'))
>>> print(term.on_color_hex('#ff5733')('System Error!'))
```

These methods accept multiple hex formats, and the # prefix is optional:

- **3-digit**: #RGB expands to #RRGGBB (e.g., `#faf` -> `#ffaaff`)
- **6-digit**: #RRGGBB standard web format (e.g., `#ff5733`)
- **12-digit**: #RRRRGGGGBBBB

3.3 Querying Terminal Colors

You can query the terminal's default foreground and background colors using *get_fgcolor()* and *get_bgcolor()*, or use *get_fgcolor_hex()* and *get_bgcolor_hex()* for hex string output. These return the colors used for default, uncolored text - not any currently active color set by escape sequences:

```
#!/usr/bin/env python
"""Query terminal foreground and background colors."""
from blessed import Terminal

term = Terminal()

fg_hex = term.get_fgcolor_hex()
bg_hex = term.get_bgcolor_hex()

bg_rgb = term.get_bgcolor(bits=8)
fg_rgb = term.get_fgcolor(bits=8)

print(f"Foreground {fg_hex} {fg_rgb}")
print(f"Background {bg_hex} {bg_rgb}")
```

Note

The keyword argument `bits=8` is recommended for the more common RGB return values in range of 0-255 with *get_fgcolor()* and *get_bgcolor()* as demonstrated here.

These methods otherwise return 16-bit RGB values (0-65535) per the XParseColor specification used by the underlying protocol.

These methods are useful for adapting your application's color scheme to match the user's terminal theme, or for detecting whether the terminal has a light or dark background. The RGB methods return `(-1, -1, -1)` on timeout, while the hex methods return an empty string.

3.4 256 Colors

The built-in capability *color()* accepts a numeric index of any value between 0 and 254, you could call this "Color by number…" and is the highest color depth for some legacy terminals and systems. It not recommended to use directly. It is better to use RGB and Hex value methods, which map to the nearest color automatically.

3.5 16 Colors

Recommended for common CLI applications, where the user's preferred color palette is used.

Traditional terminals are only capable of 8 colors:

- `black`
- `red`
- `green`
- `yellow`
- `blue`
- `magenta`
- `cyan`
- `white`

Prefixed with *on_*, the given color is used as the background color:

- `on_black`
- `on_red`
- `on_green`
- `on_yellow`
- `on_blue`
- `on_magenta`
- `on_cyan`
- `on_white`

The same colors, prefixed with *bright_* or *bold_*, such as *bright_blue*, provides the other 8 colors of a 16-color terminal:

- `bright_black`
- `bright_red`
- `bright_green`
- `bright_yellow`
- `bright_blue`
- `bright_magenta`
- `bright_cyan`

- `bright_white`

Combined, there are actually **three shades of grey** for 16-color terminals, in ascending order of intensity:

- `bright_black`: is dark grey.
- `white`: a mild white.
- `bright_white`: pure white (`#ffffff`).

Note

- *bright_black* is actually a very dark shade of grey!
- *yellow is brown*, only high-intensity yellow (`bright_yellow`) is yellow!
- purple is magenta.

Warning

Terminal emulators use different values for any of these 16 colors, the most common of these are displayed at https://en.wikipedia.org/wiki/ANSI_escape_code#3-bit_and_4-bit. Users can customize these 16 colors as a common "theme", so that one CLI application appears of the same color theme as the next.

When exact color values are needed, *24-bit Colors* should be preferred, by their name or RGB value.

3.6 Monochrome

One small consideration for targeting legacy terminals, such as a *vt220*, which do not support colors but do support reverse video: select a foreground color, followed by reverse video, rather than selecting a background color directly:: the same desired background color effect as `on_background`:

```
>>> print(term.on_green('This will not standout on a vt220'))
>>> print(term.green_reverse('Though some terminals standout more than others'))
```

The second phrase appears as *black on green* on both color terminals and a green monochrome vt220.

3.7 Color chart

Table 1: All Terminal colors, by name

Name	Image	R	G	B	H	S	V
red		100.0%	0.0%	0.0%	0.0%	100.0%	100.0%
red2		93.3%	0.0%	0.0%	0.0%	100.0%	93.3%
red3		80.4%	0.0%	0.0%	0.0%	100.0%	80.4%
snow		100.0%	98.0%	98.0%	0.0%	2.0%	100.0%
snow2		93.3%	91.4%	91.4%	0.0%	2.1%	93.3%
snow3		80.4%	78.8%	78.8%	0.0%	2.0%	80.4%
snow4		54.5%	53.7%	53.7%	0.0%	1.4%	54.5%
brown		64.7%	16.5%	16.5%	0.0%	74.5%	64.7%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
brown1		100.0%	25.1%	25.1%	0.0%	74.9%	100.0%
brown2		93.3%	23.1%	23.1%	0.0%	75.2%	93.3%
brown3		80.4%	20.0%	20.0%	0.0%	75.1%	80.4%
brown4		54.5%	13.7%	13.7%	0.0%	74.8%	54.5%
darkred		54.5%	0.0%	0.0%	0.0%	100.0%	54.5%
indianred		80.4%	36.1%	36.1%	0.0%	55.1%	80.4%
indianred1		100.0%	41.6%	41.6%	0.0%	58.4%	100.0%
indianred2		93.3%	38.8%	38.8%	0.0%	58.4%	93.3%
indianred3		80.4%	33.3%	33.3%	0.0%	58.5%	80.4%
indianred4		54.5%	22.7%	22.7%	0.0%	58.3%	54.5%
firebrick		69.8%	13.3%	13.3%	0.0%	80.9%	69.8%
firebrick1		100.0%	18.8%	18.8%	0.0%	81.2%	100.0%
firebrick2		93.3%	17.3%	17.3%	0.0%	81.5%	93.3%
firebrick3		80.4%	14.9%	14.9%	0.0%	81.5%	80.4%
firebrick4		54.5%	10.2%	10.2%	0.0%	81.3%	54.5%
webma-roon		50.2%	0.0%	0.0%	0.0%	100.0%	50.2%
rosybrown		73.7%	56.1%	56.1%	0.0%	23.9%	73.7%
rosy-brown1		100.0%	75.7%	75.7%	0.0%	24.3%	100.0%
rosy-brown2		93.3%	70.6%	70.6%	0.0%	24.4%	93.3%
rosy-brown3		80.4%	60.8%	60.8%	0.0%	24.4%	80.4%
rosy-brown4		54.5%	41.2%	41.2%	0.0%	24.5%	54.5%
lightcoral		94.1%	50.2%	50.2%	0.0%	46.7%	94.1%
salmon		98.0%	50.2%	44.7%	1.7%	54.4%	98.0%
mistyrose		100.0%	89.4%	88.2%	1.7%	11.8%	100.0%
mistyrose2		93.3%	83.5%	82.4%	1.8%	11.8%	93.3%
mistyrose3		80.4%	71.8%	71.0%	1.4%	11.7%	80.4%
coral1		100.0%	44.7%	33.7%	2.8%	66.3%	100.0%
coral2		93.3%	41.6%	31.4%	2.7%	66.4%	93.3%
coral3		80.4%	35.7%	27.1%	2.7%	66.3%	80.4%
coral4		54.5%	24.3%	18.4%	2.7%	66.2%	54.5%
tomato		100.0%	38.8%	27.8%	2.5%	72.2%	100.0%
tomato2		93.3%	36.1%	25.9%	2.5%	72.3%	93.3%
tomato3		80.4%	31.0%	22.4%	2.5%	72.2%	80.4%
tomato4		54.5%	21.2%	14.9%	2.6%	72.7%	54.5%
mistyrose4		54.5%	49.0%	48.2%	2.1%	11.5%	54.5%
salmon1		100.0%	54.9%	41.2%	3.9%	58.8%	100.0%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
salmon2		93.3%	51.0%	38.4%	3.8%	58.8%	93.3%
salmon3		80.4%	43.9%	32.9%	3.9%	59.0%	80.4%
salmon4		54.5%	29.8%	22.4%	3.9%	59.0%	54.5%
coral		100.0%	49.8%	31.4%	4.5%	68.6%	100.0%
orangered		100.0%	27.1%	0.0%	4.5%	100.0%	100.0%
orangered2		93.3%	25.1%	0.0%	4.5%	100.0%	93.3%
orangered3		80.4%	21.6%	0.0%	4.5%	100.0%	80.4%
orangered4		54.5%	14.5%	0.0%	4.4%	100.0%	54.5%
darksalmon		91.4%	58.8%	47.8%	4.2%	47.6%	91.4%
light-salmon		100.0%	62.7%	47.8%	4.8%	52.2%	100.0%
light-salmon2		93.3%	58.4%	44.7%	4.7%	52.1%	93.3%
light-salmon3		80.4%	50.6%	38.4%	4.8%	52.2%	80.4%
light-salmon4		54.5%	34.1%	25.9%	4.8%	52.5%	54.5%
sienna		62.7%	32.2%	17.6%	5.4%	71.9%	62.7%
sienna1		100.0%	51.0%	27.8%	5.3%	72.2%	100.0%
sienna2		93.3%	47.5%	25.9%	5.3%	72.3%	93.3%
sienna3		80.4%	40.8%	22.4%	5.3%	72.2%	80.4%
sienna4		54.5%	27.8%	14.9%	5.4%	72.7%	54.5%
seashell		100.0%	96.1%	93.3%	6.9%	6.7%	100.0%
chocolate		82.4%	41.2%	11.8%	6.9%	85.7%	82.4%
chocolate1		100.0%	49.8%	14.1%	6.9%	85.9%	100.0%
chocolate2		93.3%	46.3%	12.9%	6.9%	86.1%	93.3%
chocolate3		80.4%	40.0%	11.4%	6.9%	85.9%	80.4%
chocolate4		54.5%	27.1%	7.5%	6.9%	86.3%	54.5%
seashell2		93.3%	89.8%	87.1%	7.3%	6.7%	93.3%
seashell3		80.4%	77.3%	74.9%	7.1%	6.8%	80.4%
seashell4		54.5%	52.5%	51.0%	7.4%	6.5%	54.5%
peachpuff		100.0%	85.5%	72.5%	7.9%	27.5%	100.0%
peachpuff2		93.3%	79.6%	67.8%	7.7%	27.3%	93.3%
peachpuff3		80.4%	68.6%	58.4%	7.7%	27.3%	80.4%
peachpuff4		54.5%	46.7%	39.6%	7.9%	27.3%	54.5%
sandy-brown		95.7%	64.3%	37.6%	7.7%	60.7%	95.7%
tan1		100.0%	64.7%	31.0%	8.1%	69.0%	100.0%
tan2		93.3%	60.4%	28.6%	8.2%	69.3%	93.3%
tan4		54.5%	35.3%	16.9%	8.2%	69.1%	54.5%
peru		80.4%	52.2%	24.7%	8.2%	69.3%	80.4%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
linen		98.0%	94.1%	90.2%	8.3%	8.0%	98.0%
bisque3		80.4%	71.8%	62.0%	8.9%	22.9%	80.4%
darkorange1		100.0%	49.8%	0.0%	8.3%	100.0%	100.0%
darkorange2		93.3%	46.3%	0.0%	8.3%	100.0%	93.3%
darkorange3		80.4%	40.0%	0.0%	8.3%	100.0%	80.4%
darkorange4		54.5%	27.1%	0.0%	8.3%	100.0%	54.5%
tan		82.4%	70.6%	54.9%	9.5%	33.3%	82.4%
bisque		100.0%	89.4%	76.9%	9.0%	23.1%	100.0%
bisque2		93.3%	83.5%	71.8%	9.1%	23.1%	93.3%
bisque4		54.5%	49.0%	42.0%	9.4%	23.0%	54.5%
burlywood		87.1%	72.2%	52.9%	9.4%	39.2%	87.1%
burlywood1		100.0%	82.7%	60.8%	9.3%	39.2%	100.0%
burlywood2		93.3%	77.3%	56.9%	9.3%	39.1%	93.3%
burlywood3		80.4%	66.7%	49.0%	9.4%	39.0%	80.4%
burlywood4		54.5%	45.1%	33.3%	9.3%	38.8%	54.5%
darkorange		100.0%	54.9%	0.0%	9.2%	100.0%	100.0%
navajowhite		100.0%	87.1%	67.8%	10.0%	32.2%	100.0%
navajowhite2		93.3%	81.2%	63.1%	10.0%	32.4%	93.3%
antiquewhite		98.0%	92.2%	84.3%	9.5%	14.0%	98.0%
antiquewhite1		100.0%	93.7%	85.9%	9.3%	14.1%	100.0%
antiquewhite2		93.3%	87.5%	80.0%	9.3%	14.3%	93.3%
antiquewhite3		80.4%	75.3%	69.0%	9.2%	14.1%	80.4%
antiquewhite4		54.5%	51.4%	47.1%	9.6%	13.7%	54.5%
wheat		96.1%	87.1%	70.2%	10.9%	26.9%	96.1%
wheat1		100.0%	90.6%	72.9%	10.9%	27.1%	100.0%
wheat2		93.3%	84.7%	68.2%	10.9%	26.9%	93.3%
wheat3		80.4%	72.9%	58.8%	10.9%	26.8%	80.4%
wheat4		54.5%	49.4%	40.0%	10.8%	26.6%	54.5%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
orange		100.0%	64.7%	0.0%	10.8%	100.0%	100.0%
orange2		93.3%	60.4%	0.0%	10.8%	100.0%	93.3%
orange3		80.4%	52.2%	0.0%	10.8%	100.0%	80.4%
orange4		54.5%	35.3%	0.0%	10.8%	100.0%	54.5%
oldlace		99.2%	96.1%	90.2%	10.9%	9.1%	99.2%
moccasin		100.0%	89.4%	71.0%	10.6%	29.0%	100.0%
papayawhip		100.0%	93.7%	83.5%	10.3%	16.5%	100.0%
navajowhite3		80.4%	70.2%	54.5%	10.1%	32.2%	80.4%
navajowhite4		54.5%	47.5%	36.9%	10.0%	32.4%	54.5%
blanchedalmond		100.0%	92.2%	80.4%	10.0%	19.6%	100.0%
goldenrod		85.5%	64.7%	12.5%	11.9%	85.3%	85.5%
goldenrod1		100.0%	75.7%	14.5%	11.9%	85.5%	100.0%
goldenrod2		93.3%	70.6%	13.3%	11.9%	85.7%	93.3%
goldenrod3		80.4%	60.8%	11.4%	11.9%	85.9%	80.4%
goldenrod4		54.5%	41.2%	7.8%	11.9%	85.6%	54.5%
floralwhite		100.0%	98.0%	94.1%	11.1%	5.9%	100.0%
darkgoldenrod		72.2%	52.5%	4.3%	11.8%	94.0%	72.2%
darkgoldenrod1		100.0%	72.5%	5.9%	11.8%	94.1%	100.0%
darkgoldenrod2		93.3%	67.8%	5.5%	11.8%	94.1%	93.3%
darkgoldenrod3		80.4%	58.4%	4.7%	11.8%	94.1%	80.4%
darkgoldenrod4		54.5%	39.6%	3.1%	11.8%	94.2%	54.5%
cornsilk		100.0%	97.3%	86.3%	13.3%	13.7%	100.0%
cornsilk2		93.3%	91.0%	80.4%	13.6%	13.9%	93.3%
cornsilk3		80.4%	78.4%	69.4%	13.7%	13.7%	80.4%
lightgoldenrod1		100.0%	92.5%	54.5%	13.9%	45.5%	100.0%
lightgoldenrod2		93.3%	86.3%	51.0%	13.9%	45.4%	93.3%
lightgoldenrod3		80.4%	74.5%	43.9%	14.0%	45.4%	80.4%
gold		100.0%	84.3%	0.0%	14.1%	100.0%	100.0%
gold2		93.3%	78.8%	0.0%	14.1%	100.0%	93.3%
gold3		80.4%	67.8%	0.0%	14.1%	100.0%	80.4%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
gold4		54.5%	45.9%	0.0%	14.0%	100.0%	54.5%
cornsilk4		54.5%	53.3%	47.1%	14.0%	13.7%	54.5%
lemonchiffon2		93.3%	91.4%	74.9%	14.9%	19.7%	93.3%
lightgoldenrod		93.3%	86.7%	51.0%	14.0%	45.4%	93.3%
lightgoldenrod4		54.5%	50.6%	29.8%	14.0%	45.3%	54.5%
khaki		94.1%	90.2%	54.9%	15.0%	41.7%	94.1%
khaki1		100.0%	96.5%	56.1%	15.3%	43.9%	100.0%
khaki2		93.3%	90.2%	52.2%	15.4%	44.1%	93.3%
khaki3		80.4%	77.6%	45.1%	15.4%	43.9%	80.4%
khaki4		54.5%	52.5%	30.6%	15.3%	43.9%	54.5%
darkkhaki		74.1%	71.8%	42.0%	15.4%	43.4%	74.1%
lemonchiffon		100.0%	98.0%	80.4%	15.0%	19.6%	100.0%
lemonchiffon3		80.4%	78.8%	64.7%	15.0%	19.5%	80.4%
lemonchiffon4		54.5%	53.7%	43.9%	15.4%	19.4%	54.5%
palegoldenrod		93.3%	91.0%	66.7%	15.2%	28.6%	93.3%
beige		96.1%	96.1%	86.3%	16.7%	10.2%	96.1%
olive		50.2%	50.2%	0.0%	16.7%	100.0%	50.2%
ivory		100.0%	100.0%	94.1%	16.7%	5.9%	100.0%
ivory2		93.3%	93.3%	87.8%	16.7%	5.9%	93.3%
ivory3		80.4%	80.4%	75.7%	16.7%	5.9%	80.4%
ivory4		54.5%	54.5%	51.4%	16.7%	5.8%	54.5%
yellow		100.0%	100.0%	0.0%	16.7%	100.0%	100.0%
yellow2		93.3%	93.3%	0.0%	16.7%	100.0%	93.3%
yellow3		80.4%	80.4%	0.0%	16.7%	100.0%	80.4%
yellow4		54.5%	54.5%	0.0%	16.7%	100.0%	54.5%
lightyellow		100.0%	100.0%	87.8%	16.7%	12.2%	100.0%
lightyellow2		93.3%	93.3%	82.0%	16.7%	12.2%	93.3%
lightyellow3		80.4%	80.4%	70.6%	16.7%	12.2%	80.4%
lightyellow4		54.5%	54.5%	47.8%	16.7%	12.2%	54.5%
lightgoldenrodyellow		98.0%	98.0%	82.4%	16.7%	16.0%	98.0%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
olivedrab		42.0%	55.7%	13.7%	22.1%	75.4%	55.7%
olivedrab1		75.3%	100.0%	24.3%	22.1%	75.7%	100.0%
olivedrab2		70.2%	93.3%	22.7%	22.1%	75.6%	93.3%
olivedrab3		60.4%	80.4%	19.6%	22.2%	75.6%	80.4%
olivedrab4		41.2%	54.5%	13.3%	22.1%	75.5%	54.5%
darkolivegreen		33.3%	42.0%	18.4%	22.8%	56.1%	42.0%
darkolivegreen1		79.2%	100.0%	43.9%	22.8%	56.1%	100.0%
darkolivegreen2		73.7%	93.3%	40.8%	22.9%	56.3%	93.3%
darkolivegreen3		63.5%	80.4%	35.3%	22.9%	56.1%	80.4%
darkolivegreen4		43.1%	54.5%	23.9%	22.9%	56.1%	54.5%
greenyellow		67.8%	100.0%	18.4%	23.2%	81.6%	100.0%
lawngreen		48.6%	98.8%	0.0%	25.1%	100.0%	98.8%
chartreuse		49.8%	100.0%	0.0%	25.0%	100.0%	100.0%
chartreuse2		46.3%	93.3%	0.0%	25.1%	100.0%	93.3%
chartreuse3		40.0%	80.4%	0.0%	25.0%	100.0%	80.4%
chartreuse4		27.1%	54.5%	0.0%	25.1%	100.0%	54.5%
green		0.0%	100.0%	0.0%	33.3%	100.0%	100.0%
green2		0.0%	93.3%	0.0%	33.3%	100.0%	93.3%
green3		0.0%	80.4%	0.0%	33.3%	100.0%	80.4%
green4		0.0%	54.5%	0.0%	33.3%	100.0%	54.5%
webgreen		0.0%	50.2%	0.0%	33.3%	100.0%	50.2%
honeydew		94.1%	100.0%	94.1%	33.3%	5.9%	100.0%
honeydew2		87.8%	93.3%	87.8%	33.3%	5.9%	93.3%
honeydew3		75.7%	80.4%	75.7%	33.3%	5.9%	80.4%
honeydew4		51.4%	54.5%	51.4%	33.3%	5.8%	54.5%
darkgreen		0.0%	39.2%	0.0%	33.3%	100.0%	39.2%
palegreen		59.6%	98.4%	59.6%	33.3%	39.4%	98.4%
palegreen1		60.4%	100.0%	60.4%	33.3%	39.6%	100.0%
palegreen3		48.6%	80.4%	48.6%	33.3%	39.5%	80.4%
palegreen4		32.9%	54.5%	32.9%	33.3%	39.6%	54.5%
limegreen		19.6%	80.4%	19.6%	33.3%	75.6%	80.4%
lightgreen		56.5%	93.3%	56.5%	33.3%	39.5%	93.3%
forestgreen		13.3%	54.5%	13.3%	33.3%	75.5%	54.5%
darkseagreen		56.1%	73.7%	56.1%	33.3%	23.9%	73.7%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
darksea-green1		75.7%	100.0%	75.7%	33.3%	24.3%	100.0%
darksea-green2		70.6%	93.3%	70.6%	33.3%	24.4%	93.3%
darksea-green3		60.8%	80.4%	60.8%	33.3%	24.4%	80.4%
darksea-green4		41.2%	54.5%	41.2%	33.3%	24.5%	54.5%
seagreen		18.0%	54.5%	34.1%	40.7%	66.9%	54.5%
seagreen1		32.9%	100.0%	62.4%	40.6%	67.1%	100.0%
seagreen2		30.6%	93.3%	58.0%	40.6%	67.2%	93.3%
seagreen3		26.3%	80.4%	50.2%	40.7%	67.3%	80.4%
medium-seagreen		23.5%	70.2%	44.3%	40.8%	66.5%	70.2%
mintcream		96.1%	100.0%	98.0%	41.7%	3.9%	100.0%
spring-green		0.0%	100.0%	49.8%	41.6%	100.0%	100.0%
spring-green2		0.0%	93.3%	46.3%	41.6%	100.0%	93.3%
spring-green3		0.0%	80.4%	40.0%	41.6%	100.0%	80.4%
spring-green4		0.0%	54.5%	27.1%	41.6%	100.0%	54.5%
medium-spring-green		0.0%	98.0%	60.4%	43.6%	100.0%	98.0%
aquama-rine		49.8%	100.0%	83.1%	44.4%	50.2%	100.0%
aquama-rine2		46.3%	93.3%	77.6%	44.4%	50.4%	93.3%
aquama-rine3		40.0%	80.4%	66.7%	44.3%	50.2%	80.4%
aquama-rine4		27.1%	54.5%	45.5%	44.5%	50.4%	54.5%
turquoise		25.1%	87.8%	81.6%	48.3%	71.4%	87.8%
lightsea-green		12.5%	69.8%	66.7%	49.1%	82.0%	69.8%
medium-turquoise		28.2%	82.0%	80.0%	49.4%	65.6%	82.0%
teal		0.0%	50.2%	50.2%	50.0%	100.0%	50.2%
aqua		0.0%	100.0%	100.0%	50.0%	100.0%	100.0%
cyan2		0.0%	93.3%	93.3%	50.0%	100.0%	93.3%
cyan3		0.0%	80.4%	80.4%	50.0%	100.0%	80.4%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
cyan4		0.0%	54.5%	54.5%	50.0%	100.0%	54.5%
azure		94.1%	100.0%	100.0%	50.0%	5.9%	100.0%
azure2		87.8%	93.3%	93.3%	50.0%	5.9%	93.3%
azure3		75.7%	80.4%	80.4%	50.0%	5.9%	80.4%
azure4		51.4%	54.5%	54.5%	50.0%	5.8%	54.5%
cadetblue		37.3%	62.0%	62.7%	50.5%	40.6%	62.7%
lightcyan		87.8%	100.0%	100.0%	50.0%	12.2%	100.0%
lightcyan2		82.0%	93.3%	93.3%	50.0%	12.2%	93.3%
lightcyan3		70.6%	80.4%	80.4%	50.0%	12.2%	80.4%
lightcyan4		47.8%	54.5%	54.5%	50.0%	12.2%	54.5%
turquoise1		0.0%	96.1%	100.0%	50.7%	100.0%	100.0%
turquoise2		0.0%	89.8%	93.3%	50.6%	100.0%	93.3%
turquoise3		0.0%	77.3%	80.4%	50.7%	100.0%	80.4%
turquoise4		0.0%	52.5%	54.5%	50.6%	100.0%	54.5%
dark-slategray		18.4%	31.0%	31.0%	50.0%	40.5%	31.0%
dark-slategray1		59.2%	100.0%	100.0%	50.0%	40.8%	100.0%
dark-slategray2		55.3%	93.3%	93.3%	50.0%	40.8%	93.3%
dark-slategray3		47.5%	80.4%	80.4%	50.0%	41.0%	80.4%
dark-slategray4		32.2%	54.5%	54.5%	50.0%	41.0%	54.5%
dark-turquoise		0.0%	80.8%	82.0%	50.2%	100.0%	82.0%
pale-turquoise		68.6%	93.3%	93.3%	50.0%	26.5%	93.3%
pale-turquoise1		73.3%	100.0%	100.0%	50.0%	26.7%	100.0%
pale-turquoise2		68.2%	93.3%	93.3%	50.0%	26.9%	93.3%
pale-turquoise3		58.8%	80.4%	80.4%	50.0%	26.8%	80.4%
pale-turquoise4		40.0%	54.5%	54.5%	50.0%	26.6%	54.5%
cadetblue1		59.6%	96.1%	100.0%	51.6%	40.4%	100.0%
cadetblue2		55.7%	89.8%	93.3%	51.6%	40.3%	93.3%
cadetblue3		47.8%	77.3%	80.4%	51.6%	40.5%	80.4%
cadetblue4		32.5%	52.5%	54.5%	51.5%	40.3%	54.5%
pow-derblue		69.0%	87.8%	90.2%	51.9%	23.5%	90.2%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
lightblue4		40.8%	51.4%	54.5%	53.8%	25.2%	54.5%
skyblue		52.9%	80.8%	92.2%	54.8%	42.6%	92.2%
lightblue		67.8%	84.7%	90.2%	54.1%	24.8%	90.2%
lightblue1		74.9%	93.7%	100.0%	54.2%	25.1%	100.0%
lightblue2		69.8%	87.5%	93.3%	54.2%	25.2%	93.3%
lightblue3		60.4%	75.3%	80.4%	54.2%	24.9%	80.4%
deepsky-blue		0.0%	74.9%	100.0%	54.2%	100.0%	100.0%
deepsky-blue2		0.0%	69.8%	93.3%	54.2%	100.0%	93.3%
deepsky-blue3		0.0%	60.4%	80.4%	54.1%	100.0%	80.4%
deepsky-blue4		0.0%	40.8%	54.5%	54.2%	100.0%	54.5%
lightsky-blue3		55.3%	71.4%	80.4%	56.0%	31.2%	80.4%
skyblue1		52.9%	80.8%	100.0%	56.8%	47.1%	100.0%
skyblue2		49.4%	75.3%	93.3%	56.8%	47.1%	93.3%
skyblue3		42.4%	65.1%	80.4%	56.7%	47.3%	80.4%
skyblue4		29.0%	43.9%	54.5%	56.9%	46.8%	54.5%
lightsky-blue		52.9%	80.8%	98.0%	56.4%	46.0%	98.0%
lightsky-blue1		69.0%	88.6%	100.0%	56.1%	31.0%	100.0%
lightsky-blue2		64.3%	82.7%	93.3%	56.1%	31.1%	93.3%
lightsky-blue4		37.6%	48.2%	54.5%	56.2%	30.9%	54.5%
aliceblue		94.1%	97.3%	100.0%	57.8%	5.9%	100.0%
steelblue		27.5%	51.0%	70.6%	57.6%	61.1%	70.6%
steelblue1		38.8%	72.2%	100.0%	57.6%	61.2%	100.0%
steelblue2		36.1%	67.5%	93.3%	57.5%	61.3%	93.3%
steelblue3		31.0%	58.0%	80.4%	57.5%	61.5%	80.4%
steelblue4		21.2%	39.2%	54.5%	57.6%	61.2%	54.5%
slategray		43.9%	50.2%	56.5%	58.3%	22.2%	56.5%
slategray1		77.6%	88.6%	100.0%	58.5%	22.4%	100.0%
slategray2		72.5%	82.7%	93.3%	58.5%	22.3%	93.3%
slategray3		62.4%	71.4%	80.4%	58.3%	22.4%	80.4%
slategray4		42.4%	48.2%	54.5%	58.6%	22.3%	54.5%
dodgerblue		11.8%	56.5%	100.0%	58.2%	88.2%	100.0%
dodgerblue2		11.0%	52.5%	93.3%	58.3%	88.2%	93.3%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
dodgerblue3		9.4%	45.5%	80.4%	58.2%	88.3%	80.4%
dodgerblue4		6.3%	30.6%	54.5%	58.3%	88.5%	54.5%
lights-lategray		46.7%	53.3%	60.0%	58.3%	22.2%	60.0%
lightsteel-blue		69.0%	76.9%	87.1%	59.4%	20.7%	87.1%
lightsteel-blue1		79.2%	88.2%	100.0%	59.4%	20.8%	100.0%
lightsteel-blue2		73.7%	82.4%	93.3%	59.3%	21.0%	93.3%
lightsteel-blue3		63.5%	71.0%	80.4%	59.3%	21.0%	80.4%
lightsteel-blue4		43.1%	48.2%	54.5%	59.2%	20.9%	54.5%
corn-flowerblue		39.2%	58.4%	92.9%	60.7%	57.8%	92.9%
royalblue		25.5%	41.2%	88.2%	62.5%	71.1%	88.2%
royalblue1		28.2%	46.3%	100.0%	62.5%	71.8%	100.0%
royalblue2		26.3%	43.1%	93.3%	62.5%	71.8%	93.3%
royalblue3		22.7%	37.3%	80.4%	62.5%	71.7%	80.4%
royalblue4		15.3%	25.1%	54.5%	62.5%	71.9%	54.5%
blue		0.0%	0.0%	100.0%	66.7%	100.0%	100.0%
blue2		0.0%	0.0%	93.3%	66.7%	100.0%	93.3%
blue3		0.0%	0.0%	80.4%	66.7%	100.0%	80.4%
blue4		0.0%	0.0%	54.5%	66.7%	100.0%	54.5%
navy		0.0%	0.0%	50.2%	66.7%	100.0%	50.2%
lavender		90.2%	90.2%	98.0%	66.7%	8.0%	98.0%
ghostwhite		97.3%	97.3%	100.0%	66.7%	2.7%	100.0%
midnight-blue		9.8%	9.8%	43.9%	66.7%	77.7%	43.9%
slateblue		41.6%	35.3%	80.4%	69.0%	56.1%	80.4%
slateblue1		51.4%	43.5%	100.0%	69.0%	56.5%	100.0%
slateblue3		41.2%	34.9%	80.4%	69.0%	56.6%	80.4%
slateblue4		27.8%	23.5%	54.5%	69.0%	56.8%	54.5%
lightslate-blue		51.8%	43.9%	100.0%	69.0%	56.1%	100.0%
slateblue2		47.8%	40.4%	93.3%	69.0%	56.7%	93.3%
darkslate-blue		28.2%	23.9%	54.5%	69.0%	56.1%	54.5%
mediums-lateblue		48.2%	40.8%	93.3%	69.0%	56.3%	93.3%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
medi-umpurple		57.6%	43.9%	85.9%	72.1%	48.9%	85.9%
medi-umpurple1		67.1%	51.0%	100.0%	72.1%	49.0%	100.0%
medi-umpurple2		62.4%	47.5%	93.3%	72.1%	49.2%	93.3%
medi-umpurple3		53.7%	40.8%	80.4%	72.1%	49.3%	80.4%
medi-umpurple4		36.5%	27.8%	54.5%	72.1%	48.9%	54.5%
purple1		60.8%	18.8%	100.0%	75.3%	81.2%	100.0%
purple2		56.9%	17.3%	93.3%	75.3%	81.5%	93.3%
purple3		49.0%	14.9%	80.4%	75.3%	81.5%	80.4%
purple4		33.3%	10.2%	54.5%	75.4%	81.3%	54.5%
blueviolet		54.1%	16.9%	88.6%	75.3%	81.0%	88.6%
rebecca-purple		40.0%	20.0%	60.0%	75.0%	66.7%	60.0%
indigo		29.4%	0.0%	51.0%	76.3%	100.0%	51.0%
purple		62.7%	12.5%	94.1%	76.9%	86.7%	94.1%
darkorchid		60.0%	19.6%	80.0%	77.8%	75.5%	80.0%
darko-rchid1		74.9%	24.3%	100.0%	77.8%	75.7%	100.0%
darko-rchid2		69.8%	22.7%	93.3%	77.8%	75.6%	93.3%
darko-rchid3		60.4%	19.6%	80.4%	77.8%	75.6%	80.4%
darko-rchid4		40.8%	13.3%	54.5%	77.8%	75.5%	54.5%
darkviolet		58.0%	0.0%	82.7%	78.4%	100.0%	82.7%
mediu-morchid1		87.8%	40.0%	100.0%	80.0%	60.0%	100.0%
mediu-morchid2		82.0%	37.3%	93.3%	80.0%	60.1%	93.3%
mediu-morchid3		70.6%	32.2%	80.4%	79.9%	60.0%	80.4%
mediu-morchid4		47.8%	21.6%	54.5%	80.0%	60.4%	54.5%
mediu-morchid		72.9%	33.3%	82.7%	80.0%	59.7%	82.7%
plum		86.7%	62.7%	86.7%	83.3%	27.6%	86.7%
plum1		100.0%	73.3%	100.0%	83.3%	26.7%	100.0%
plum2		93.3%	68.2%	93.3%	83.3%	26.9%	93.3%
plum3		80.4%	58.8%	80.4%	83.3%	26.8%	80.4%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
plum4		54.5%	40.0%	54.5%	83.3%	26.6%	54.5%
orchid		85.5%	43.9%	83.9%	84.0%	48.6%	85.5%
orchid4		54.5%	27.8%	53.7%	83.8%	48.9%	54.5%
violet		93.3%	51.0%	93.3%	83.3%	45.4%	93.3%
magenta2		93.3%	0.0%	93.3%	83.3%	100.0%	93.3%
magenta3		80.4%	0.0%	80.4%	83.3%	100.0%	80.4%
fuchsia		100.0%	0.0%	100.0%	83.3%	100.0%	100.0%
thistle		84.7%	74.9%	84.7%	83.3%	11.6%	84.7%
thistle1		100.0%	88.2%	100.0%	83.3%	11.8%	100.0%
thistle2		93.3%	82.4%	93.3%	83.3%	11.8%	93.3%
thistle3		80.4%	71.0%	80.4%	83.3%	11.7%	80.4%
thistle4		54.5%	48.2%	54.5%	83.3%	11.5%	54.5%
webpurple		50.2%	0.0%	50.2%	83.3%	100.0%	50.2%
darkmagenta		54.5%	0.0%	54.5%	83.3%	100.0%	54.5%
orchid1		100.0%	51.4%	98.0%	84.0%	48.6%	100.0%
orchid2		93.3%	47.8%	91.4%	84.1%	48.7%	93.3%
orchid3		80.4%	41.2%	78.8%	84.0%	48.8%	80.4%
maroon1		100.0%	20.4%	70.2%	89.6%	79.6%	100.0%
maroon2		93.3%	18.8%	65.5%	89.6%	79.8%	93.3%
maroon3		80.4%	16.1%	56.5%	89.5%	80.0%	80.4%
maroon4		54.5%	11.0%	38.4%	89.5%	79.9%	54.5%
violetred		81.6%	12.5%	56.5%	89.4%	84.6%	81.6%
mediumvioletred		78.0%	8.2%	52.2%	89.5%	89.4%	78.0%
deeppink		100.0%	7.8%	57.6%	91.0%	92.2%	100.0%
deeppink2		93.3%	7.1%	53.7%	91.0%	92.4%	93.3%
deeppink4		54.5%	3.9%	31.4%	91.0%	92.8%	54.5%
hotpink		100.0%	41.2%	70.6%	91.7%	58.8%	100.0%
hotpink1		100.0%	43.1%	70.6%	92.0%	56.9%	100.0%
hotpink4		54.5%	22.7%	38.4%	91.8%	58.3%	54.5%
deeppink3		80.4%	6.3%	46.3%	91.0%	92.2%	80.4%
hotpink2		93.3%	41.6%	65.5%	92.3%	55.5%	93.3%
hotpink3		80.4%	37.6%	56.5%	92.7%	53.2%	80.4%
violetred1		100.0%	24.3%	58.8%	92.4%	75.7%	100.0%
violetred2		93.3%	22.7%	54.9%	92.4%	75.6%	93.3%
violetred3		80.4%	19.6%	47.1%	92.5%	75.6%	80.4%
violetred4		54.5%	13.3%	32.2%	92.4%	75.5%	54.5%
maroon		69.0%	18.8%	37.6%	93.8%	72.7%	69.0%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
lavenderblush4		54.5%	51.4%	52.5%	93.8%	5.8%	54.5%
lavenderblush		100.0%	94.1%	96.1%	94.4%	5.9%	100.0%
lavenderblush2		93.3%	87.8%	89.8%	94.0%	5.9%	93.3%
lavenderblush3		80.4%	75.7%	77.3%	94.4%	5.9%	80.4%
palevioletred		85.9%	43.9%	57.6%	94.5%	48.9%	85.9%
palevioletred1		100.0%	51.0%	67.1%	94.5%	49.0%	100.0%
palevioletred2		93.3%	47.5%	62.4%	94.6%	49.2%	93.3%
palevioletred3		80.4%	40.8%	53.7%	94.6%	49.3%	80.4%
palevioletred4		54.5%	27.8%	36.5%	94.6%	48.9%	54.5%
pink1		100.0%	71.0%	77.3%	96.4%	29.0%	100.0%
pink2		93.3%	66.3%	72.2%	96.4%	29.0%	93.3%
pink3		80.4%	56.9%	62.0%	96.4%	29.3%	80.4%
pink4		54.5%	38.8%	42.4%	96.2%	28.8%	54.5%
crimson		86.3%	7.8%	23.5%	96.7%	90.9%	86.3%
pink		100.0%	75.3%	79.6%	97.1%	24.7%	100.0%
lightpink		100.0%	71.4%	75.7%	97.5%	28.6%	100.0%
lightpink1		100.0%	68.2%	72.5%	97.7%	31.8%	100.0%
lightpink2		93.3%	63.5%	67.8%	97.6%	31.9%	93.3%
lightpink3		80.4%	54.9%	58.4%	97.7%	31.7%	80.4%
lightpink4		54.5%	37.3%	39.6%	97.7%	31.7%	54.5%
black		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
gray1		1.2%	1.2%	1.2%	0.0%	0.0%	1.2%
gray2		2.0%	2.0%	2.0%	0.0%	0.0%	2.0%
gray3		3.1%	3.1%	3.1%	0.0%	0.0%	3.1%
gray4		3.9%	3.9%	3.9%	0.0%	0.0%	3.9%
gray5		5.1%	5.1%	5.1%	0.0%	0.0%	5.1%
gray6		5.9%	5.9%	5.9%	0.0%	0.0%	5.9%
gray7		7.1%	7.1%	7.1%	0.0%	0.0%	7.1%
gray8		7.8%	7.8%	7.8%	0.0%	0.0%	7.8%
gray9		9.0%	9.0%	9.0%	0.0%	0.0%	9.0%
gray10		10.2%	10.2%	10.2%	0.0%	0.0%	10.2%
gray11		11.0%	11.0%	11.0%	0.0%	0.0%	11.0%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
gray12		12.2%	12.2%	12.2%	0.0%	0.0%	12.2%
gray13		12.9%	12.9%	12.9%	0.0%	0.0%	12.9%
gray14		14.1%	14.1%	14.1%	0.0%	0.0%	14.1%
gray15		14.9%	14.9%	14.9%	0.0%	0.0%	14.9%
gray16		16.1%	16.1%	16.1%	0.0%	0.0%	16.1%
gray17		16.9%	16.9%	16.9%	0.0%	0.0%	16.9%
gray18		18.0%	18.0%	18.0%	0.0%	0.0%	18.0%
gray19		18.8%	18.8%	18.8%	0.0%	0.0%	18.8%
gray20		20.0%	20.0%	20.0%	0.0%	0.0%	20.0%
gray21		21.2%	21.2%	21.2%	0.0%	0.0%	21.2%
gray22		22.0%	22.0%	22.0%	0.0%	0.0%	22.0%
gray23		23.1%	23.1%	23.1%	0.0%	0.0%	23.1%
gray24		23.9%	23.9%	23.9%	0.0%	0.0%	23.9%
gray25		25.1%	25.1%	25.1%	0.0%	0.0%	25.1%
gray26		25.9%	25.9%	25.9%	0.0%	0.0%	25.9%
gray27		27.1%	27.1%	27.1%	0.0%	0.0%	27.1%
gray28		27.8%	27.8%	27.8%	0.0%	0.0%	27.8%
gray29		29.0%	29.0%	29.0%	0.0%	0.0%	29.0%
gray30		30.2%	30.2%	30.2%	0.0%	0.0%	30.2%
gray31		31.0%	31.0%	31.0%	0.0%	0.0%	31.0%
gray32		32.2%	32.2%	32.2%	0.0%	0.0%	32.2%
gray33		32.9%	32.9%	32.9%	0.0%	0.0%	32.9%
gray34		34.1%	34.1%	34.1%	0.0%	0.0%	34.1%
gray35		34.9%	34.9%	34.9%	0.0%	0.0%	34.9%
gray36		36.1%	36.1%	36.1%	0.0%	0.0%	36.1%
gray37		36.9%	36.9%	36.9%	0.0%	0.0%	36.9%
gray38		38.0%	38.0%	38.0%	0.0%	0.0%	38.0%
gray39		38.8%	38.8%	38.8%	0.0%	0.0%	38.8%
gray40		40.0%	40.0%	40.0%	0.0%	0.0%	40.0%
dimgray		41.2%	41.2%	41.2%	0.0%	0.0%	41.2%
gray42		42.0%	42.0%	42.0%	0.0%	0.0%	42.0%
gray43		43.1%	43.1%	43.1%	0.0%	0.0%	43.1%
gray44		43.9%	43.9%	43.9%	0.0%	0.0%	43.9%
gray45		45.1%	45.1%	45.1%	0.0%	0.0%	45.1%
gray46		45.9%	45.9%	45.9%	0.0%	0.0%	45.9%
gray47		47.1%	47.1%	47.1%	0.0%	0.0%	47.1%
gray48		47.8%	47.8%	47.8%	0.0%	0.0%	47.8%
gray49		49.0%	49.0%	49.0%	0.0%	0.0%	49.0%
gray50		49.8%	49.8%	49.8%	0.0%	0.0%	49.8%
webgray		50.2%	50.2%	50.2%	0.0%	0.0%	50.2%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
gray51		51.0%	51.0%	51.0%	0.0%	0.0%	51.0%
gray52		52.2%	52.2%	52.2%	0.0%	0.0%	52.2%
gray53		52.9%	52.9%	52.9%	0.0%	0.0%	52.9%
gray54		54.1%	54.1%	54.1%	0.0%	0.0%	54.1%
gray55		54.9%	54.9%	54.9%	0.0%	0.0%	54.9%
gray56		56.1%	56.1%	56.1%	0.0%	0.0%	56.1%
gray57		56.9%	56.9%	56.9%	0.0%	0.0%	56.9%
gray58		58.0%	58.0%	58.0%	0.0%	0.0%	58.0%
gray59		58.8%	58.8%	58.8%	0.0%	0.0%	58.8%
gray60		60.0%	60.0%	60.0%	0.0%	0.0%	60.0%
gray61		61.2%	61.2%	61.2%	0.0%	0.0%	61.2%
gray62		62.0%	62.0%	62.0%	0.0%	0.0%	62.0%
gray63		63.1%	63.1%	63.1%	0.0%	0.0%	63.1%
gray64		63.9%	63.9%	63.9%	0.0%	0.0%	63.9%
gray65		65.1%	65.1%	65.1%	0.0%	0.0%	65.1%
gray66		65.9%	65.9%	65.9%	0.0%	0.0%	65.9%
darkgray		66.3%	66.3%	66.3%	0.0%	0.0%	66.3%
gray67		67.1%	67.1%	67.1%	0.0%	0.0%	67.1%
gray68		67.8%	67.8%	67.8%	0.0%	0.0%	67.8%
gray69		69.0%	69.0%	69.0%	0.0%	0.0%	69.0%
gray70		70.2%	70.2%	70.2%	0.0%	0.0%	70.2%
gray71		71.0%	71.0%	71.0%	0.0%	0.0%	71.0%
gray72		72.2%	72.2%	72.2%	0.0%	0.0%	72.2%
gray73		72.9%	72.9%	72.9%	0.0%	0.0%	72.9%
gray74		74.1%	74.1%	74.1%	0.0%	0.0%	74.1%
gray		74.5%	74.5%	74.5%	0.0%	0.0%	74.5%
gray75		74.9%	74.9%	74.9%	0.0%	0.0%	74.9%
silver		75.3%	75.3%	75.3%	0.0%	0.0%	75.3%
gray76		76.1%	76.1%	76.1%	0.0%	0.0%	76.1%
gray77		76.9%	76.9%	76.9%	0.0%	0.0%	76.9%
gray78		78.0%	78.0%	78.0%	0.0%	0.0%	78.0%
gray79		78.8%	78.8%	78.8%	0.0%	0.0%	78.8%
gray80		80.0%	80.0%	80.0%	0.0%	0.0%	80.0%
gray81		81.2%	81.2%	81.2%	0.0%	0.0%	81.2%
gray82		82.0%	82.0%	82.0%	0.0%	0.0%	82.0%
lightgray		82.7%	82.7%	82.7%	0.0%	0.0%	82.7%
gray83		83.1%	83.1%	83.1%	0.0%	0.0%	83.1%
gray84		83.9%	83.9%	83.9%	0.0%	0.0%	83.9%
gray85		85.1%	85.1%	85.1%	0.0%	0.0%	85.1%
gray86		85.9%	85.9%	85.9%	0.0%	0.0%	85.9%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
gainsboro		86.3%	86.3%	86.3%	0.0%	0.0%	86.3%
gray87		87.1%	87.1%	87.1%	0.0%	0.0%	87.1%
gray88		87.8%	87.8%	87.8%	0.0%	0.0%	87.8%
gray89		89.0%	89.0%	89.0%	0.0%	0.0%	89.0%
gray90		89.8%	89.8%	89.8%	0.0%	0.0%	89.8%
gray91		91.0%	91.0%	91.0%	0.0%	0.0%	91.0%
gray92		92.2%	92.2%	92.2%	0.0%	0.0%	92.2%
gray93		92.9%	92.9%	92.9%	0.0%	0.0%	92.9%
gray94		94.1%	94.1%	94.1%	0.0%	0.0%	94.1%
gray95		94.9%	94.9%	94.9%	0.0%	0.0%	94.9%
gray96		96.1%	96.1%	96.1%	0.0%	0.0%	96.1%
gray97		96.9%	96.9%	96.9%	0.0%	0.0%	96.9%
gray98		98.0%	98.0%	98.0%	0.0%	0.0%	98.0%
gray99		98.8%	98.8%	98.8%	0.0%	0.0%	98.8%
gray100		100.0%	100.0%	100.0%	0.0%	0.0%	100.0%

CHAPTER

FOUR

KEYBOARD INPUT

Python's built-in `input()` function is great for simple prompts, but it has one limitation: it waits for the Enter key. This makes it unsuitable for interactive applications that need to respond to individual keystrokes, arrow keys, or function keys.

Blessed provides a solution with *`inkey()`*, which returns keystrokes *as they are pressed*, as *`Keystroke`* objects.

4.1 Overview

The *`cbreak()`* context manager enables immediate key detection.

The *`inkey()`* method returns a *`Keystroke`* object representing the key that was immediately pressed.

4.2 Getting Started

Here's a simple example that reads a single keystroke:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()
with term.cbreak():
    key = term.inkey()
    print(f"You pressed: {key!r}")
```

The *`inkey()`* method also accepts a `timeout` parameter that defaults to 1 second. When `timeout` is exceeded without input, an empty *`Keystroke`* is returned, `''`.

In this example, a 10Hz animation is displayed by `timeout=0.1`, stopped by pressing any key:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()
print("Cross animation, press any key to stop: ", end="", flush=True)
with term.cbreak(), term.hidden_cursor():
    cross = '|'

    while True:
        key = term.inkey(timeout=0.1)
        if key:
            print(f'STOP by {key!r}')
```

(continues on next page)

(continued from previous page)

```
            break

        cross = {'|': '-', '-': '|'}[cross]
        print(f'{cross}\b', end='', flush=True)
```

4.3 Keystroke

The *Keystroke* class makes it easy to work with keyboard input. It inherits from `str`, so you can compare it directly to other strings, but it also provides special properties for detecting modifier keys and special sequences.

- Use *is_sequence* to detect special keys
- Use *name* to identify special keys by name (e.g., `KEY_F1`, `KEY_CTRL_Q`)
- Or, by using magic methods like `keystroke.is_f1()` or `keystroke.is_key_ctrl('q')`.

Be careful printing *Keystroke* objects directly. Our examples uses format string, `f'{ks!r}'` for `repr()`, because `str(ks)` may contain control characters or escape sequences beginning with (`KEY_ESCAPE`) and are generally unprintable.

4.3.1 Special Keys

The *is_sequence* property returns `True` for arrow keys, function keys, and any character key combined with modifiers (Ctrl, Alt, Shift).

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()
with term.cbreak():
    key = term.inkey()

    if key.is_sequence:
        print(f"Special key: {key.name} ({key!r})")
    else:
        print(f"Regular character: {key}")
```

The `str(key)` value for *Keystroke* should not be directly printed when *is_sequence* is True as done in this and other examples.

The *name* property provides a readable name for special keys, and can be used for basic equality tests, like in this "paint by arrow key" example:

```
#!/usr/bin/env python3
from blessed import Terminal

header_msg = "Press arrow keys (or 'q' to quit): "
term = Terminal()
position = [term.height // 2, term.width // 2]
with term.cbreak(), term.fullscreen(), term.hidden_cursor():
    print(term.home + header_msg + term.clear_eos)

    while True:
        # show arrow-controlled block
```

(continues on next page)

(continued from previous page)

```
        print(term.move_yx(*position) + '', end='', flush=True)

        # get key,
        key = term.inkey()

        # take action,
        if key == 'q':
            break
        if key.name == 'KEY_UP':
            position[0] = max(0, position[0] - 1)
        elif key.name == 'KEY_LEFT':
            position[1] = max(0, position[1] - 1)
        elif key.name == 'KEY_DOWN':
            position[0] = min(term.height, position[0] + 1)
        elif key.name == 'KEY_RIGHT':
            position[1] = min(term.width, position[1] + 1)
```

Common key names include:

- `KEY_UP`, `KEY_DOWN`, `KEY_LEFT`, `KEY_RIGHT` - Arrow keys
- `KEY_ENTER` - Enter/Return key
- `KEY_BACKSPACE`, `KEY_DELETE` - Backspace and Delete keys
- `KEY_TAB` - Tab key
- `KEY_ESCAPE` - Escape key
- `KEY_F1` through `KEY_F12` - Function keys
- `KEY_PGUP`, `KEY_PGDOWN` - Page Up and Page Down
- `KEY_HOME`, `KEY_END` - Home and End keys

For regular characters without modifiers, *`name`* returns `None` in legacy mode. In *Kitty keyboard protocol* mode, all ASCII alphanumeric and punctuation keys receive synthesized names – see *Key Name Synthesis*.

Feel free to try the demonstration program, *keymatrix.py* to experiment with possible keyboard inputs and combinations.

4.3.2 Modifiers

Alphanumeric keys with modifiers follow the pattern:

- `KEY_CTRL_A`
- `KEY_ALT_Q`
- `KEY_CTRL_ALT_Y`

For alphanumeric keys, `SHIFT` is **not** represented in the key name. Instead, case is handled through the character itself and the `ignore_case` parameter in *Magic Methods*. Control characters are case-insensitive at the protocol level (`Ctrl+A` and `Ctrl+Shift+A` are identical). For Alt modifiers, use `ignore_case=False` to distinguish case: `key.is_alt('a', ignore_case=False)` matches only lowercase, while `key.is_alt('A', ignore_case=False)` matches only uppercase.

Application keys (arrows, function keys, etc.) support `SHIFT` in their names:

- `KEY_SHIFT_LEFT`
- `KEY_CTRL_ALT_BACKSPACE`

- KEY_ALT_DELETE
- KEY_CTRL_SHIFT_F3
- KEY_CTRL_ALT_SHIFT_F9

When multiple modifiers are specified in application key names, they are always in the following order:

- CTRL
- ALT
- SHIFT

The escape sequence '\x1b[' is always decoded as name CSI in legacy mode when it arrives without any known matching sequence.

The *value* property returns the text character for keys that produce text, stripping away modifier information.

Special keys like KEY_UP, KEY_F1, KEY_BACKSPACE have an empty *value* string.

Control characters like KEY_CTRL_C are *value* c (lowercase). Similarly for alt, KEY_ALT_a is *value* a.

4.3.3 Event Types

The *Keystroke* class provides properties about key events:

- *pressed* - True if this is a key press event
- *repeated* - True if this is a key repeat event
- *released* - True if this is a key release event

For press/release tracking, two additional properties provide stable identifiers across event types:

- *key_name* - like *name*, but without the _RELEASED or _REPEATED suffix, so that the same key always returns the same name regardless of event type.
- *key_value* - like *value*, but returns the character even for release events (where *value* returns '').

Note: These event types can only be distinguished when using the *Kitty Keyboard Protocol*. Without it, all keystrokes will have pressed=True and repeated=False, released=False.

See *Kitty Keyboard Protocol* for more information about enabling and using the Kitty Keyboard Protocol.

4.3.4 Magic Methods

The *Keystroke* class provides convenient "magic methods" for checking keys with modifiers. These methods all start with is_:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()
print("Press 'q' or F10 to exit! Press F1 for help")
with term.cbreak():
    while True:
        key = term.inkey()

        # Check for specific character with modifier
        if key.is_ctrl('q') or key.is_f10():
            print(f"Exit by key named {key.name}")
            break
```

(continues on next page)

(continued from previous page)

```
        # Check for function key
        elif key.is_f1():
            print("* don't panic")
```

Some examples, given *key* object of *Keystroke*:

- `key.is_ctrl('x')`
- `key.is_alt('q')`
- `key.is_ctrl_alt('s')`
- `key.is_ctrl_shift_alt('a')`
- `key.is_f1()`
- `key.is_up()`
- `key.is_enter()`
- `key.is_backspace()`
- `key.is_ctrl_left()`
- `key.is_alt_backspace()`
- `key.is_shift_f5()`

By default, character matching is case-insensitive. You can change this with the `ignore_case` parameter. For example, "Alt" with capital letter `U` matches both methods:

- `key.is_alt('u')`
- `key.is_alt_shift('u')`

To explicitly match only "Alt + u" (lowercase 'U'), set `ignore_case` argument to False:

- `key.is_alt('u', ignore_case=False)`

4.4 All Names

These are duplicated from the key names found in curses(3), or those constants in `curses` beginning with phrase *KEY_*, as follows:

Table 1: All Terminal class attribute Keyboard codes, by name

Name	Value	Example Sequence(s)
KEY_BACKSPACE	263	'\x08', '\x7f'
KEY_BEGIN	354	
KEY_BTAB	353	
KEY_CANCEL	355	
KEY_CAPS_LOCK	57358	
KEY_CATAB	342	
KEY_CENTER	350	'\x1b[E'
KEY_CLEAR	333	
KEY_CLOSE	356	
KEY_COMMAND	357	
KEY_COPY	358	

continues on next page

Table 1 – continued from previous page

Name	Value	Example Sequence(s)
KEY_CREATE	359	
KEY_CTAB	341	
KEY_DELETE	330	‘\x1b[3~’
KEY_DL	328	
KEY_DOWN	258	‘\x1b[B’, ‘\x1b[OB’
KEY_DOWN_LEFT	351	
KEY_DOWN_RIGHT	352	
KEY_EIC	332	
KEY_END	360	‘\x1b[F’, ‘\x1b[K’, ‘\x1b[4~’, ‘\x1b[8~’, ‘\x1b[OF’
KEY_ENTER	343	‘\n’, ‘\r’, ‘\x1bOM’
KEY_EOL	335	
KEY_EOS	334	
KEY_ESCAPE	361	‘\x1b’
KEY_F0	264	
KEY_F1	265	‘\x1bOP’, ‘\x1b[P’
KEY_F10	274	
KEY_F11	275	
KEY_F12	276	
KEY_F13	57376	
KEY_F14	57377	
KEY_F15	57378	
KEY_F16	57379	
KEY_F17	57380	
KEY_F18	57381	
KEY_F19	57382	
KEY_F2	266	‘\x1bOQ’, ‘\x1b[Q’
KEY_F20	57383	
KEY_F21	57384	
KEY_F22	57385	
KEY_F23	57386	
KEY_F3	267	‘\x1bOR’, ‘\x1b[13~’
KEY_F4	268	‘\x1bOS’, ‘\x1b[S’
KEY_F5	269	
KEY_F6	270	
KEY_F7	271	
KEY_F8	272	
KEY_F9	273	
KEY_FIND	362	
KEY_HELP	363	
KEY_HOME	262	‘\x1b[H’, ‘\x1b[1~’, ‘\x1b[7~’, ‘\x1b[OH’
KEY_IL	329	
KEY_INSERT	331	‘\x1b[2~’
KEY_ISO_LEVEL3_SHIFT	57453	
KEY_ISO_LEVEL5_SHIFT	57454	
KEY_KP_0	520	‘\x1bOp’
KEY_KP_1	521	‘\x1bOq’
KEY_KP_2	522	‘\x1bOr’
KEY_KP_3	523	‘\x1bOs’

continues on next page

Table 1 – continued from previous page

Name	Value	Example Sequence(s)
KEY_KP_4	524	'\x1bOt'
KEY_KP_5	525	'\x1bOu'
KEY_KP_6	526	'\x1bOv'
KEY_KP_7	527	'\x1bOw'
KEY_KP_8	528	'\x1bOx'
KEY_KP_9	529	'\x1bOy'
KEY_KP_ADD	514	'\x1bOk'
KEY_KP_BEGIN	57427	
KEY_KP_DECIMAL	517	'\x1bOn'
KEY_KP_DELETE	57426	
KEY_KP_DIVIDE	518	'\x1bOo'
KEY_KP_DOWN	57420	
KEY_KP_END	57424	
KEY_KP_ENTER	57414	
KEY_KP_EQUAL	519	'\x1bOX'
KEY_KP_HOME	57423	
KEY_KP_INSERT	57425	
KEY_KP_LEFT	57417	
KEY_KP_MULTIPLY	513	'\x1bOj'
KEY_KP_PAGE_DOWN	57422	
KEY_KP_PAGE_UP	57421	
KEY_KP_RIGHT	57418	
KEY_KP_SEPARATOR	515	'\x1bOl'
KEY_KP_SUBTRACT	516	'\x1bOm'
KEY_KP_UP	57419	
KEY_LEFT	260	'\x1b[D', '\x1b[OD'
KEY_LEFT_ALT	57443	
KEY_LEFT_CONTROL	57442	
KEY_LEFT_HYPER	57445	
KEY_LEFT_META	57446	
KEY_LEFT_SHIFT	57441	
KEY_LEFT_SUPER	57444	
KEY_LL	347	
KEY_LOWER_VOLUME	57438	
KEY_MARK	364	
KEY_MAX	511	
KEY_MEDIA_FAST_FORWARI	57433	
KEY_MEDIA_PAUSE	57429	
KEY_MEDIA_PLAY	57428	
KEY_MEDIA_PLAY_PAUSE	57430	
KEY_MEDIA_RECORD	57437	
KEY_MEDIA_REVERSE	57431	
KEY_MEDIA_REWIND	57434	
KEY_MEDIA_STOP	57432	
KEY_MEDIA_TRACK_NEXT	57435	
KEY_MEDIA_TRACK_PREVIC	57436	
KEY_MENU	57363	
KEY_MESSAGE	365	
KEY_MIN	257	
KEY_MOUSE	409	

continues on next page

Table 1 – continued from previous page

Name	Value	Example Sequence(s)
KEY_MOVE	366	
KEY_MUTE_VOLUME	57440	
KEY_NEXT	367	
KEY_NUM_LOCK	57360	
KEY_OPEN	368	
KEY_OPTIONS	369	
KEY_PAUSE	57362	
KEY_PGDOWN	338	'\x1b[U', '\x1b[6~'
KEY_PGUP	339	'\x1b[V', '\x1b[5~'
KEY_PREVIOUS	370	
KEY_PRINT	346	
KEY_PRINT_SCREEN	57361	
KEY_RAISE_VOLUME	57439	
KEY_REDO	371	
KEY_REFERENCE	372	
KEY_REFRESH	373	
KEY_REPLACE	374	
KEY_RESET	345	
KEY_RESIZE	410	
KEY_RESTART	375	
KEY_RESUME	376	
KEY_RIGHT	261	'\x1b[C', '\x1b[OC'
KEY_RIGHT_ALT	57449	
KEY_RIGHT_CONTROL	57448	
KEY_RIGHT_HYPER	57451	
KEY_RIGHT_META	57452	
KEY_RIGHT_SHIFT	57447	
KEY_RIGHT_SUPER	57450	
KEY_SAVE	377	
KEY_SBEG	378	
KEY_SCANCEL	379	
KEY_SCOMMAND	380	
KEY_SCOPY	381	
KEY_SCREATE	382	
KEY_SCROLL_LOCK	57359	
KEY_SDC	383	
KEY_SDL	384	
KEY_SDOWN	336	'\x1b[1;2B'
KEY_SELECT	385	
KEY_SEND	386	
KEY_SEOL	387	
KEY_SEXIT	388	
KEY_SFIND	389	
KEY_SHELP	390	
KEY_SHOME	391	
KEY_SIC	392	
KEY_SLEFT	393	'\x1b[1;2D'
KEY_SMESSAGE	394	
KEY_SMOVE	395	
KEY_SNEXT	396	

continues on next page

Table 1 – continued from previous page

Name	Value	Example Sequence(s)
KEY_SOPTIONS	397	
KEY_SPREVIOUS	398	
KEY_SPRINT	399	
KEY_SREDO	400	
KEY_SREPLACE	401	
KEY_SRESET	344	
KEY_SRIGHT	402	'\x1b[1;2C'
KEY_SRSUME	403	
KEY_SSAVE	404	
KEY_SSUSPEND	405	
KEY_STAB	340	
KEY_SUNDO	406	
KEY_SUP	337	'\x1b[1;2A'
KEY_SUSPEND	407	
KEY_TAB	512	'\t'
KEY_UNDO	408	
KEY_UP	259	'\x1b[A', '\x1b[OA'
KEY_UP_LEFT	348	
KEY_UP_RIGHT	349	

However, these keys do not represent the full range of keys that can be detected with their modifiers, such as `KEY_CTRL_LEFT` is not matched by any Keycode constant, but rather a combination of existing `KEY_LEFT` with the `CTRL` modifier.

For Legacy API of classic curses applications, *code* may be be compared with attributes of *Terminal*, which are duplicated from those found in curses(3), or those constants in `curses` beginning with phrase *KEY_*. These have numeric values that can be used for all basic application keys.

4.5 Async Input

The *async_inkey()* method is an asyncio-compatible version of *inkey()*. It yields control to the event loop while waiting for input, making it suitable for use with `async`/`await`.

```
import asyncio
from blessed import Terminal

async def main():
    term = Terminal()
    with term.cbreak():
        key = await term.async_inkey(timeout=5.0)
        if key:
            print(f"You pressed: {key!r}")
        else:
            print("Timed out")

asyncio.run(main())
```

The method accepts the same `timeout` and `esc_delay` parameters as *inkey()*. It must be called within a *cbreak()* or *raw()* context.

For a complete example using `async_inkey` with the *Line Editor*, see *Line Editor*.

CHAPTER

FIVE

KITTY KEYBOARD PROTOCOL

The Kitty Keyboard Protocol provides enhanced keyboard input capabilities.

It allows your application to distinguish between some special kinds of keys, between key press, repeat, and release events, and provides improved support for modifiers and special characters.

The Kitty keyboard protocol is known to be supported by many popular terminals: Kitty, alacritty, foot, ghostty, iTerm2, rio, WezTerm.

On terminals that do not support the protocol, the *enable_kitty_keyboard()* context manager gracefully does nothing, and keyboard input continues to work normally, application code does not need to change.

5.1 Overview

This protocol is designed mainly to resolve ambiguity, such as those related to the Escape key, application keys, Control keys, and Alt+Key sequences. For example, Ctrl+I and TAB key both send `\t`, but, with disambiguation mode enabled, these are detected separately with unique sequences and names, `KEY_TAB` and `KEY_CTRL_I`.

The protocol is automatically detected and enabled when you use the *enable_kitty_keyboard()* context manager. If the terminal doesn't support it, your code continues to work normally using standard keyboard input.

Unlike standard keyboard input where *name* is `None` for plain characters, the Kitty protocol synthesizes names for all ASCII alphanumeric and punctuation keys. See *Key Name Synthesis* for details.

5.2 Getting Started

Here's a simple example showing key press, repeat, and release events:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

print("Press and hold keys to see raw kitty keystrokes and their names (press 'q' to␣
↪quit)")
with term.enable_kitty_keyboard(report_events=True, report_all_keys=True, report_
↪alternates=True):
    print("kitty keyboard state:", term.get_kitty_keyboard_state())
    with term.raw():
        while True:
            key = term.inkey()

            kind = ("pressed" if key.pressed
```

(continues on next page)

(continued from previous page)

```
                    else "repeated" if key.repeated
                    else "released")
            if key.pressed and key.value == 'q':
                break
            print(f"name={key.key_name} value={key.key_value} kind={kind}, "
                  f"sequence={key!r}", end="\r\n")
```

In this example, pressing and holding a key will show "pressed" once, then multiple "repeating" messages while held, and finally "released" when you let go. On terminals that don't support the Kitty protocol, you'll only see "pressed" events.

5.3 Event Types

When `report_events=True` is enabled, the *Keystroke* class provides three properties to distinguish between key event types:

- *pressed* - `True` if this is a key press event
- *repeated* - `True` if this is a key repeat event (repeat events issued by OS when held down)
- *released* - `True` if this is a key release event

These properties are only meaningful when Kitty keyboard protocol is enabled with `report_events=True`. Without the protocol, all keystrokes will have `pressed=True` and `repeated=False`, `released=False`.

The *name* attribute also includes event type suffixes: `KEY_CTRL_J` for press, `KEY_CTRL_J_REPEATED` for repeat, and `KEY_CTRL_J_RELEASED` for release events.

For press/release tracking (e.g., key maps), use *key_name* instead – it returns the same name regardless of event type (`KEY_CTRL_J` for press, repeat, and release). Similarly, *key_value* returns the character even for release events, where *value* returns `''`.

Note: To distinguish press, repeat, and release for plain text keys like `a` or `5`, you must also enable `report_all_keys=True`. Without it, only release events are encoded with the protocol – press and repeat arrive as plain characters with no event-type information.

Example use case - detect only initial key presses and ignore repeats:

```
from blessed import Terminal

term = Terminal()

with term.enable_kitty_keyboard(report_events=True):
    with term.cbreak():
        while True:
            key = term.inkey()

            # Only respond to initial keypress, ignoring repeat and release
            if key.pressed:
                if key == 'q':
                    break
                print(f"Key {key!r} pressed")
```

5.4 Protocol Features

The *enable_kitty_keyboard()* context manager accepts any combination of the following feature flags as keyword arguments of type bool:

disambiguate
: Enables disambiguated escape codes. With this enabled, pressing the Escape keys, control, and some application keys produce distinct sequences that more correctly identify the user's keypress. For example, Ctrl+I and Tab can be distinguished as `KEY_CTRL_I` and `KEY_TAB`.

report_events
: Reports key repeat and release events in addition to key press events. This allows detecting when keys are held down or released.

report_alternates
: Reports both the shifted and base layout keys for keyboard shortcuts. Useful for handling shortcuts consistently across different keyboard layouts (e.g., matching both Ctrl+Shift+Equal and Ctrl+Plus as the same shortcut).

report_all_keys
: Reports all keys as escape codes, including normal text keys that would normally be sent as plain characters.

report_text
: Reports the associated text with key events (requires `report_all_keys`). This provides the actual character that would be typed alongside the key code.

Basic usage:

```
with term.enable_kitty_keyboard(disambiguate=True, report_events=True):
    # Your code here
    pass
```

Feel free to try the demonstration program, *keymatrix.py* to experiment with combining any or all possible kitty protocol features using Shift+F1 through Shift+F5.

5.5 Key Name Synthesis

The *name* is synthesized for all ASCII alphanumeric keys, punctuation, and symbols while in kitty keyboard mode.

Some example **Alphanumeric keys**:

```
KEY_A, KEY_Z, KEY_0, KEY_9
KEY_CTRL_A, KEY_ALT_5, KEY_CTRL_ALT_SHIFT_M
```

Punctuation and symbols use descriptive names:

```
KEY_SPACE, KEY_PERIOD, KEY_COMMA, KEY_MINUS, KEY_EQUALS
KEY_LEFT_SQUARE_BRACKET, KEY_RIGHT_SQUARE_BRACKET
KEY_SLASH, KEY_BACKSLASH, KEY_SEMICOLON, KEY_APOSTROPHE
KEY_GRAVE_ACCENT, KEY_TILDE, KEY_EXCLAMATION_MARK
KEY_AT, KEY_HASH, KEY_DOLLAR, KEY_PERCENT, KEY_CARET
KEY_AMPERSAND, KEY_ASTERISK, KEY_PLUS, KEY_PIPE
KEY_LEFT_PARENTHESIS, KEY_RIGHT_PARENTHESIS
KEY_LEFT_CURLY_BRACKET, KEY_RIGHT_CURLY_BRACKET
KEY_LESS_THAN, KEY_GREATER_THAN, KEY_QUESTION_MARK
KEY_DOUBLE_QUOTE, KEY_COLON, KEY_UNDERSCORE
```

Modifiers and event-type suffixes combine as expected:

```
KEY_ALT_LEFT_SQUARE_BRACKET
KEY_CTRL_PERIOD
KEY_SPACE_RELEASED
KEY_ALT_MINUS_REPEATED
```

5.5.1 Shifted Key Reporting

The Kitty protocol reports the *base key* and modifiers separately rather than the resulting character. On a US keyboard layout, pressing Shift+3 produces #, but the terminal reports codepoint 51 (digit 3) with the Shift modifier. This means:

- The name is `KEY_SHIFT_3`, not `KEY_HASH`
- The *value* is the actual character #

Symbol names like `KEY_HASH`, `KEY_AMPERSAND`, and `KEY_TILDE` are still reachable when a terminal sends the symbol codepoint directly, but on most layouts Shift+digit combinations will produce names based on the base digit key.

5.5.2 Layout-Independent Key Handling

By default, the protocol reports whichever key the current keyboard layout maps to a given position. This means the keys of a physical position on the keyboard will produce different codepoints on different layouts, QWERTY, Dvorak, AZERTY, or many non-US layouts.

The `report_alternates` flag solves this by requesting the terminal to also report the *base layout key* – the key identity independent of the active layout. When `report_alternates=True`, blessed uses the base layout key for name synthesis, so a shortcut like `KEY_CTRL_Z` works regardless of whether the user's layout places `Z` in the QWERTY position or elsewhere.

```
with term.enable_kitty_keyboard(report_alternates=True):
    with term.cbreak():
        key = term.inkey()
        if key.name == 'KEY_CTRL_Z':
            undo()
```

This is especially useful for applications that bind shortcuts to physical key positions (like Ctrl+Z for undo) rather than to specific characters.

5.6 Compatibility

You can optionally check for protocol support:

```
from blessed import Terminal

term = Terminal()
state = term.get_kitty_keyboard_state()
if state is not None:
    print("Kitty keyboard protocol is supported")
else:
    print("No kitty protocol support.")
```

This check is not necessary but may be useful in some cases.

5.7 Timeout

The `timeout` parameter of *`enable_kitty_keyboard()`* controls how long to await negotiation, in seconds.

When negotiating, both DA1 and Kitty protocol status request sequences are transmitted, making it possible to automatically detect protocol support in almost all cases, but terminals that support neither DA1 or Kitty protocol ("dumb" terminals) will block forever unless `timeout` is set.

5.8 See Also

- *Keyboard Input* - Basic keyboard input handling
- `Terminal.enable_kitty_keyboard()` - Enable Kitty protocol features
- `Terminal.get_kitty_keyboard_state()` - Query current protocol state
- `Keystroke.pressed` - Check if key was pressed
- `Keystroke.repeated` - Check if key is repeating
- `Keystroke.released` - Check if key was released
- `Keystroke.key_name` - Key identity without event-type suffix
- `Keystroke.key_value` - Character for the key, even for release events

CHAPTER

SIX

LINE EDITOR

The *blessed.line_editor* module provides *LineEditor*, a headless single-line editor with readline-style keybindings, grapheme-aware cursor movement, auto-suggest from history, password masking, and horizontal scrolling.

Note

The editor never writes to the terminal directly, your application controls when and where output appears.

Use the built-in *render methods* to produce ready-to-print escape sequences, or read *display* for raw display state if you need fully custom rendering.

6.1 Overview

Feed keystrokes (from *inkey()* or *async_inkey()*) into *feed_key()*. It returns a *LineEditResult* with these fields:

- `line` the accepted string when Enter is pressed, otherwise `None`.
- `interrupt`, `eof`: `True` on Ctrl+C / Ctrl+D respectively.
- `changed` `True` when the display needs redrawing.
- `bell` a bell string to emit (empty when silent).

For bracketed paste, feed the pasted text through *insert_text()* instead of *feed_key()*.

For async usage, simply replace `term.inkey()` with `await term.async_inkey()`. See *Async Input* in the keyboard documentation for more on `async_inkey`.

6.2 Example

A line editor with history, auto-suggest, password mode, and styled rendering:

```
#!/usr/bin/env python3
"""Line editor demo with clipboard, bracketed paste, history, and scrolling."""
from functools import partial

from blessed import Terminal
from blessed.line_editor import LineEditor, LineHistory, LineEditResult

term = Terminal()
history = LineHistory()
```

(continues on next page)

(continued from previous page)

```
echo = partial(print, end='\r\n', flush=True)

has_clipboard = term.does_osc52_clipboard()


def copy_line(ed):
    term.clipboard_copy(ed.line)
    return LineEditResult()


def paste_line(ed):
    text = term.clipboard_paste()
    return ed.insert_text(text) if text else LineEditResult()


with term.raw(), term.cursor_shape(term.CursorShape.BLINKING_BLOCK), term.bracketed_
↪paste():
    term.detect_ambiguous_width()
    if has_clipboard:
        echo("press ^C and ^V for OS clipboard, type 'quit' to exit")
    else:
        echo("type 'quit' to exit")
    echo()
    while True:
        margin = max(1, term.width // 5)
        width = term.width - margin * 2
        prompt = "Prompt> "
        col = margin
        ed_width = width - len(prompt)
        keymap = {}
        if has_clipboard:
            keymap['KEY_CTRL_C'] = copy_line
            keymap['KEY_CTRL_V'] = paste_line
        ed = LineEditor(history=history, max_width=ed_width, limit=200,
                        ambiguous_width=term.ambiguous_width,
                        term_program=term.term_program,
                        bg_sgr=term.on_brown,
                        keymap=keymap or None)
        echo(term.move_x(col) + prompt, end='')
        row = term.get_location()[0]
        ed_col = col + len(prompt)
        echo(ed.render(term, row, ed_width, col=ed_col), end='')
        while True:
            key = term.inkey()
            if key.name == 'BRACKETED_PASTE':
                result = ed.insert_text(key.text)
            else:
                result = ed.feed_key(key)
            if result.bell:
                echo(result.bell, end='')
            if result.changed:
                key_str = str(key)
```

(continues on next page)

(continued from previous page)

```
                out = None
                if key_str and key_str.isprintable() and len(key_str) == 1:
                    out = ed.render_insert(term, row, key_str)
                elif getattr(key, "name", None) == "KEY_BACKSPACE":
                    out = ed.render_backspace(term, row)
                if out is None:
                    out = ed.render(term, row, ed_width, col=ed_col)
                echo(out, end='')
            if result.line is not None:
                echo()
                if result.line:
                    echo(term.move_x(col) + f"  => {result.line!r}")
                break
        if (result.line or '').strip() == 'quit':
            break
```

6.3 History

LineHistory stores command history in memory. History navigation (Up/Down) and auto-suggest (type a prefix, press Right to accept) are enabled automatically when a history instance is attached to the editor.

For on-disk persistence, read/write the `entries` list directly (most recent entry last).

6.4 Display State & Styling

Each call to *display* returns a *DisplayState* with the visible text, cursor position, suggestion suffix, and clipping indicators.

The editor ships with default SGR styling (light cream text, dark suggestion). Override with `text_sgr`, `suggestion_sgr`, `bg_sgr`, or `ellipsis_sgr`:

```
editor = LineEditor(bg_sgr=term.on_brown, max_width=term.width)
```

When `max_width` is set, display width is measured with `wcwidth`; pass *ambiguous_width* and *term_program* so that ambiguous and terminal-corrected characters are measured as the terminal renders them.

When `max_width` is set and text overflows, `overflow_left` and `overflow_right` indicate which edges are truncated. Use `ellipsis_sgr` to style the overflow indicator.

6.5 Rendering Helpers

LineEditor provides three render methods that build complete escape-sequence strings from the current display state:

render()
: Full redraw, always produces correct output.

render_insert()
: Fast-path after a character insert at end of buffer. Returns `None` when a full redraw is needed instead.

render_backspace()
: Fast-path after a backspace at end of buffer. Returns `None` when a full redraw is needed instead.

Try the fast-path first and fall back to `render()` when it returns `None`. See the *example above* for the complete pattern.

6.6 Other Methods

clear()
: Reset the buffer, cursor, and undo history.

set_password_mode()
: Toggle password masking on or off mid-session.

6.7 Constructor Options

Beyond the styling and keybinding options shown above, *LineEditor* accepts:

password
: If `True`, start in password mode (characters are masked). Toggle at runtime with *set_password_mode()*.

password_char
: Replacement character shown in password mode (default `""`).

limit
: Maximum buffer length in characters (default 65536).

limit_bell
: Bell string emitted when the limit is reached (default `"\\a"`).

scroll_jump
: Fraction of `max_width` to scroll when the cursor overflows (default 0.5).

6.8 Custom Keybindings

Pass a `keymap` dict to override or extend the default emacs/readline bindings. Keys are *Keystroke* `.name` strings (e.g. `"KEY_CTRL_K"`), values are callables accepting a *LineEditor* and returning a *LineEditResult*.

Override an existing binding:

```
def my_enter(editor):
    # custom accept logic
    return LineEditResult(line=editor.line, changed=True)

editor = LineEditor(keymap={"KEY_ENTER": my_enter})
```

Add a new binding:

```
def handle_f1(editor):
    editor.insert_text("help!")
    return LineEditResult(changed=True)

editor = LineEditor(keymap={"KEY_F1": handle_f1})
```

Disable a binding by setting it to `None`:

```
# Ctrl+C becomes a silent no-op instead of raising interrupt
editor = LineEditor(keymap={"KEY_CTRL_C": None})
```

6.9 Default Keybindings

These are the default emacs/readline bindings. All can be overridden via `keymap`.

Key	Action
Enter	Accept line
Ctrl+C	Cancel (interrupt)
Ctrl+D	EOF on empty line, delete at cursor otherwise
Left, Ctrl+B	Move cursor left
Right	Accept auto-suggest at end, otherwise move right
Ctrl+F	Move cursor right
Home, Ctrl+A	Move to start of line
End, Ctrl+E	Move to end of line
Shift+Left, Ctrl+Left	Move word left
Shift+Right, Ctrl+Right	Move word right
Backspace	Delete character before cursor
Delete	Delete character at cursor
Ctrl+K	Kill to end of line
Ctrl+U	Kill to start of line
Ctrl+W	Kill word backward
Ctrl+Y	Yank (paste from kill ring)
Up, Ctrl+P	Previous history entry
Down, Ctrl+N	Next history entry
Ctrl+Z	Undo

CHAPTER

SEVEN

MOUSE INPUT

Blessed supports mouse input in the terminal! Your applications can respond to clicks, drags, scroll wheel, or track live mouse cursor movement, even at the pixel-level, for creating interactive games and apps.

Overview:

- Check for support using `does_mouse()`
- Enable mouse input with `mouse_enabled()`
- Receive events through `inkey()`

Mouse events work seamlessly with keyboard events - both come through the same `inkey()` method.

7.1 Getting Started

Here is a basic example:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

if not term.does_mouse():
    print("This example won't work on your terminal!")
else:
    print("Click anywhere! ^C to quit")
    with term.cbreak(), term.mouse_enabled():
        while True:
            inp = term.inkey()
            if inp.name and inp.name.startswith('MOUSE_'):
                print(f"button {inp.name} at (y={inp.y}, x={inp.x})")
```

The *mouse_enabled()* context manager enables mouse tracking and automatically disables it when done. Mouse events arrive through *inkey()* just like keyboard events.

Mouse events can be detected by checking if *name* starts with `'MOUSE_'`. The name includes the button and any modifiers, such as `'MOUSE_LEFT'`, `'MOUSE_CTRL_LEFT'`, or `'MOUSE_SCROLL_UP'`. You can also use magic method predicates like `inp.is_mouse_left()`.

Note

Mouse coordinates are **0-indexed**, matching blessed's terminal movement functions like *move_yx()*. The top-left corner is `(y=0, x=0)`, not `(1, 1)`. This allows direct use of mouse coordinates with movement functions.

7.2 Understanding Buttons

Mouse events come through *inkey()* just like keyboard events.

You can detect mouse events using either the *name* property or magic method predicates:

Using the name property:

The *name* returns button names with the MOUSE_ prefix, following the pattern MOUSE_[MODIFIERS_]BUTTON[_RELEASED]:

```
inp = term.inkey()
if inp.name == 'MOUSE_LEFT':
    print("Left button pressed")
```

Using magic method predicates:

```
inp = term.inkey()
if inp.is_mouse_left():
    print("Left button pressed")
elif inp.is_mouse_ctrl_left():
    print("Ctrl+Left button pressed")
```

Button names include:

- Basic events: MOUSE_LEFT, MOUSE_MIDDLE, MOUSE_RIGHT, MOUSE_SCROLL_UP, MOUSE_SCROLL_DOWN
- Release events: MOUSE_LEFT_RELEASED, MOUSE_MIDDLE_RELEASED, MOUSE_RIGHT_RELEASED, or plain MOUSE_RELEASED when the terminal does not say which button was released, see *Legacy Mouse Protocols*
- With modifiers: MOUSE_CTRL_LEFT, MOUSE_SHIFT_SCROLL_UP, MOUSE_META_RIGHT, MOUSE_CTRL_SHIFT_META_MIDDLE
- Extended buttons: MOUSE_BUTTON_6 through MOUSE_BUTTON_11, reported by mice with thumb or tilt buttons

Modifiers are included in order CTRL, SHIFT, and META

In this example, all possible combinations may be entered and recorded, see if you have enough fingers for CTRL_SHIFT_META_MIDDLE, imagine the possibilities!

```
#!/usr/bin/env python
from blessed import Terminal

term = Terminal()
counts = dict()

with term.fullscreen(), term.cbreak(), term.mouse_enabled():
    while True:
        inp = term.inkey(timeout=None)

        # Check if this is a mouse event
        if inp.name and inp.name.startswith('MOUSE_'):
            # Use the keystroke name for button identification
            counts[inp.name] = counts.get(inp.name, 0) + 1
            with term.synchronized_output():
                print(term.home + term.clear)
                print(term.bold("Mouse Modifier Example, press Ctrl+C to quit"))
                print()
```

(continues on next page)

(continued from previous page)

```
                # Display the most recent event
                print(f"button={inp.name} at (y={inp.y}, x={inp.x})")
                print()

                # Display totals
                print("Totals: ")
                for button_name, count in sorted(counts.items()):
                    print(f"{button_name}: {count}")
```

7.3 Understanding Mouse Coordinates

Mouse coordinates are accessed by `mouse_yx` attribute in as tuple `(int, int)` of the corresponding row and column. This matches the signature of `move_yx()`. If `mouse_yx` is used on a keystroke that is not a mouse event, values `(-1, -1)` are returned.

There is also a `mouse_xy` attribute that mirrors the signature of `move_xy()`.

- *`mouse_yx`* - position as a `(y, x)` tuple
- *`mouse_xy`* - position as an `(x, y)` tuple

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

if not term.does_mouse():
    print("This example won't work on your terminal!")
else:
    with term.cbreak(), term.fullscreen(), term.mouse_enabled(report_drag=True):
        print("Click to move cursor! ^C to quit")
        while True:
            inp = term.inkey()
            if inp.name and inp.name.startswith('MOUSE_'):
                print(term.move_yx(*inp.mouse_yx), end='', flush=True)
```

7.4 Checking Support

Not all terminals support mouse tracking or all kinds of mouse tracking.

Use `does_mouse()` to check before enabling:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

# check basic mouse support
if not term.does_mouse():
    print(f"mouse_enabled() {term.bright_red('not supported')} on this Terminal")
else:
```

(continues on next page)

(continued from previous page)

```
    # check for, enable, and report all supported advanced features
    feature_kwargs = {mouse_feature: True
                      for mouse_feature in ('report_pixels', 'report_drag', 'report_
↪motion')
                      if term.does_mouse(**{mouse_feature: True})}
    with term.mouse_enabled(**feature_kwargs):
        print(f"mouse_enabled({', '.join(feature_kwargs)}) enabled")
```

The *does_mouse()* method accepts the same parameters as *mouse_enabled()* and returns `True` if all of given modes are supported.

> **Note**
>
> On Windows, *does_mouse()* always returns `True` because the native console API provides mouse events on all versions. Native `MOUSE_EVENT` records are automatically converted to the same event format used on Unix, so the same code works on all platforms.

7.5 Using mouse_enabled()

The `mouse_enabled()` context manager enables the appropriate *DEC Private Modes* depending on the simplified arguments given.

`mouse_enabled()` accepts these keyword-only parameters:

- `clicks=True` - Enable basic click reporting (default).
- `report_drag=False` - Report motion while a button is held.
- `report_motion=False` - Report all mouse movement.
- `report_pixels=False` - Report position in pixels instead of cells.
- `timeout=1.0` - Timeout for mode queries, in seconds.

Parameter Precedence

The tracking modes have precedence: `report_motion` > `report_drag` > `clicks`. When you enable a higher-precedence mode, it automatically includes the functionality of lower modes. For example, `report_motion=True` will also track drags and clicks.

7.5.1 report_drag

Reports motion only while a button is held down:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

if not term.does_mouse(report_drag=True):
    print("This example won't work on your terminal!")
else:
    with term.cbreak(), term.mouse_enabled(report_drag=True):
        while True:
```

(continues on next page)

(continued from previous page)

```
            inp = term.inkey()
            if inp.name and inp.name.endswith('_MOTION'):
                print(f"Drag event at ({inp.mouse_yx})")
```

When using `report_drag=True` or `report_motion=True`, you'll receive motion events in the `name` attribute with a `_MOTION` suffix:

- `MOUSE_MOTION` - Motion without any button pressed (`report_motion`)
- `MOUSE_LEFT_MOTION`, `MOUSE_MIDDLE_MOTION`, `MOUSE_RIGHT_MOTION` - Dragging with a button held, usually follows click, eg. `MOUSE_LEFT`.

Motion events include modifiers just like click events, for example `MOUSE_CTRL_LEFT_MOTION`.

7.5.2 report_motion

Reports all mouse clicks movement, even without buttons pressed. The `name` attribute of `MOUSE_MOTION` is given when no button or scroll wheel events have occurred, only an updated `mouse_yx` position.

In this example, the terminal cursor tracks with the mouse pointer because the `move_yx()` sequence is displayed following any mouse event, especially `MOUSE_MOTION`, tracking the `Keystroke.mouse_yx` coordinate.

```
#!/usr/bin/env python
from blessed import Terminal

term = Terminal()

if not term.does_mouse(report_motion=True):
    print("This terminal does not support mouse motion tracking!")
else:
    # Track current color for painting
    color_idx = 7
    num_colors = min(256, term.number_of_colors)
    header = "Mouse wheel sets color=[{0}], LEFT button paints, RIGHT erases, ^C:quit"

    def make_header():
        return term.home + term.center(header.format(term.color(color_idx)('')))
    text = make_header()

    with term.cbreak(), term.fullscreen(), term.mouse_enabled(report_motion=True):
        while True:
            print(text, end='', flush=True)
            inp = term.inkey()

            if inp.name and inp.name.startswith('MOUSE_'):
                # process scroll wheel changes color
                _offset = (1 if inp.name == 'MOUSE_SCROLL_UP' else
                           -1 if inp.name == 'MOUSE_SCROLL_DOWN' else 0)
                color_idx = (color_idx + _offset) % num_colors

                # and left mouse paints, right erases
                char = (term.color(color_idx)('')
                        if inp.name.startswith('MOUSE_LEFT')
                        else ' ' if inp.name.startswith('MOUSE_RIGHT')
```

(continues on next page)

(continued from previous page)

```
                        else '')

                # update draw text using mouse_yx
                text = make_header() + term.move_yx(*inp.mouse_yx) + char
```

Painting is done while the left mouse button is held down, tracking both `MOUSE_LEFT` and `MOUSE_LEFT_MOTION`, erased with `MOUSE_RIGHT`, and color selection changed by `MOUSE_SCROLL_UP` and `MOUSE_SCROLL_DOWN`.

> **Note**
>
> When using `report_motion=True`, process events quickly! Mouse movement generates many events that can fill the input buffer if not consumed promptly.

7.5.3 report_pixels

By default, mouse positions are reported in character cell coordinates - each position corresponds to a single character in the terminal grid.

For higher precision, use `report_pixels=True` to get pixel coordinates instead. This is especially useful when combined with graphics protocols like Sixel:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

if not term.does_mouse(report_pixels=True):
    print("This terminal does not support pixel coordinate mouse tracking!")
else:
    print("Click to display Pixel coordinates, ^C to quit:")
    with term.cbreak(), term.mouse_enabled(report_pixels=True):
        while True:
            event = term.inkey()

            if event.name and event.name.startswith('MOUSE_'):
                print(f"Pixel position: (y={event.y}, x={event.x})")
```

When using pixel mode, mouse events still use the same *name* pattern (e.g., `'MOUSE_LEFT'`) and magic method predicates (e.g., `inp.is_mouse_left()`). The `x` and `y` properties represent pixels instead of character cells.

7.6 Legacy Mouse Protocols

Blessed always prefers the SGR protocol (*DEC Private Modes* 1006 and 1016), and *mouse_enabled()* negotiates it for you. However, a legacy terminal supporting the X10-era protocol of modes 1000, 1002, and 1003 is decoded when it arrives, but is reached only by historical terminals that dot support SGR.

Legacy mouse protocols have one major difference, that a button release does not report *which* button was released. Blessed names this `MOUSE_RELEASED` (or, with modifiers, `MOUSE_CTRL_RELEASED`).

Legacy mouse protocols also have coordinate limitations, values above 223 cannot be encoded, and, some terminals like `conhost.exe` are even further limited to 95.

CHAPTER

EIGHT

CURSOR SHAPES

DECSCUSR (DEC Set Cursor Style) escape sequences allow changing the cursor shape between block, underline, and bar styles, with optional blinking.

The blessed library provides:

- A *CursorShape* class with constants for each style
- A `cursor_shape()` context manager for temporary changes

8.1 Available Shapes

Constant	Value	Description
`DEFAULT`	0	Reset to terminal default
`BLINKING_BLOCK`	1	Blinking block cursor
`STEADY_BLOCK`	2	Steady (non-blinking) block
`BLINKING_UNDERLINE`	3	Blinking underline cursor
`STEADY_UNDERLINE`	4	Steady underline cursor
`BLINKING_BAR`	5	Blinking bar (line) cursor
`STEADY_BAR`	6	Steady bar (line) cursor

8.2 Usage

Use the `cursor_shape()` context manager to temporarily change the cursor shape. The cursor is automatically restored to the terminal default on exit:

```
from blessed import Terminal

term = Terminal()

with term.cursor_shape(term.CursorShape.BLINKING_BAR):
    # cursor is a blinking bar
    val = term.inkey()
```

String names are also accepted:

```
with term.cursor_shape('steady_underline'):
    val = term.inkey()
```

8.3 Direct Sequences

You can also use `sequence()` to get the raw escape string for manual use:

```
import sys
from blessed.cursor_shape import CursorShape

sys.stdout.write(CursorShape.sequence(CursorShape.STEADY_BAR))
sys.stdout.flush()

# ... do work ...

sys.stdout.write(CursorShape.sequence(CursorShape.DEFAULT))
sys.stdout.flush()
```

8.4 Example

```
#!/usr/bin/env python3
"""Demonstrate DECSCUSR cursor shape changes with line editor input."""
from blessed import Terminal
from blessed.line_editor import LineEditor

term = Terminal()


def readline(col, width=40):
    """Read a line of input at current position."""
    ed = LineEditor(max_width=width, limit=200)
    while True:
        result = ed.feed_key(term.inkey())
        if result.line is not None or result.eof or result.interrupt:
            return result.line or ''
        if result.changed:
            ds = ed.display
            print(term.move_x(col) + term.clear_eol + ds.text
                  + term.move_x(col + ds.cursor), end='', flush=True)


print("Cursor shape demo -- type with each shape, press Enter to continue.\n")

with term.cbreak():
    for name, value in sorted(term.CursorShape.STYLES.items()):
        if name == 'default':
            continue
        label = f"  {name}: "
        print(label, end='', flush=True)
        with term.cursor_shape(value):
            readline(col=len(label))
        print()

print("\nCursor restored to terminal default.")
```

CHAPTER

NINE

DEC PRIVATE MODES

DEC Private Modes are terminal control sequences that enable or disable specific terminal behaviors like cursor visibility, mouse tracking, alternate screen buffers, and modern features like synchronized output.

The blessed library provides a clean API for working with these modes:

- Query mode support with `get_dec_mode()`
- Enable modes with `dec_modes_enabled()` context manager
- Disable modes with `dec_modes_disabled()` context manager

Each mode is identified by a constant from `Terminal.DecPrivateMode`. Our mode catalog is derived from https://wiki.tau.garden/dec-modes/

9.1 Overview

DEC Private Modes control a wide variety of terminal features. Some common examples include:

- **DECTCEM** (25) - Cursor visibility
- **MOUSE_REPORT_CLICK** (1000) - Basic mouse click reporting
- **MOUSE_EXTENDED_SGR** (1006) - Extended mouse reporting with pixel coordinates
- **BRACKETED_PASTE** (2004) - Receive clipboard paste as a single event
- **FOCUS_IN_OUT_EVENTS** (1004) - Detect when terminal window gains/loses focus
- **SYNCHRONIZED_OUTPUT** (2026) - Eliminate screen flicker during redraws

The context managers gracefully handle unsupported modes - your code works normally even on terminals that don't support specific features.

9.2 Timeout Behavior

All DEC mode context managers use a default `timeout` of 1 second when querying mode support. This timeout is designed to handle "dumb" terminals that don't support DEC mode queries and won't respond.

The timeout delay only occurs on the **first** call to any mode query method, as subsequent calls use cached results. Unsupported modes are gracefully ignored - your code continues to work even when a terminal doesn't support a specific feature.

You can verify mode support quickly using `get_dec_mode()` (see *Querying Mode Support* below), which also benefits from caching after the first query.

Terminals with leaky output, such as Terminal.app, display a DECRQM query as a stray `p` instead of parsing it. They are identified by an XTVERSION query made by the first mode query, and are never sent a DECRQM query, reporting every mode as *NOT_QUERIED*.

9.3 Getting Started

Here's a simple example that temporarily hides the cursor:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

print("Watch the cursor disappear, ")
with term.dec_modes_disabled(term.DecPrivateMode.DECTCEM):
    print("Cursor is hidden - working...")
    term.inkey(2)

print()
print("Cursor is back!")
```

The cursor automatically reappears when the context exits, even if an exception occurs.

This usually emits the same sequences recorded in the terminfo database of modern terminals for the `term.hide_cursor` and `term.normal_cursor` attributes and offered by our context manager method, `hidden_cursor()`.

The difference is that we can also make inquiries into the whether the mode is supported at all.

9.4 Querying Mode Support

You can check if a terminal supports a specific mode using `get_dec_mode()`. This is useful for adapting your application to different terminal capabilities:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

# Query mouse support
mode = term.DecPrivateMode(term.DecPrivateMode.MOUSE_REPORT_CLICK)
response = term.get_dec_mode(mode)

print(f"Checking {mode.name} (mode {mode.value}) {mode.long_description}: ", end="")

if response.supported:
    status = "enabled" if response.enabled else "disabled"
    state = "permanently" if response.permanent else "temporarily"
    print(f"Supported and {status} {state}")
elif response.failed:
    print("Terminal does not support DEC mode queries")
else:
    print("Mode not supported by this terminal")
```

The `timeout` value is unspecified and defaults to 1 second. If a Terminal fails to respond in this amount of time, the special property `failed` becomes `True`.

The *DecModeResponse* object provides helper properties:

- *supported* - Mode is recognized
- *enabled* - Mode is currently active
- *disabled* - Mode is currently inactive
- *permanent* - Mode setting cannot be changed
- *failed* - Query failed or timed out

Query results are automatically cached. Use `force=True` to bypass the cache:

Try the *display-modes.py* example program to detect and report all supported sequences for a given terminal.

9.5 Quick Boolean Detection

For the most commonly used modes, convenience methods return a simple boolean without needing to inspect the full *DecModeResponse*:

- `does_bracketed_paste()` – Bracketed Paste (mode 2004)
- `does_synchronized_output()` – Synchronized Output (mode 2026)
- `does_grapheme_clustering()` – Grapheme Clustering (mode 2027)
- `does_focus_events()` – Focus In/Out Events (mode 1004)

```
term = Terminal()

if term.does_synchronized_output():
    print("Synchronized output supported")

if term.does_bracketed_paste():
    print("Bracketed paste supported")
```

These methods accept the same `timeout` parameter as `get_dec_mode()` and benefit from the same caching.

9.6 Context Managers

The recommended way to work with modes is through context managers:

- `dec_modes_enabled()` - Temporarily enable one or more modes
- `dec_modes_disabled()` - Temporarily disable one or more modes

These context managers:

1. Query the Terminal's support for a mode within given timeout
2. Change the mode if allowed by negotiation
3. Restore the original state on exit if changed
4. Handle unsupported modes gracefully

You can pass multiple modes to enable them simultaneously:

```
with term.dec_modes_enabled(
    term.DecPrivateMode.DECTCEM,
    term.DecPrivateMode.MOUSE_REPORT_CLICK,
):
    # Both modes enabled here
    pass
```

For commonly-used DEC modes, blessed provides convenient context managers on the *Terminal* object that make accessing them easier.

These convenience wrappers all contain a default `timeout` argument of 1. Given this, for terminals that do not support DEC Mode negotiation, the first call will cause up to a 1 second delay while awaiting a possible terminal response to confirm support.

All subsequent calls use a cache of the failed query result and incur no further delays.

9.7 Synchronized Output

Synchronized Output (mode 2026) eliminates screen flicker by buffering all output until the mode is exited. This is perfect for animations and full-screen redraws.

Without synchronized output, rapidly clearing and redrawing the screen creates a visible blink effect. With it, updates appear instantly:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

fill = "" * term.height * term.width
empty = " " * term.height * term.width

print(term.bold_red("Warning! Screen may blink rapidly!"))
print()
print("Press return to continue, 'q' to stop test")
term.inkey()

with term.fullscreen():
    for step in range(300):
        with term.synchronized_output():
            print(term.home + empty, flush=True)
            print(term.home + fill, flush=True)
            print(term.home + f'step={step}')
        if term.inkey(0.01) == 'q':
            break

print(term.clear + "Test complete!")
```

On terminals that support this mode, you'll just see screen of `fill` characters with a counter in the top-left corner. On terminals that do not support it, partial draws of `empty` spaces will cause the screen to occasionally blink or flash.

9.8 Bracketed Paste

Bracketed Paste (mode 2004) allows your application to receive clipboard paste operations as a single event rather than a stream of individual characters. This makes it easy to distinguish between typed and pasted text:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

print("Paste some text (press 'q' to quit)...")

with term.bracketed_paste():
    with term.cbreak():
        while True:
            ks = term.inkey()

            if ks.name == 'BRACKETED_PASTE':
                print(f"Pasted: {term.reverse(repr(ks.text))}")
            elif ks == 'q':
                print("Goodbye!")
                break
            elif ks:
                print(f"Regular key: {ks!r}")
```

When `Keystroke.mode` equals `BRACKETED_PASTE`, the `mode_values()` method returns a *BracketedPasteEvent* with a `text` attribute containing the pasted content.

9.9 Focus Events

Focus tracking (mode 1004) reports when the terminal window gains or loses focus. This is useful for pausing animations or updating status indicators:

```
#!/usr/bin/env python3
from blessed import Terminal

term = Terminal()

print("Switch focus to/from this terminal window, 'q' to stop.")

with term.focus_events():
    with term.cbreak():
        while True:
            inp = term.inkey()
            if inp.name == 'FOCUS_IN':
                print("Focus gained")
            elif inp.name == 'FOCUS_OUT':
                print("Focus lost")
            elif inp == 'q':
                break
```

When `Keystroke.mode` equals `FOCUS_IN_OUT_EVENTS`, the `mode_values()` method returns a *FocusEvent* with a `gained` attribute indicating whether focus was gained (`True`) or lost (`False`).

CHAPTER

TEN

LOCATION

If you just want to move the location of the cursor before writing text, and aren't worried about returning, do something like this:

```
>>> print(term.home + term.clear, end='')
>>> print(term.move_down(2) + term.move_right(20) + term.bright_red('fire!'), end='')
>>> print(term.move_xy(20, 7) + term.bold('Direct hit!'), end='')
>>> print(term.move_y(term.height - 3), end='')
```

Note our use of `end=''` to the built-in `print()` function, this is because the default `end='\n'` value causes the cursor to move to the first column of the next row.

There are four direct movement capabilities:

`move_xy(x, y)`
: Position cursor at given **x**, **y**.

`move_x(x)`
: Position cursor at column **x**.

`move_y(y)`
: Position cursor at row **y**.

`home`
: Position cursor at (0, 0).

And four relative capabilities:

`move_up` or `move_up(y)`
: Position cursor 1 or **y** row cells above the current position.

`move_down(y)`
: Position cursor 1 or **y** row cells below the current position.

> **Note**
>
> `move_down()` (no arguments) is often valued as `\\n`, which additionally returns the carriage to column 0 in the default terminal mode, (when not in `Terminal.raw` or `Terminal.cbreak`), and, depending on your terminal emulator, may also destroy remaining characters at current location to the end of the line.
>
> `move_down(1)` is *always* a safe, non-destructive, one-notch movement in the downward direction and should be preferred.

`move_left` or `move_left(x)`
: Position cursor 1 or **x** column cells left of the current position.

move_right or move_right(x)
Position cursor 1 or **x** column cells right of the current position.

10.1 Example

The source code of *bounce.py* combines a small bit of *Keyboard Input* input with many of the Terminal location capabilities, `home`, `width`, `height`, and `move_xy` are used to create a classic game of tennis:

```
# std imports
from math import floor

# local
from blessed import Terminal


def roundxy(x, y):
    return int(floor(x)), int(floor(y))


term = Terminal()

x, y, xs, ys = 2, 2, 0.4, 0.3
with term.cbreak(), term.hidden_cursor():
    # clear the screen
    print(term.home + term.black_on_olivedrab4 + term.clear)

    # loop every 20ms
    while term.inkey(timeout=0.02) != 'q':
        # erase,
        txt_erase = term.move_xy(*roundxy(x, y)) + ' '

        # bounce,
        if x >= (term.width - 1) or x <= 0:
            xs *= -1
        if y >= term.height or y <= 0:
            ys *= -1

        # move,
        x, y = x + xs, y + ys

        # draw !
        txt_ball = term.move_xy(*roundxy(x, y)) + ''
        print(txt_erase + txt_ball, end='', flush=True)
```

10.2 Context Manager

A `contextlib.contextmanager()`, *`location()`* is provided to move the cursor to an *(x, y)* screen position and *restore the previous position* on exit:

```
with term.location(0, term.height - 1):
    print('Here is the bottom.')

print('This is back where I came from.')
```

All parameters to *location()* are **optional**, we can use it without any arguments at all to restore the cursor location:

```
with term.location():
    print(term.move_xy(1, 1) + 'Hi Mom!' + term.clear_eol)
```

Note

calls to *location()* may not be nested.

10.3 Finding The Cursor

We can determine the cursor's current position at anytime using *get_location()*. The default `timeout` value of 1 second will return coordinates `(-1, -1)` if no response was received in time.

This uses a kind of "answer back" sequence that *all* terminal emulators automatically respond to.

```
>>> term.get_location()
(32, 0)
```

The return value of *get_location()* mirrors the arguments of `location()`:

```
with term.location(12, 12):
     val = term.get_location()
print(val)
```

Produces output, `(12, 12)`

Although this wouldn't be suggested in most applications because of its latency, it certainly simplifies many applications, and, can also be timed, to make a determination of the round-trip time, perhaps even the bandwidth constraints, of a remote terminal!

10.4 CPR_RESPONSE

The *get_location()* method is blocking, sending a cursor position report sequence and awaiting the response. It may sometimes be necessary send and capture Cursor Position Report responses asynchronously as keyboard input events.

- Print Cursor Position Report sequence, `term.u7 or '\x1b[6n'`
- Call `inkey()` with `capture_cpr=True`
- When return `name` is `CPR_RESPONSE`, use the `cpr_yx()` or `cpr_xy()` methods to receive the coordinates.

Although `inkey()` is capable of receiving *most* Cursor Position Report (CPR) sequences as Keystroke of name `CPR_RESPONSE`, Some coordinate responses are in conflict with the "F3" key of the DEC vt220, but few very terminals transmit them in this conflicting form. If legacy vt220 key compatibility is not required, but asynchronous receipt of CPR Responses with correct coordinates are, use `inkey()` argument `capture_cpr=True`.

In the following example, keyboard input is rapidly polled for and displayed, CPR Sequences are sent at a fixed interval, and the response or round-trip time is displayed.

```
"""Example of Terminal.inkey(capture_cpr=True)."""
import blessed
import time

term = blessed.Terminal()


def msg(term, txt):
    print(term.move_yx(term.height // 2, 0) + term.center(txt))


with term.cbreak(), term.fullscreen():
    print(term.home + term.clear)
    interval = 0.5
    cpr_last_sent = time.time() - interval
    while True:
        stime = time.time()
        if stime - cpr_last_sent > interval:
            print(term.u7 or '\x1b[6n', flush=True, end='')
            cpr_last_sent = time.time()
        inp = term.inkey(timeout=0.1, capture_cpr=True)
        print(inp)
        if inp.name == 'CPR_RESPONSE':
            elapsed_ms = (time.time() - cpr_last_sent) * 1000
            msg(term, f'CPR {elapsed_ms:3.2f}ms rtt; yx={inp.cpr_yx}')
        elif inp:
            msg(term, f"Keystroke: {inp!r} !")
```

10.5 Scroll Region

A **scroll region** restricts scrolling to a portion of the screen. When text scrolls within the region, content outside the region remains fixed. This is the mechanism behind interfaces like less(1), or irc chat clients, where new messages scroll while an input area stays in place.

A `contextlib.contextmanager()`, *`scroll_region()`* is provided to set a scrolling region on entry and restore to full screen region on exit:

```
"""Example of Terminal.scroll_region()."""
from blessed import Terminal

term = Terminal()

with term.fullscreen(), term.cbreak():
    # Draw header
    print(term.home + term.clear, end='')
    print(term.reverse(term.center("Scroll region demo")), end='')

    # Draw footer
    print(term.move_yx(term.height - 1, 0), end='')
    print(term.reverse(term.center("Press any key to stop")), end='')

    # create scrolling region
```

(continues on next page)

(continued from previous page)

```
    with term.scroll_region(top=1, height=term.height - 2):
        # Print text within scrolling region
        print(term.move_yx(1, 0), end='')
        for line_num in range(1000):
            print(f'\n{line_num:4d}: Lorem ipsum dolor sit amet.', end='')
            if term.inkey(0.01):
                break
```

CHAPTER

ELEVEN

SIZING & ALIGNMENT

The *height* and *width* properties always provide a current readout of the size of the window in character cells:

```
from blessed import Terminal
term = Terminal()

>>> term.height, term.width
(34, 102)
```

The *pixel_height* and *pixel_width* properties provide the window size in pixels when available:

```
from blessed import Terminal
term = Terminal()

>>> term.pixel_height, term.pixel_width
(1080, 1920)
```

Note

For sixel graphics, use *get_sixel_height_and_width()* instead, which returns the actual drawable area by accounting for margins. See *Sixel Graphics* for details.

11.1 Length

Any string **containing sequences** can be measured using *length()*.

This means that blessed can measure, right-align, center, truncate, or word-wrap its own output! Text containing colors and styling, hyperlinks, or text sizing protocol are measured for their visual width:

```
from blessed import Terminal
term = Terminal()

>>> print(term.length(term.bold_blue('123')))
3
>>> print(term.length('abc\b\b\bxyz'))
3
>>> print(term.length(term.text_sized('BIG', scale=3)))
9
>>> print(term.length(term.link('http://example.com', 'link')))
4
```

(continues on next page)

(continued from previous page)

```
>> phrase = (
    "\u26F9"        # PERSON WITH BALL
    "\U0001F3FB"    # EMOJI MODIFIER FITZPATRICK TYPE-1-2
    "\u200D"        # ZERO WIDTH JOINER
    "\u2640"        # FEMALE SIGN
    "\uFE0F")       # VARIATION SELECTOR-16
>> print(term.length(phrase))
2
```

The length measurements of blessed very accurately matches most popular terminals, but not all. The python wcwidth library provides support of complex emojis, CJK, flags, and complex languages, and support for unicode may vary by terminal, as discovered by the results published by ucs-detect.

length() makes a best effort without returning error, even when the cursor position is ambiguous, such as text containing `term.move_x(0)`, or containing vertical movement like `'\n'` or `term.move_up(99)`, the text is assumed for display on the first column and vertical sequences are ignored. Carriage return, `'\r'` is treated like `term.move_x(0)`:

```
from blessed import Terminal
term = Terminal()

>>> print(term.length('abc' + term.move_x(0) + 'xyz'))
3
>>> print(term.length('abc' + term.move_up(99) + 'xyz'))
6
>>> print(term.length('abc' + '\r' + 'xyz'))
3
>>> print(term.length('abc' + '\n' + 'xyz'))
6
```

11.2 Alignment

By combining the measure of the printable width of strings containing sequences with the terminal width, the *center()*, *ljust()*, *rjust()*, `truncate()`, `wrap()`, and `clip()` methods "just work" for strings that contain sequences.

Where `truncate()` removes everything right of a single column, `clip()` extracts a window of display columns, from `start` to `end`:

```
from blessed import Terminal
term = Terminal()

print(term.clip('the quick brown fox', 4, 9))  # 'quick'
```

```
from blessed import Terminal
term = Terminal()

with term.location(y=term.height // 2):
    print(term.center(term.bold('press return to begin!')))
    term.inkey()
```

In the following example, `wrap()` word-wraps a short poem containing sequences:

```
from blessed import Terminal
term = Terminal()

poem = (term.bold_cyan('Plan difficult tasks'),
        term.cyan('through the simplest tasks'),
        term.bold_cyan('Achieve large tasks'),
        term.cyan('through the smallest tasks'))

for line in poem:
    print('\n'.join(term.wrap(line, width=25, subsequent_indent=' ' * 4)))
```

11.3 Terminal Corrections

The rendered width of some characters depends on the terminal software, as discovered by the ucs-detect project. At class-initialization, blessed identifies the terminal software by the `TERM_PROGRAM` environment variable, set by most modern terminal emulators, by a `TERM` value recognized by wcwidth, and otherwise by the XTVERSION query (`CSI > q`), then applies the matching wcwidth correction tables to the methods above.

Whether East Asian ambiguous characters are rendered wide or narrow is measured on demand, by *detect_ambiguous_width()*, which draws and erases a test character and assigns the result to *ambiguous_width*. Until it is called, the value is 1, or `AMBIGUOUS_WIDE` when that environment variable is `1` or `2`, so call it after constructing the terminal and before any measurement:

```
from blessed import Terminal
term = Terminal()
term.detect_ambiguous_width()
term.detect_term_program()

print(term.term_program, term.ambiguous_width)
```

The call measures even when `AMBIGUOUS_WIDE` is set, which only supplies the value until then, such as for automated tests. When no environment value identifies the terminal and no terminal answers, such as when output is redirected, correction tables do not apply and ambiguous width remains narrow.

detect_term_program() applies the same detection and assigns *term_program*, which could be changed after construction when integrating blessed with remote streams like telnet or ssh, or any other circumstance where correction overrides are needed.

11.4 Text Sizing

The *text_sized()* method wraps text in an `OSC 66` escape sequence for terminals that support the kitty text sizing protocol. Call *does_text_sizing()* to determine whether it is supported. This enables per-character control over cell footprint, fractional font scaling, and alignment:

```
from blessed import Terminal
term = Terminal()

# determine if text sizing protocol is supported
if not term.does_text_sizing():
    exit("No support for text sizing protocol")

# 'scale' is great for printing large headings
```

(continues on next page)

(continued from previous page)

```
_scale = 3
heading = term.text_sized('Big text', scale=_scale)
print(heading + ('\n' * (_scale)) + '=' * term.length(heading))

# fractional scaling is great for superscript and subscript
print('2' + term.text_sized('128', width=1, numerator=1, denominator=3))

# text sizing sequences can be bookended or wrapped by any other sequence,
footnote_txt = term.text_sized('[2]', width=2, numerator=2, denominator=3, vertical_
↪align='top')
footnote_hlink = term.link('https://sw.kovidgoyal.net/kitty/text-sizing-protocol/',␣
↪footnote_txt)
print('Text sizing protocol' + footnote_hlink)

# and it may also be aligned
print(term.center(term.reverse_yellow(term.text_sized('Big Yellow', scale=2, vertical_
↪align='bottom'))))
```



Note

Because the first call to *text_sized()* will cause the side effect of writing destructive spaces to the columns and row following the current cursor position as required for detection of kitty text sizing protocol, it is suggested to first call *does_text_sizing()* during program initialization.

See also

text_sizing_protocol.py and *scroller.py* demonstration programs.

11.5 Detecting Resize

The terminal can notify your application when the window size changes. Blessed provides a modern cross-platform method using in-band resize notifications, with SIGWINCH as a fallback for older terminals. On Windows, native console `WINDOW_BUFFER_SIZE_EVENT` records are automatically converted to in-band resize sequences.

11.6 Using SIGWINCH

For terminals that don't support in-band resize notifications (DEC mode 2048), you can use SIGWINCH signals on Unix systems.

```
import signal
import threading
from blessed import Terminal

term = Terminal()
_resize_pending = threading.Event()

def on_resize(*args):
    _resize_pending.set()

signal.signal(signal.SIGWINCH, on_resize)

with term.cbreak():
    while True:
        if _resize_pending.is_set():
            _resize_pending.clear()
            print(f'New size: {term.height}x{term.width}')

        inp = term.inkey(timeout=0.1)
        if inp == 'q':
            break
```

Warning

SIGWINCH has limitations:

- Not available on Windows
- Signal handlers should avoid blocking operations
- Race conditions can occur between signal delivery and terminal state
- Requires careful synchronization with application state

11.7 Using notify_on_resize

Use *does_inband_resize()* to check if the terminal supports in-band resize notifications (DEC mode 2048):

```
from blessed import Terminal

term = Terminal()

if term.does_inband_resize():
    print('In-band resize notifications are supported!')
else:
    print('Falling back to SIGWINCH or polling')
```

Note

In-band resize notification support (DEC mode 2048) is currently **very limited** among terminal emulators on Unix. Most terminals do not support this feature yet. Always check with *does_inband_resize()* and provide a fallback using SIGWINCH on Unix systems.

On Windows, *does_inband_resize()* always returns `True` because the native console API provides `WINDOW_BUFFER_SIZE_EVENT` on all versions – no DEC mode support is required.

The *notify_on_resize()* context manager enables automatic resize event reporting. When the window is resized, *inkey()* will return a keystroke with *name* equal to `'RESIZE_EVENT'`. The new dimensions are immediately available through *height*, *width*, *pixel_height*, and *pixel_width*.

This method is preferred because it:

- Works cross-platform (Linux, Mac, BSD, Windows)
- Avoids race conditions inherent in signal handlers
- Delivers resize events in-band with other input
- Automatically caches dimensions for fast access

11.8 Using both

The following example demonstrates checking for support, using in-band resize notifications when available, and **falling back to SIGWINCH on Unix systems**:

```
#!/usr/bin/env python
import threading
```

(continues on next page)

(continued from previous page)

```
from blessed import Terminal

term = Terminal()

_resize_pending = threading.Event()


def on_resize(*args):
    _resize_pending.set()


def display_size(term):
    # conditionally refresh sixel size when enabled
    sixel_height, sixel_width = 0, 0
    if term.does_sixel():
        sixel_height, sixel_width = term.get_sixel_height_and_width(force=True)
    print()
    print(f'height={term.height}, width={term.width}, ' +
          f'pixel_height={term.pixel_height}, pixel_width={term.pixel_width}, ' +
          f'sixel_height={sixel_height}, sixel_width={sixel_width}',
          end='', flush=True)


if not term.does_inband_resize():
    print('In-band Window Resize not supported on this terminal')
    import sys
    if sys.platform != 'win32':
        import signal
        signal.signal(signal.SIGWINCH, on_resize)

with term.cbreak(), term.notify_on_resize():
    print("press 'q' to quit.")
    display_size(term)

    while True:
        inp = term.inkey(timeout=0.1)

        if inp == 'q':
            break

        if inp.name == 'RESIZE_EVENT':
            _resize_pending.set()
        elif _resize_pending.is_set():
            _resize_pending.clear()
            display_size(term)
```

Make note of these designs:

1. **Signal handlers only set a flag**: The `on_resize()` signal handler only calls `_resize_pending.set()` and does nothing else. Signal handlers should never perform I/O operations, terminal queries, or other complex work, because these callbacks can occur *at any time* in your code, even part-way through drawing, for example.
2. **Main loop processes events**: The main event loop checks `_resize_pending` periodically using `inkey(timeout=0.1)`. This 100ms delay naturally debounces rapid resize events from both signals and in-

band notifications, displaying only one size update regardless of how many events arrive within that window.

Note the use of `force=True` when calling *`get_sixel_height_and_width()`*, the return value is cached, so `force=True` ensures updated dimensions are retrieved after a resize event. See *Sixel Graphics* for more on caching behavior.

CHAPTER

TWELVE

SEQUENCES

Sometimes it is necessary to make sense of terminal sequences, and to distinguish them from plain text. Blessed provides methods to split, strip, and analyze strings containing escape sequences.

12.1 Splitting Sequences

The *split_seqs()* method allows you to iterate over a terminal string by its characters or sequences:

```
>>> term.split_seqs(term.bold('bbq'))
['\x1b[1m', 'b', 'b', 'q', '\x1b(B', '\x1b[m']
```

This is useful for processing terminal output character-by-character while preserving the escape sequences that control formatting.

12.2 Stripping Sequences

The *strip_seqs()* method removes all escape sequences from a string, leaving only the printable text:

```
>>> phrase = term.sienna4('coffee')
>>> phrase
'\x1b[1m\x1b[30mcoffee\x1b(B\x1b[m'
>>> term.strip_seqs(phrase)
'coffee'
```

This is useful when you need the raw text content without any formatting codes, such as when logging to a file or comparing string content.

CHAPTER

THIRTEEN

SIXEL GRAPHICS

Blessed provides support for querying sixel graphics capabilities in modern terminal emulators. Sixel is a bitmap graphics format that allows applications to display inline images directly in the terminal.

13.1 Checking Sixel Support

Before using sixel graphics, check if your terminal supports them using `does_sixel()`:

```
term = Terminal()

if term.does_sixel():
    # Terminal supports sixel graphics
    display_sixel_image()
else:
    # Fall back to text
    display_text_fallback()
```

13.2 Window Dimensions

Get the maximum pixel dimensions available for sixel graphics using *get_sixel_height_and_width()*:

```
term = Terminal()

height, width = term.get_sixel_height_and_width()
if (height, width) != (-1, -1):
    print(f"Sixel dimensions: {width}x{height} pixels")
    # Scale your sixel graphics to fit within these dimensions
else:
    print("Could not determine sixel dimensions")
```

The returned dimensions represent the maximum height and width in pixels for sixel graphics rendering. If the query times out or fails, `(-1, -1)` is returned.

Note

The sixel area dimensions may differ from window pixel dimensions reported by *pixel_height* and *pixel_width*.

Window pixel dimensions from *pixel_height* and *pixel_width* may include margins and window decorations, reporting values too large to contain an image.

13.3 Cell Dimensions

Get the pixel dimensions of a single character cell using `get_cell_height_and_width()`:

```
term = Terminal()

cell_height, cell_width = term.get_cell_height_and_width()
if (cell_height, cell_width) != (-1, -1):
    print(f"Character cell size: {cell_width}x{cell_height} pixels")
    # Useful for pixel-perfect positioning of graphics
else:
    print("Could not determine cell dimensions")
```

Note

Although not necessarily a sixel feature, this information is crucial for precise positioning and sizing of graphics relative to text. However, many modern terminals may not report these values accurately.

13.4 Colors

Determine how many colors are available for sixel graphics using `get_sixel_colors()`:

```
term = Terminal()

colors = term.get_sixel_colors()
if colors != -1:
    print(f"Sixel color registers: {colors}")
    # Use this to optimize your sixel color palette
else:
    print("Could not determine sixel color support")
```

Returns the number of color registers available, defaults to `256` for terminals that support sixel but fail to respond to `XTSMGRAPHICS` query.

Returns `-1` when sixel is not supported.

13.5 Caching

`get_sixel_height_and_width()` queries the terminal for automatic response using XTSMGRAPHICS and XTWINOPS within combined `timeout`, and, once received, that value is cached and always returned without further query unless `force=True` is set.

When In-band resizing is used, the return values are updated automatically.

Bypassing the cache with `force=True` is recommended when window dimensions change, such as when handling resize events. This ensures that sixel graphics dimensions are re-queried to reflect the new terminal size:

```
term = Terminal()

# First call queries the terminal
height1, width1 = term.get_sixel_height_and_width()
```

(continues on next page)

(continued from previous page)

```
# Second call returns cached result (instant)
# and always returns same result
height2, width2 = term.get_sixel_height_and_width()
assert width2, height2 == width1, height2
```

Note

When handling window resize events (via *notify_on_resize()* or SIGWINCH), use `force=True` to **re-query** sixel dimensions, avoiding the last known cached value:

```
def on_resize(term):
    # Re-query sixel dimensions after resize
    height, width = term.get_sixel_height_and_width(force=True)
```

See *Sizing & Alignment* for a complete example.

13.6 Complete Workflow

Here is a complete example of checking for Sixel support, Window, and Cell dimensions:

```
#!/usr/bin/env python
from blessed import Terminal

term = Terminal()

# Check for sixel support
if not term.does_sixel():
    print("This terminal does not support sixel graphics")
else:
    print("This terminal probably supports sixel graphics")

# Get display dimensions
height, width = term.get_sixel_height_and_width()
if (height, width) == (-1, -1):
    print("Could not determine sixel dimensions")
else:
    print(f"Sixel area: {width}x{height} (px)")

# Get color support
colors = term.get_sixel_colors()
if colors == -1:
    print("Could not determine color support")
else:
    print(f"Colors available: {colors}")

# Get cell dimensions for positioning
cell_height, cell_width = term.get_cell_height_and_width()
if (cell_height, cell_width) == (-1, -1):
    print("Could not determine cell size")
else:
    print(f"Character cells: {cell_width}x{cell_height} (px)")
```

CHAPTER

FOURTEEN

EXAMPLES

A few programs are provided with blessed to help interactively test the various API features, but also serve as examples of using blessed to develop applications.

These examples are not distributed with the package – they are only available in the github repository. You can retrieve them by cloning the repository, or simply downloading the "raw" file link.

Fig. 1: *x11_colorpicker.py*, *bounce.py*, *worms.py*, and *plasma.py*.

14.1 bounce.py

https://github.com/jquast/blessed/blob/master/bin/bounce.py

This is a very brief, basic primitive non-interactive version of a "classic tennis" video game. It demonstrates basic timed refresh of a bouncing terminal cell.

14.2 cellestial.py

https://github.com/jquast/blessed/blob/master/bin/cellestial.py

This is an Elementary Cellular Automata browser, of the visualizations made popular by Stephen Wolfram. A popular "rule 30" is first displayed with random initial values and can be manipulated by hotkeys, or automatic viewing with arguments,

```
bin/cellestial.py --autoscroll --fullscreen
```

14.3 cnn.py

https://github.com/jquast/blessed/blob/master/bin/cnn.py

This program uses 3rd-party BeautifulSoup and requests library to fetch the cnn website and display news article titles using the `link()` method, so that they may be clicked.

14.4 detect-multibyte.py

https://github.com/jquast/blessed/blob/master/bin/detect-multibyte.py

This program also demonstrates how the `get_location()` method can be used to reliably test whether the terminal emulator supports utf-8 or other multibyte encoding, by rendering multibyte characters and measuring the change in cursor location.

14.5 editor.py

https://github.com/jquast/blessed/blob/master/bin/editor.py

This program demonstrates using the directional keys and noecho input mode. It acts as a (very dumb) fullscreen editor, with support for saving a file, as well as including a rudimentary line-editor.

14.6 keymatrix.py

https://github.com/jquast/blessed/blob/master/bin/keymatrix.py

This program displays a special REPL for using *inkey()* and displaying the returned *Keystroke* events, their names, events, and values.

This also provides an interface for toggling special modes, such as any of the supported *Kitty Keyboard Protocol* Keyboard Protocols.

14.7 on_resize.py

https://github.com/jquast/blessed/blob/master/bin/on_resize.py

This program uses in-band resize notifications when available, with a SIGWINCH signal handler as a fallback on Unix. On Windows, native console resize events are used automatically.

This demonstrates how a program can react to screen resize events cross-platform.

14.8 plasma.py

https://github.com/jquast/blessed/blob/master/bin/plasma.py

This demonstrates using only *on_color_rgb()* and the built-in `colorsys` module to quickly display all of the colors of a rainbow in a classic demoscene plasma effect

14.9 progress_bar.py

https://github.com/jquast/blessed/blob/master/bin/progress_bar.py

This program demonstrates a simple ascii progress bar along with the `OSC 9;4` progress bar notification sequence using *progress_bar()*.

14.10 resize.py

https://github.com/jquast/blessed/blob/master/bin/resize.py

This program demonstrates the *get_location()* method, behaving similar to resize(1) : set environment and terminal settings to current window size. The window size is determined by eliciting an answerback sequence from the connecting terminal emulator.

14.11 tprint.py

https://github.com/jquast/blessed/blob/master/bin/tprint.py

This program demonstrates how users may customize FormattingString styles. Accepting a string style, such as "bold" or "bright_red" as the first argument, all subsequent arguments are displayed by the given style. This shows how a program could provide user-customizable compound formatting names to configure a program's styling.

14.12 worms.py

https://github.com/jquast/blessed/blob/master/bin/worms.py

This program demonstrates how an interactive game could be made with blessed. It is similar to NIBBLES.BAS or "snake" of early mobile platforms.

14.13 x11_colorpicker.py

https://github.com/jquast/blessed/blob/master/bin/x11_colorpicker.py

This program shows all of the X11 colors, demonstrates a basic keyboard-interactive program and color selection, but is also a useful utility to pick colors!

14.14 display-modes.py

https://github.com/jquast/blessed/blob/master/bin/display-modes.py

Detect and report all known DEC Private Modes supported by the Terminal and display a report.

14.15 display-unicode.py

https://github.com/jquast/blessed/blob/master/bin/display-unicode.py

Basic Unicode detection support, for Wide characters, Emojis with ZWJ, Emojis with VS-16, and Ambiguous Width as Wide or Narrow by writing effects to the screen and measuring the cursor position.

14.16 strip.py

https://github.com/jquast/blessed/blob/master/bin/strip.py

This wrapper of the `strip_seqs()` method reads text from standard input, "strips" away all known output sequences and writes the result to stdout.

14.17 mouse_paint.py

https://github.com/jquast/blessed/blob/master/bin/mouse_paint.py

This is a basic "paint" program using *Mouse Input* with the *report_motion* feature.

14.18 scroll_region.py

https://github.com/jquast/blessed/blob/master/bin/scroll_region.py

This program demonstrates the *scroll_region()* context manager to create a scrollable area with a fixed header and status bar.

14.19 text_sizing_protocol.py

https://github.com/jquast/blessed/blob/master/bin/text_sizing_protocol.py

This program demonstrates the various uses of *text_sized()*, the OSC 66 kitty text sizing protocol.

14.20 detect-tofus.py

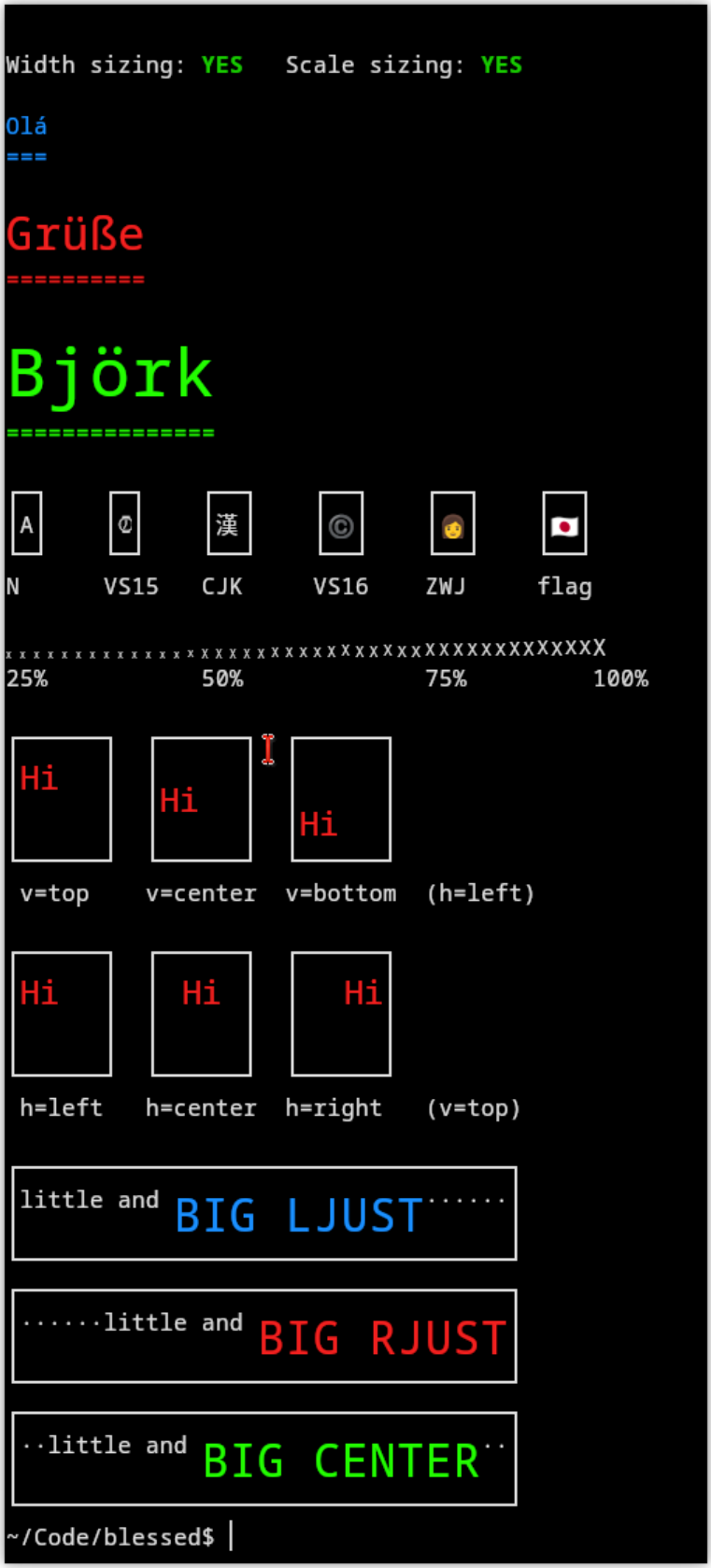
https://github.com/jquast/blessed/blob/master/bin/detect-tofus.py

This program demonstrates *get_font_coverage()*, which reports only per-codepoint coverage. Given text files as arguments, reports any "tofus" that might result when rendering them to terminal. A "tofu" is drawn for characters the terminal font cannot render. Requires a terminal implementing one of the font glyph coverage protocols.

14.21 scroller.py

https://github.com/jquast/blessed/blob/master/bin/scroller.py

A classic sine-wave "demoscene" text scroller effect that uses *text_sized()*, the OSC 66 kitty text sizing protocol.


Width sizing: YES Scale sizing: YES
Olá
===
Grüße
==========
Björk
===============
A
✆
漢
©
👩
🇯🇵
N
VS15
CJK
VS16
ZWJ
flag
25%
50%
75%
100%
Hi
Hi
Hi
v=top
v=center
v=bottom
(h=left)
Hi
Hi
Hi
h=left
h=center
h=right
(v=top)
little and BIG LJUST
little and BIG RJUST
little and BIG CENTER
~/Code/blessed$

CHAPTER

FIFTEEN

API DOCUMENTATION

15.1 color.py

Sub-module providing color functions.

References,

- https://en.wikipedia.org/wiki/Color_difference
- http://www.easyrgb.com/en/math.php
- Measuring Colour by R.W.G. Hunt and M.R. Pointer

rgb_to_xyz(*red: int*, *green: int*, *blue: int*) → Tuple[float, float, float]

Convert standard RGB color to XYZ color.

D65/2° standard illuminant.

Parameters

- **red** (*int*) – RGB value of Red.
- **green** (*int*) – RGB value of Green.
- **blue** (*int*) – RGB value of Blue.

Returns
Tuple (X, Y, Z) representing XYZ color

Return type
tuple

xyz_to_lab(*x_val: float*, *y_val: float*, *z_val: float*) → Tuple[float, float, float]

Convert XYZ color to CIE-Lab color.

Parameters

- **x_val** (*float*) – XYZ value of X.
- **y_val** (*float*) – XYZ value of Y.
- **z_val** (*float*) – XYZ value of Z.

Returns
Tuple (L, a, b) representing CIE-Lab color

Return type
tuple D65/2° standard illuminant

rgb_to_lab(*red: int*, *green: int*, *blue: int*) → Tuple[float, float, float]

Convert RGB color to CIE-Lab color.

Parameters

- **red** (*int*) – RGB value of Red.
- **green** (*int*) – RGB value of Green.
- **blue** (*int*) – RGB value of Blue.

Returns
Tuple (L, a, b) representing CIE-Lab color

Return type
tuple D65/2° standard illuminant

dist_rgb(*rgb1: Tuple[int, int, int]*, *rgb2: Tuple[int, int, int]*) → float

Determine distance between two rgb colors.

Parameters

- **rgb1** (*tuple*) – RGB color definition
- **rgb2** (*tuple*) – RGB color definition

Returns
Square of the distance between provided colors

Return type
float

This works by treating RGB colors as coordinates in three dimensional space and finding the closest point within the configured color range using the formula:

```
d^2 = (r2 - r1)^2 + (g2 - g1)^2 + (b2 - b1)^2
```

For efficiency, the square of the distance is returned which is sufficient for comparisons

dist_rgb_weighted(*rgb1: Tuple[int, int, int]*, *rgb2: Tuple[int, int, int]*) → float

Determine the weighted distance between two rgb colors.

Parameters

- **rgb1** (*tuple*) – RGB color definition
- **rgb2** (*tuple*) – RGB color definition

Returns
Square of the distance between provided colors

Return type
float Similar to a standard distance formula, the values are weighted to approximate human perception of color differences For efficiency, the square of the distance is returned which is sufficient for comparisons

dist_cie76(*rgb1: Tuple[int, int, int]*, *rgb2: Tuple[int, int, int]*) → float

Determine distance between two rgb colors using the CIE76 algorithm.

Parameters

- **rgb1** (*tuple*) – RGB color definition
- **rgb2** (*tuple*) – RGB color definition

Returns
Square of the distance between provided colors

Return type
float For efficiency, the square of the distance is returned which is sufficient for comparisons

dist_cie94(*rgb1: Tuple[int, int, int]*, *rgb2: Tuple[int, int, int]*) → float

Determine distance between two rgb colors using the CIE94 algorithm.

Parameters

- **rgb1** (*tuple*) – RGB color definition
- **rgb2** (*tuple*) – RGB color definition

Returns
Square of the distance between provided colors

Return type
float For efficiency, the square of the distance is returned which is sufficient for comparisons

dist_cie2000(*rgb1: Tuple[int, int, int]*, *rgb2: Tuple[int, int, int]*) → float

Determine distance between two rgb colors using the CIE2000 algorithm.

Parameters

- **rgb1** (*tuple*) – RGB color definition
- **rgb2** (*tuple*) – RGB color definition

Returns
Square of the distance between provided colors

Return type
float For efficiency, the square of the distance is returned which is sufficient for comparisons

xterm256color_from_rgb(*red: int*, *green: int*, *blue: int*) → Tuple[int, Tuple[int, int, int]]

Convert RGB values to xterm 256-color cube index and RGB approximation.

Uses the 6x6x6 color cube (indices 16-231) with levels [0,95,135,175,215,255].

Parameters

- **red** (*int*) – RGB value of Red (0-255).
- **green** (*int*) – RGB value of Green (0-255).
- **blue** (*int*) – RGB value of Blue (0-255).

Returns
Tuple (cube_index, (r, g, b)) representing the xterm cube index and RGB approximation

Return type
tuple

xterm256gray_from_rgb(*red: int*, *green: int*, *blue: int*) → Tuple[int, Tuple[int, int, int]]

Convert RGB values to xterm 256-color grayscale index and RGB approximation.

Uses the 24 grayscale entries (indices 232-255) with values 8+10*i.

Parameters

- **red** (*int*) – RGB value of Red (0-255).
- **green** (*int*) – RGB value of Green (0-255).

- **blue** (*int*) – RGB value of Blue (0-255).

Returns
Tuple (gray_index, (r, g, b)) representing the xterm gray index and RGB approximation

Return type
tuple

15.2 colorspace.py

Color reference data.

References,

- https://github.com/freedesktop/xorg-rgb/blob/master/rgb.txt
- https://github.com/ThomasDickey/xterm-snapshots/blob/master/256colres.h
- https://github.com/ThomasDickey/xterm-snapshots/blob/master/XTerm-col.ad
- https://en.wikipedia.org/wiki/ANSI_escape_code#Colors
- https://gist.github.com/XVilka/8346728
- https://devblogs.microsoft.com/commandline/24-bit-color-in-the-windows-console/
- http://jdebp.uk/Softwares/nosh/guide/TerminalCapabilities.html

class RGBColor(*red*, *green*, *blue*)

Named tuple for an RGB color definition.

Create new instance of RGBColor(red, green, blue)

```
RGB_256TABLE: Tuple[RGBColor, ...] = ((0, 0, 0), (205, 0, 0), (0, 205, 0), (205, 205, 0),
(0, 0, 238), (205, 0, 205), (0, 205, 205), (229, 229, 229), (127, 127, 127), (255, 0, 0),
(0, 255, 0), (255, 255, 0), (92, 92, 255), (255, 0, 255), (0, 255, 255), (255, 255, 255),
(0, 0, 0), (0, 0, 95), (0, 0, 135), (0, 0, 175), (0, 0, 215), (0, 0, 255), (0, 95, 0),
(0, 95, 95), (0, 95, 135), (0, 95, 175), (0, 95, 215), (0, 95, 255), (0, 135, 0), (0,
135, 95), (0, 135, 135), (0, 135, 175), (0, 135, 215), (0, 135, 255), (0, 175, 0), (0,
175, 95), (0, 175, 135), (0, 175, 175), (0, 175, 215), (0, 175, 255), (0, 215, 0), (0,
215, 95), (0, 215, 135), (0, 215, 175), (0, 215, 215), (0, 215, 255), (0, 255, 0), (0,
255, 95), (0, 255, 135), (0, 255, 175), (0, 255, 215), (0, 255, 255), (95, 0, 0), (95, 0,
95), (95, 0, 135), (95, 0, 175), (95, 0, 215), (95, 0, 255), (95, 95, 0), (95, 95, 95),
(95, 95, 135), (95, 95, 175), (95, 95, 215), (95, 95, 255), (95, 135, 0), (95, 135, 95),
(95, 135, 135), (95, 135, 175), (95, 135, 215), (95, 135, 255), (95, 175, 0), (95, 175,
95), (95, 175, 135), (95, 175, 175), (95, 175, 215), (95, 175, 255), (95, 215, 0), (95,
215, 95), (95, 215, 135), (95, 215, 175), (95, 215, 215), (95, 215, 255), (95, 255, 0),
(95, 255, 95), (95, 255, 135), (95, 255, 175), (95, 255, 215), (95, 255, 255), (135, 0,
0), (135, 0, 95), (135, 0, 135), (135, 0, 175), (135, 0, 215), (135, 0, 255), (135, 95,
0), (135, 95, 95), (135, 95, 135), (135, 95, 175), (135, 95, 215), (135, 95, 255), (135,
135, 0), (135, 135, 95), (135, 135, 135), (135, 135, 175), (135, 135, 215), (135, 135,
255), (135, 175, 0), (135, 175, 95), (135, 175, 135), (135, 175, 175), (135, 175, 215),
(135, 175, 255), (135, 215, 0), (135, 215, 95), (135, 215, 135), (135, 215, 175), (135,
215, 215), (135, 215, 255), (135, 255, 0), (135, 255, 95), (135, 255, 135), (135, 255,
175), (135, 255, 215), (135, 255, 255), (175, 0, 0), (175, 0, 95), (175, 0, 135), (175,
0, 175), (175, 0, 215), (175, 0, 255), (175, 95, 0), (175, 95, 95), (175, 95, 135), (175,
95, 175), (175, 95, 215), (175, 95, 255), (175, 135, 0), (175, 135, 95), (175, 135, 135),
(175, 135, 175), (175, 135, 215), (175, 135, 255), (175, 175, 0), (175, 175, 95), (175,
175, 135), (175, 175, 175), (175, 175, 215), (175, 175, 255), (175, 215, 0), (175, 215,
95), (175, 215, 135), (175, 215, 175), (175, 215, 215), (175, 215, 255), (175, 255, 0),
(175, 255, 95), (175, 255, 135), (175, 255, 175), (175, 255, 215), (175, 255, 255), (215,
0, 0), (215, 0, 95), (215, 0, 135), (215, 0, 175), (215, 0, 215), (215, 0, 255), (215,
95, 0), (215, 95, 95), (215, 95, 135), (215, 95, 175), (215, 95, 215), (215, 95, 255),
(215, 135, 0), (215, 135, 95), (215, 135, 135), (215, 135, 175), (215, 135, 215), (215,
135, 255), (215, 175, 0), (215, 175, 95), (215, 175, 135), (215, 175, 175), (215, 175,
215), (215, 175, 255), (215, 215, 0), (215, 215, 95), (215, 215, 135), (215, 215, 175),
(215, 215, 215), (215, 215, 255), (215, 255, 0), (215, 255, 95), (215, 255, 135), (215,
255, 175), (215, 255, 215), (215, 255, 255), (255, 0, 0), (255, 0, 135), (255, 0, 95),
(255, 0, 175), (255, 0, 215), (255, 0, 255), (255, 95, 0), (255, 95, 95), (255, 95, 135),
(255, 95, 175), (255, 95, 215), (255, 95, 255), (255, 135, 0), (255, 135, 95), (255, 135,
135), (255, 135, 175), (255, 135, 215), (255, 135, 255), (255, 175, 0), (255, 175, 95),
(255, 175, 135), (255, 175, 175), (255, 175, 215), (255, 175, 255), (255, 215, 0), (255,
215, 95), (255, 215, 135), (255, 215, 175), (255, 215, 215), (255, 215, 255), (255, 255,
0), (255, 255, 95), (255, 255, 135), (255, 255, 175), (255, 255, 215), (255, 255, 255),
(8, 8, 8), (18, 18, 18), (28, 28, 28), (38, 38, 38), (48, 48, 48), (58, 58, 58), (68, 68,
68), (78, 78, 78), (88, 88, 88), (98, 98, 98), (108, 108, 108), (118, 118, 118), (128,
128, 128), (138, 138, 138), (148, 148, 148), (158, 158, 158), (168, 168, 168), (178, 178,
178), (188, 188, 188), (198, 198, 198), (208, 208, 208), (218, 218, 218), (228, 228,
228), (238, 238, 238))
```

Curses color indices of 8, 16, and 256-color terminals

```
X11_COLORNAMES_TO_RGB: Dict[str, RGBColor] = {'aliceblue': (240, 248, 255),
'antiquewhite': (250, 235, 215), 'antiquewhite1': (255, 239, 219), 'antiquewhite2':
(238, 223, 204), 'antiquewhite3': (205, 192, 176), 'antiquewhite4': (139, 131, 120),
'aqua': (0, 255, 255), 'aquamarine': (127, 255, 212), 'aquamarine1': (127, 255, 212),
'aquamarine2': (118, 238, 198), 'aquamarine3': (102, 205, 170), 'aquamarine4': (69,
139, 116), 'azure': (240, 255, 255), 'azure1': (240, 255, 255), 'azure2': (224, 238,
238), 'azure3': (193, 205, 205), 'azure4': (131, 139, 139), 'beige': (245, 245, 220),
'bisque': (255, 228, 196), 'bisque1': (255, 228, 196), 'bisque2': (238, 213, 183),
'bisque3': (205, 183, 158), 'bisque4': (139, 125, 107), 'black': (0, 0, 0),
'blanchedalmond': (255, 235, 205), 'blue': (0, 0, 255), 'blue1': (0, 0, 255), 'blue2':
(0, 0, 238), 'blue3': (0, 0, 205), 'blue4': (0, 0, 139), 'blueviolet': (138, 43, 226),
'brown': (165, 42, 42), 'brown1': (255, 64, 64), 'brown2': (238, 59, 59), 'brown3':
(205, 51, 51), 'brown4': (139, 35, 35), 'burlywood': (222, 184, 135), 'burlywood1':
(255, 211, 155), 'burlywood2': (238, 197, 145), 'burlywood3': (205, 170, 125),
'burlywood4': (139, 115, 85), 'cadetblue': (95, 158, 160), 'cadetblue1': (152, 245,
255), 'cadetblue2': (142, 229, 238), 'cadetblue3': (122, 197, 205), 'cadetblue4': (83,
134, 139), 'chartreuse': (127, 255, 0), 'chartreuse1': (127, 255, 0), 'chartreuse2':
(118, 238, 0), 'chartreuse3': (102, 205, 0), 'chartreuse4': (69, 139, 0), 'chocolate':
(210, 105, 30), 'chocolate1': (255, 127, 36), 'chocolate2': (238, 118, 33),
'chocolate3': (205, 102, 29), 'chocolate4': (139, 69, 19), 'coral': (255, 127, 80),
'coral1': (255, 114, 86), 'coral2': (238, 106, 80), 'coral3': (205, 91, 69), 'coral4':
(139, 62, 47), 'cornflowerblue': (100, 149, 237), 'cornsilk': (255, 248, 220),
'cornsilk1': (255, 248, 220), 'cornsilk2': (238, 232, 205), 'cornsilk3': (205, 200,
177), 'cornsilk4': (139, 136, 120), 'crimson': (220, 20, 60), 'cyan': (0, 255, 255),
'cyan1': (0, 255, 255), 'cyan2': (0, 238, 238), 'cyan3': (0, 205, 205), 'cyan4': (0,
139, 139), 'darkblue': (0, 0, 139), 'darkcyan': (0, 139, 139), 'darkgoldenrod': (184,
134, 11), 'darkgoldenrod1': (255, 185, 15), 'darkgoldenrod2': (238, 173, 14),
'darkgoldenrod3': (205, 149, 12), 'darkgoldenrod4': (139, 101, 8), 'darkgray': (169,
169, 169), 'darkgreen': (0, 100, 0), 'darkgrey': (169, 169, 169), 'darkkhaki': (189,
183, 107), 'darkmagenta': (139, 0, 139), 'darkolivegreen': (85, 107, 47),
'darkolivegreen1': (202, 255, 112), 'darkolivegreen2': (188, 238, 104),
'darkolivegreen3': (162, 205, 90), 'darkolivegreen4': (110, 139, 61), 'darkorange':
(255, 140, 0), 'darkorange1': (255, 127, 0), 'darkorange2': (238, 118, 0),
'darkorange3': (205, 102, 0), 'darkorange4': (139, 69, 0), 'darkorchid': (153, 50,
204), 'darkorchid1': (191, 62, 255), 'darkorchid2': (178, 58, 238), 'darkorchid3':
(154, 50, 205), 'darkorchid4': (104, 34, 139), 'darkred': (139, 0, 0), 'darksalmon':
(233, 150, 122), 'darkseagreen': (143, 188, 143), 'darkseagreen1': (193, 255, 193),
'darkseagreen2': (180, 238, 180), 'darkseagreen3': (155, 205, 155), 'darkseagreen4':
(105, 139, 105), 'darkslateblue': (72, 61, 139), 'darkslategray': (47, 79, 79),
'darkslategray1': (151, 255, 255), 'darkslategray2': (141, 238, 238), 'darkslategray3':
(121, 205, 205), 'darkslategray4': (82, 139, 139), 'darkslategrey': (47, 79, 79),
'darkturquoise': (0, 206, 209), 'darkviolet': (148, 0, 211), 'deeppink': (255, 20,
147), 'deeppink1': (255, 20, 147), 'deeppink2': (238, 18, 137), 'deeppink3': (205, 16,
118), 'deeppink4': (139, 10, 80), 'deepskyblue': (0, 191, 255), 'deepskyblue1': (0,
191, 255), 'deepskyblue2': (0, 178, 238), 'deepskyblue3': (0, 154, 205),
'deepskyblue4': (0, 104, 139), 'dimgray': (105, 105, 105), 'dimgrey': (105, 105, 105),
'dodgerblue': (30, 144, 255), 'dodgerblue1': (30, 144, 255), 'dodgerblue2': (28, 134,
238), 'dodgerblue3': (24, 116, 205), 'dodgerblue4': (16, 78, 139), 'firebrick': (178,
34, 34), 'firebrick1': (255, 48, 48), 'firebrick2': (238, 44, 44), 'firebrick3': (205,
38, 38), 'firebrick4': (139, 26, 26), 'floralwhite': (255, 250, 240), 'forestgreen':
(34, 139, 34), 'fuchsia': (255, 0, 255), 'gainsboro': (220, 220, 220), 'ghostwhite':
(248, 248, 255), 'gold': (255, 215, 0), 'gold1': (255, 215, 0), 'gold2': (238, 201,
0), 'gold3': (205, 173, 0), 'gold4': (139, 117, 0), 'goldenrod': (218, 165, 32),
'goldenrod1': (255, 193, 37), 'goldenrod2': (238, 180, 34), 'goldenrod3': (205, 155,
29), 'goldenrod4': (139, 105, 20), 'gray': (190, 190, 190), 'gray0': (0, 0, 0),
'gray1': (3, 3, 3), 'gray10': (26, 26, 26), 'gray100': (255, 255, 255), 'gray11':
(28, 28, 28), 'gray12': (31, 31, 31), 'gray13': (33, 33, 33), 'gray14': (36, 36, 36),
'gray15': (38, 38, 38), 'gray16': (41, 41, 41), 'gray17': (43, 43, 43), 'gray18':
(46, 46, 46), 'gray19': (48, 48, 48), 'gray2': (5, 5, 5), 'gray20': (51, 51, 51),
'gray21': (54, 54, 54), 'gray22': (56, 56, 56), 'gray23': (59, 59, 59), 'gray24':
```

X11 Color names to (XTerm-defined) RGB values from xorg-rgb/rgb.txt

hex_to_rgb(*hex_color: str*) → Tuple[int, int, int]

Convert hex color to RGB tuple.

Parameters

hex_color (*str*) – Hex color string. Accepts 3-digit (`#RGB`), 6-digit (`#RRGGBB`), or 12-digit (`#RRRRGGGGBBBB`, 16-bit per channel, converted to 8-bit). The # prefix is optional.

Returns

Tuple (r, g, b) with 8-bit values (0-255).

Raises

ValueError – If hex_color is not a valid hex color string.

rgb_to_hex(*red: int*, *green: int*, *blue: int*, *maybe_short: bool = False*) → str

Convert RGB values to hex color string.

Parameters

- **red** (*int*) – Red component (0-255).
- **green** (*int*) – Green component (0-255).
- **blue** (*int*) – Blue component (0-255).
- **maybe_short** (*bool*) – If True, return `#RGB` when possible. Shortening is only possible when each component has repeating hex digits (e.g., `0xaa` -> `a`).

Returns

Hex color string in `#RRGGBB` format (or `#RGB` if maybe_short).

xparse_color(*hex_component: str*, *bits: int = 16*) → int

Scale a hex color component to 16-bit or 8-bit per XParseColor specification.

XParseColor is the X11 color parsing standard used by terminals when responding to OSC 10/11 color queries. Each color component in the `rgb:R/G/B` response can be 1-4 hex digits, representing 4-bit to 16-bit precision.

XParseColor uses **scaling** (bit replication) to expand shorter values to 16-bit. This differs from HTML/CSS shorthand which uses digit doubling:

- XParseColor `rgb:` format (scaling): `f` -> `ffff`, `0f` -> `0f0f`
- HTML/CSS shorthand (doubling): `#f` -> `#ff`, `#0f` -> `#0f`

Scaling examples:

- 1 hex digit (4-bit): `f` -> `ffff` (15 -> 65535)
- 2 hex digits (8-bit): `e5` -> `e5e5` (229 -> 58853)
- 3 hex digits (12-bit): `abc` -> `abca` (2748 -> 43978)
- 4 hex digits (16-bit): `e5e5` -> `e5e5` (unchanged)

Parameters

- **hex_component** (*str*) – A single hex color component (1-4 hex digits).
- **bits** (*int*) – Output precision: 16 (default, 0-65535) or 8 (0-255).

Returns

Scaled integer value at the requested bit depth.

Raises
ValueError – If hex_component length is not 1-4 or bits is invalid.

15.3 dec_modes.py

Class definitions for DEC Private Modes and their Response values.

class DecModeResponse(*mode: int | DecPrivateMode*, *value: int*)

Container for DEC Private Mode query response.

Use helper properties *supported*, *enabled*, *permanent*, and *failed* to interpret responses rather than checking numeric values directly.

Initialize response for a DEC private mode query.

Parameters
- **mode** (*int*) – DEC private mode number
- **value** (*int*) – Response value from terminal

Raises
TypeError – If mode is not an integer

NOT_QUERIED = -2

NO_RESPONSE = -1

NOT_RECOGNIZED = 0

SET = 1

RESET = 2

PERMANENTLY_SET = 3

PERMANENTLY_RESET = 4

property mode: *DecPrivateMode*

The *DecPrivateMode* instance for this response.

Return type
DecPrivateMode

property description: str

Description of what this mode controls.

Return type
str

property value: int

Numeric response value for compatibility.

Prefer using helper properties like *enabled* instead of checking this value directly.

Return type
int

property recognized: bool

Check if the terminal recognizes this mode.

True for any DECRPM response value > 0 (SET, RESET, PERMANENTLY_SET, or PERMANENTLY_RESET). This indicates the terminal knows about the mode, but does not guarantee the application can use or control it.

Return type
bool

Returns
True if terminal acknowledges this mode

property supported: bool

Check if the mode is supported and usable by the terminal.

True for values SET (1), RESET (2), and PERMANENTLY_SET (3). PERMANENTLY_RESET (4) is excluded: the terminal acknowledges the mode but it is permanently off and cannot be enabled, so it is not usable by applications. See also xterm patch #395 and the Contour synchronized-output spec which both treat value 4 as effectively unsupported.

Return type
bool

Returns
True if mode is usable (changeable or permanently enabled)

property enabled: bool

Check if the mode is currently enabled.

Return type
bool

Returns
True if mode is set (temporarily or permanently)

property disabled: bool

Check if the mode is currently disabled.

Return type
bool

Returns
True if mode is reset (temporarily or permanently)

property changeable: bool

Check if the mode setting can be changed.

Return type
bool

Returns
True if mode can be toggled by applications

property permanent: bool

Check if the mode setting is permanent.

Return type
bool

Returns
True if mode cannot be changed by applications

property failed: bool

Check if the query failed.

Return type
bool

Returns
True if response indicates timeout or query failure

to_dict() → Dict[str, Any]

Return response as a dictionary.

Return type
dict

Returns
Dictionary with keys `value`, `value_description`, `mode_description`, `mode_name`, `supported`, `enabled`, and `changeable`.

class DecPrivateMode(*value: int*)

DEC Private Mode with mnemonic name and description.

Each instance provides:

- `value`: Numeric private mode identifier
- `name`: Mnemonic name or "UNKNOWN" for unrecognized modes
- *`long_description`*: Full description of mode functionality

Initialize DEC Private Mode with numeric identifier.

Parameters
value (*int*) – Numeric DEC private mode identifier

SpecialInternalLegacyCSIModifier = -3

SpecialInternalModifyOtherKeys = -2

SpecialInternalKitty = -1

DECCKM = 1

DECANM = 2

DECCOLM = 3

DECSCLM = 4

DECSCNM = 5

DECOM = 6

DECAWM = 7

DECARM = 8

DECINLM = 9

DECEDM = 10

DECLTM = 11

```
DECKANAM = 12

DECSCFDM = 13

DECTEM = 14

DECEKEM = 16

DECPFF = 18

DECPEX = 19

OV1 = 20

BA1 = 21

BA2 = 22

PK1 = 23

AH1 = 24

DECTCEM = 25

DECPSP = 27

DECPSM = 29

SHOW_SCROLLBAR_RXVT = 30

DECRLM = 34

DECHEBM = 35

DECHEM = 36

DECTEK = 38

DECCRNLM = 40

DECUPM = 41

DECNRCM = 42

DECGEPM = 43

DECGPCM = 44

DECGPCS = 45

DECGPBM = 46

DECGRPM = 47

DECTHAIM = 49

DECTHAICM = 50

DECBWRM = 51

DECOPM = 52
```

```
DEC131TM = 53

DECBPM = 55

DECNAKB = 57

DECIPEM = 58

DECKKDM = 59

DECHCCM = 60

DECVCCM = 61

DECPCCM = 64

DECBCMM = 65

DECNKM = 66

DECBKM = 67

DECKBUM = 68

DECVSSM = 69

DECFPM = 70

DECXRLM = 73

DECSDM = 80

DECKPM = 81

WY_52_LINE = 83

WYENAT_OFF = 84

REPLACEMENT_CHAR_COLOR = 85

DECTHAISCM = 90

DECNCSM = 95

DECRLCM = 96

DECCRTSM = 97

DECARSM = 98

DECMCM = 99

DECAAM = 100

DECCANSM = 101

DECNULM = 102

DECHDPXM = 103

DECESKM = 104
```

```
    DECOSCNM = 106

    DECNUMLK = 108

    DECCAPSLK = 109

    DECKLHIM = 110

    DECFWM = 111

    DECRPL = 112

    DECHWUM = 113

    DECATCUM = 114

    DECATCBM = 115

    DECBBSM = 116

    DECECM = 117

    MOUSE_REPORT_CLICK = 1000

    MOUSE_HILITE_TRACKING = 1001

    MOUSE_REPORT_DRAG = 1002

    MOUSE_ALL_MOTION = 1003

    FOCUS_IN_OUT_EVENTS = 1004

    MOUSE_EXTENDED_UTF8 = 1005

    MOUSE_EXTENDED_SGR = 1006

    ALT_SCROLL_XTERM = 1007

    SCROLL_ON_TTY_OUTPUT_RXVT = 1010

    SCROLL_ON_KEYPRESS_RXVT = 1011

    FAST_SCROLL = 1014

    MOUSE_URXVT = 1015

    MOUSE_SGR_PIXELS = 1016

    BOLD_ITALIC_HIGH_INTENSITY = 1021

    META_SETS_EIGHTH_BIT = 1034

    MODIFIERS_ALT_NUMLOCK = 1035

    META_SENDS_ESC = 1036

    KP_DELETE_SENDS_DEL = 1037

    ALT_SENDS_ESC = 1039

    KEEP_SELECTION_NO_HILITE = 1040
```

```
USE_CLIPBOARD_SELECTION = 1041

URGENCY_ON_CTRL_G = 1042

RAISE_ON_CTRL_G = 1043

REUSE_CLIPBOARD_DATA = 1044

EXTENDED_REVERSE_WRAPAROUND = 1045

ALT_SCREEN_BUFFER_SWITCH = 1046

ALT_SCREEN_BUFFER_XTERM = 1047

SAVE_CURSOR_DECSC = 1048

ALT_SCREEN_AND_SAVE_CLEAR = 1049

TERMINFO_FUNC_KEY_MODE = 1050

SUN_FUNC_KEY_MODE = 1051

HP_FUNC_KEY_MODE = 1052

SCO_FUNC_KEY_MODE = 1053

LEGACY_KBD_X11R6 = 1060

VT220_KBD_EMULATION = 1061

SIXEL_PRIVATE_PALETTE = 1070

BIDI_ARROW_KEY_SWAPPING = 1243

ITERM2_REPORT_KEY_UP = 1337

READLINE_MOUSE_BUTTON_1 = 2001

READLINE_MOUSE_BUTTON_2 = 2002

READLINE_MOUSE_BUTTON_3 = 2003

BRACKETED_PASTE = 2004

READLINE_CHARACTER_QUOTING = 2005

READLINE_NEWLINE_PASTING = 2006

SYNCHRONIZED_OUTPUT = 2026

GRAPHEME_CLUSTERING = 2027

TEXT_REFLOW = 2028

PASSIVE_MOUSE_TRACKING = 2029

REPORT_GRID_CELL_SELECTION = 2030

COLOR_PALETTE_UPDATES = 2031

IN_BAND_WINDOW_RESIZE = 2048
```

BRACKETED_PASTE_MIME = 5522

MIRROR_BOX_DRAWING = 2500

BIDI_AUTODETECTION = 2501

AMBIGUOUS_WIDTH_REPORTING = 7700

SCROLL_MARKERS = 7711

REWRAP_ON_RESIZE_MINTTY = 7723

APPLICATION_ESCAPE_KEY = 7727

ESC_KEY_SENDS_BACKSLASH = 7728

GRAPHICS_POSITION = 7730

ALT_MODIFIED_MOUSEWHEEL = 7765

SHOW_HIDE_SCROLLBAR = 7766

FONT_CHANGE_REPORTING = 7767

GRAPHICS_POSITION_2 = 7780

SHORTCUT_KEY_MODE = 7783

MOUSEWHEEL_REPORTING = 7786

APPLICATION_MOUSEWHEEL = 7787

BIDI_CURRENT_LINE = 7796

TTCTH = 8200

SIXEL_SCROLLING_LEAVES_CURSOR = 8452

CHARACTER_MAPPING_SERVICE = 8800

AMBIGUOUS_WIDTH_DOUBLE_WIDTH = 8840

WIN32_INPUT_MODE = 9001

KITTY_HANDLE_CTRL_C_Z = 19997

MINTTY_BIDI = 77096

INPUT_METHOD_EDITOR = 737769

property long_description: str

Full description of this DEC private mode's functionality.

Return type

str

Returns

Detailed description or "Unknown mode" for unrecognized modes

15.4 formatters.py

Sub-module providing sequence-formatting functions.

_build_known_capability_names() → frozenset[str]

Return frozenset of all terminal capability names known to jinxed.

_make_colors() → Set[str]

Return set of valid colors and their derivatives.

Return type
set

Returns
Color names with prefixes

```
COLORS: Set[str] = {'aliceblue', 'antiquewhite', 'antiquewhite1', 'antiquewhite2',
'antiquewhite3', 'antiquewhite4', 'aqua', 'aquamarine', 'aquamarine1', 'aquamarine2',
'aquamarine3', 'aquamarine4', 'azure', 'azure1', 'azure2', 'azure3', 'azure4', 'beige',
'bisque', 'bisque1', 'bisque2', 'bisque3', 'bisque4', 'black', 'blanchedalmond', 'blue',
'blue1', 'blue2', 'blue3', 'blue4', 'blueviolet', 'bright_black', 'bright_blue',
'bright_cyan', 'bright_green', 'bright_magenta', 'bright_red', 'bright_white',
'bright_yellow', 'brown', 'brown1', 'brown2', 'brown3', 'brown4', 'burlywood',
'burlywood1', 'burlywood2', 'burlywood3', 'burlywood4', 'cadetblue', 'cadetblue1',
'cadetblue2', 'cadetblue3', 'cadetblue4', 'chartreuse', 'chartreuse1', 'chartreuse2',
'chartreuse3', 'chartreuse4', 'chocolate', 'chocolate1', 'chocolate2', 'chocolate3',
'chocolate4', 'coral', 'coral1', 'coral2', 'coral3', 'coral4', 'cornflowerblue',
'cornsilk', 'cornsilk1', 'cornsilk2', 'cornsilk3', 'cornsilk4', 'crimson', 'cyan',
'cyan1', 'cyan2', 'cyan3', 'cyan4', 'darkblue', 'darkcyan', 'darkgoldenrod',
'darkgoldenrod1', 'darkgoldenrod2', 'darkgoldenrod3', 'darkgoldenrod4', 'darkgray',
'darkgreen', 'darkgrey', 'darkkhaki', 'darkmagenta', 'darkolivegreen', 'darkolivegreen1',
'darkolivegreen2', 'darkolivegreen3', 'darkolivegreen4', 'darkorange', 'darkorange1',
'darkorange2', 'darkorange3', 'darkorange4', 'darkorchid', 'darkorchid1', 'darkorchid2',
'darkorchid3', 'darkorchid4', 'darkred', 'darksalmon', 'darkseagreen', 'darkseagreen1',
'darkseagreen2', 'darkseagreen3', 'darkseagreen4', 'darkslateblue', 'darkslategray',
'darkslategray1', 'darkslategray2', 'darkslategray3', 'darkslategray4', 'darkslategrey',
'darkturquoise', 'darkviolet', 'deeppink', 'deeppink1', 'deeppink2', 'deeppink3',
'deeppink4', 'deepskyblue', 'deepskyblue1', 'deepskyblue2', 'deepskyblue3',
'deepskyblue4', 'dimgray', 'dimgrey', 'dodgerblue', 'dodgerblue1', 'dodgerblue2',
'dodgerblue3', 'dodgerblue4', 'firebrick', 'firebrick1', 'firebrick2', 'firebrick3',
'firebrick4', 'floralwhite', 'forestgreen', 'fuchsia', 'gainsboro', 'ghostwhite', 'gold',
'gold1', 'gold2', 'gold3', 'gold4', 'goldenrod', 'goldenrod1', 'goldenrod2',
'goldenrod3', 'goldenrod4', 'gray', 'gray0', 'gray1', 'gray10', 'gray100', 'gray11',
'gray12', 'gray13', 'gray14', 'gray15', 'gray16', 'gray17', 'gray18', 'gray19', 'gray2',
'gray20', 'gray21', 'gray22', 'gray23', 'gray24', 'gray25', 'gray26', 'gray27', 'gray28',
'gray29', 'gray3', 'gray30', 'gray31', 'gray32', 'gray33', 'gray34', 'gray35', 'gray36',
'gray37', 'gray38', 'gray39', 'gray4', 'gray40', 'gray41', 'gray42', 'gray43', 'gray44',
'gray45', 'gray46', 'gray47', 'gray48', 'gray49', 'gray5', 'gray50', 'gray51', 'gray52',
'gray53', 'gray54', 'gray55', 'gray56', 'gray57', 'gray58', 'gray59', 'gray6', 'gray60',
'gray61', 'gray62', 'gray63', 'gray64', 'gray65', 'gray66', 'gray67', 'gray68', 'gray69',
'gray7', 'gray70', 'gray71', 'gray72', 'gray73', 'gray74', 'gray75', 'gray76', 'gray77',
'gray78', 'gray79', 'gray8', 'gray80', 'gray81', 'gray82', 'gray83', 'gray84', 'gray85',
'gray86', 'gray87', 'gray88', 'gray89', 'gray9', 'gray90', 'gray91', 'gray92', 'gray93',
'gray94', 'gray95', 'gray96', 'gray97', 'gray98', 'gray99', 'green', 'green1', 'green2',
'green3', 'green4', 'greenyellow', 'grey', 'grey0', 'grey1', 'grey10', 'grey100',
'grey11', 'grey12', 'grey13', 'grey14', 'grey15', 'grey16', 'grey17', 'grey18', 'grey19',
'grey2', 'grey20', 'grey21', 'grey22', 'grey23', 'grey24', 'grey25', 'grey26', 'grey27',
'grey28', 'grey29', 'grey3', 'grey30', 'grey31', 'grey32', 'grey33', 'grey34', 'grey35',
'grey36', 'grey37', 'grey38', 'grey39', 'grey4', 'grey40', 'grey41', 'grey42', 'grey43',
'grey44', 'grey45', 'grey46', 'grey47', 'grey48', 'grey49', 'grey5', 'grey50', 'grey51',
'grey52', 'grey53', 'grey54', 'grey55', 'grey56', 'grey57', 'grey58', 'grey59', 'grey6',
'grey60', 'grey61', 'grey62', 'grey63', 'grey64', 'grey65', 'grey66', 'grey67', 'grey68',
'grey69', 'grey7', 'grey70', 'grey71', 'grey72', 'grey73', 'grey74', 'grey75', 'grey76',
'grey77', 'grey78', 'grey79', 'grey8', 'grey80', 'grey81', 'grey82', 'grey83', 'grey84',
'grey85', 'grey86', 'grey87', 'grey88', 'grey89', 'grey9', 'grey90', 'grey91', 'grey92',
'grey93', 'grey94', 'grey95', 'grey96', 'grey97', 'grey98', 'grey99', 'honeydew',
'honeydew1', 'honeydew2', 'honeydew3', 'honeydew4', 'hotpink', 'hotpink1', 'hotpink2',
'hotpink3', 'hotpink4', 'indianred', 'indianred1', 'indianred2', 'indianred3',
'indianred4', 'indigo', 'ivory', 'ivory1', 'ivory2', 'ivory3', 'ivory4', 'khaki',
'khaki1', 'khaki2', 'khaki3', 'khaki4', 'lavender', 'lavenderblush', 'lavenderblush1',
'lavenderblush2', 'lavenderblush3', 'lavenderblush4', 'lawngreen', 'lemonchiffon',
'lemonchiffon1', 'lemonchiffon2', 'lemonchiffon3', 'lemonchiffon4', 'lightblue',
'lightblue1', 'lightblue2', 'lightblue3', 'lightblue4', 'lightcoral', 'lightcyan',
'lightcyan1', 'lightcyan2', 'lightcyan3', 'lightcyan4', 'lightgoldenrod',
'lightgoldenrod1', 'lightgoldenrod2', 'lightgoldenrod3', 'lightgoldenrod4',
'lightgoldenrodyellow', 'lightgray', 'lightgreen', 'lightgrey', 'lightpink',
```

Valid colors and their background (on), bright, and bright-background derivatives.

COMPOUNDABLES: Set[str] = {'blink', 'bold', 'dim', 'italic', 'overline', 'reverse', 'standout', 'strikethrough', 'underline'}

Attributes that may be compounded with colors, by underscore, such as 'reverse_indigo'.

_tparm_cached(*cap: str*, *args: Tuple[object, ...]*) → str

Memoized call to `jinxed.tparm()`.

_tparm(*cap: str*, *args: Tuple[object, ...]*) → str

Return evaluated capability `cap` for `args`, memoized where safe.

`jinxed.tparm()` is a pure-python reimplementation of `tparm(3)`, but a bit slow. tparm() is pure function, and so results are cached.

Parameters

- **cap** (*str*) – parameterized capability string.
- **args** (*tuple*) – arguments for `cap`.

Return type
str

Returns
evaluated capability.

class ParameterizingString(*cap: str*, *normal: str = ''*, *name: str = '<not specified>'*)

A Unicode string which can be called as a parameterizing termcap.

For example:

```
>>> from blessed import Terminal
>>> term = Terminal()
>>> color = ParameterizingString(term.color, term.normal, 'color')
>>> color(9)('color #9')
'\x1b[91mcolor #9\x1b(B\x1b[m'
```

Class constructor accepting 3 positional arguments.

Parameters

- **cap** (*str*) – parameterized string suitable for jinxed.tparm()
- **normal** (*str*) – terminating sequence for this capability (optional).
- **name** (*str*) – name of this terminal capability (optional).

__call__(**args: object*) → *FormattingString*

Returning *FormattingString* instance for given parameters.

Return evaluated terminal capability (self), receiving arguments `*args`, followed by the terminating sequence (self.normal) into a *FormattingString* capable of being called.

Raises

- **TypeError** – Mismatch between capability and arguments
- **jinxed.error** – `jinxed.tparm()` raised an exception

Return type
FormattingString or *NullCallableString*

Returns
Callable string for given parameters

class ParameterizingProxyString(*fmt_pair: Tuple[str, Callable[[...], Tuple[object, ...]]], normal: str = '', name: str = '<not specified>'*)

A Unicode string which can be called to proxy missing termcap entries.

This class mirrors the behavior of *ParameterizingString*, except that instead of a capability name, receives a format string, and callable to filter the given positional `*args` of *ParameterizingProxyString.__call__()* into a terminal sequence.

For example:

```
>>> from blessed import Terminal
>>> term = Terminal('screen')
>>> hpa = ParameterizingString(term.hpa, term.normal, 'hpa')
>>> hpa(9)
''
>>> fmt = '\x1b[{0}G'
>>> fmt_arg = lambda *arg: (arg[0] + 1,)
>>> hpa = ParameterizingProxyString((fmt, fmt_arg), term.normal, 'hpa')
>>> hpa(9)
'\x1b[10G'
```

Class constructor accepting 4 positional arguments.

Parameters

- **fmt_pair** (*tuple*) – Two element tuple containing: - format string suitable for displaying terminal sequences - callable suitable for receiving __call__ arguments for formatting string
- **normal** (*str*) – terminating sequence for this capability (optional).
- **name** (*str*) – name of this terminal capability (optional).

__call__(**args: object*) → *FormattingString*

Returning *FormattingString* instance for given parameters.

Arguments are determined by the capability. For example, hpa (move_x) receives only a single integer, whereas cup (move) receives two integers. See documentation in terminfo(5) for the given capability.

Return type
FormattingString

Returns
Callable string for given parameters

class FormattingString(*sequence: str, normal: str = ''*)

A Unicode string which doubles as a callable.

This is used for terminal attributes, so that it may be used both directly, or as a callable. When used directly, it simply emits the given terminal sequence. When used as a callable, it wraps the given (string) argument with the 2nd argument used by the class constructor:

```
>>> from blessed import Terminal
>>> term = Terminal()
>>> style = FormattingString(term.bright_blue, term.normal)
>>> print(repr(style))
'\x1b[94m'
```

(continues on next page)

(continued from previous page)

```
>>> style('Big Blue')
'\x1b[94mBig Blue\x1b(B\x1b[m'
```

Class constructor accepting 2 positional arguments.

Parameters

- **sequence** (*str*) – terminal attribute sequence.
- **normal** (*str*) – terminating sequence for this attribute (optional).

__call__(**args: str*) → str

Return `text` joined by `sequence` and `normal`.

Raises
TypeError – Not a string type

Return type
str

Returns
Arguments wrapped in sequence and normal

class FormattingOtherString(*direct:* ParameterizingString, *target:* ParameterizingString)

A Unicode string which doubles as a callable for another sequence when called.

This is used for the *move_up()*, `down`, `left`, and `right()` family of functions:

```
>>> from blessed import Terminal
>>> term = Terminal()
>>> move_right = FormattingOtherString(term.cuf1, term.cuf)
>>> print(repr(move_right))
'\x1b[C'
>>> print(repr(move_right(666)))
'\x1b[666C'
>>> print(repr(move_right()))
'\x1b[C'
```

Class constructor accepting 2 positional arguments.

Parameters

- **direct** (*str*) – capability name for direct formatting, eg `('x' + term.right)`.
- **target** (*str*) – capability name for callable, eg `('x' + term.right(99))`.

__call__(**args: object*) → str

Return `text` by `target`.

class NullCallableString

A dummy callable Unicode alternative to *FormattingString*.

This is used for colors on terminals that do not support colors, it is just a basic form of unicode that may also act as a callable.

Class constructor.

__call__(**args: str*) → str

Allow empty string to be callable, returning given string, if any.

When called with an int as the first arg, return an empty Unicode. An int is a good hint that I am a *ParameterizingString*, as there are only about half a dozen string- returning capabilities listed in terminfo(5) which accept non-int arguments, they are seldom used.

When called with a non-int as the first arg (no no args at all), return the first arg, acting in place of *FormattingString* without any attributes.

get_proxy_string(*term:* Terminal, *attr: str*) → None

Proxy and return callable string for proxied attributes.

split_compound(*compound: str*) → List[str]

Split compound formatting string into segments.

```
>>> split_compound('bold_underline_bright_blue_on_red')
['bold', 'underline', 'bright_blue', 'on_red']
```

Parameters
compound (*str*) – a string that may contain compounds, separated by underline (_).

Return type
list

Returns
List of formatting string segments

resolve_capability(*term:* Terminal, *attr: str*) → str

Resolve a raw terminal capability using `tigetstr()`.

Parameters

- **term** (Terminal) – *Terminal* instance.
- **attr** (*str*) – terminal capability name.

Returns
string of the given terminal capability named by `attr`, which may be empty ('') if not found or not supported by the given *kind*.

Return type
str

resolve_color(*term:* Terminal, *color: str*) → *NullCallableString* | *FormattingString*

Resolve a simple color name to a callable capability.

This function supports *resolve_attribute()*.

Parameters

- **term** (Terminal) – *Terminal* instance.
- **color** (*str*) – any string found in set *COLORS*.

Returns
a string class instance which emits the terminal sequence for the given color, and may be used as a callable to wrap the given string with such sequence.

Returns
NullCallableString when *number_of_colors* is 0, otherwise *FormattingString*.

Return type
NullCallableString or *FormattingString*

resolve_attribute(*term:* Terminal, *attr: str*) → *ParameterizingString* | *FormattingString*

Resolve a terminal attribute name into a capability class.

Parameters

- **term** (*Terminal*) – *Terminal* instance.
- **attr** (*str*) – Sugary, ordinary, or compound formatted terminal capability, such as "red_on_white", "normal", "red", or "bold_on_black".

Returns

a string class instance which emits the terminal sequence for the given terminal capability, or may be used as a callable to wrap the given string with such sequence.

Returns

NullCallableString when *number_of_colors* is 0, otherwise *FormattingString*.

Return type

NullCallableString or *FormattingString*

15.5 keyboard.py

Sub-module providing 'keyboard awareness'.

class Keystroke(*ucs: str = ''*, *code: int* | *None = None*, *name: str* | *None = None*, *mode: int* | *None = None*, *match: Any = None*)

A unicode-derived class for describing a single "keystroke".

A class instance describes a single keystroke received on input, which may contain multiple characters as a multibyte sequence, which is indicated by properties *is_sequence* returning `True`. Note that keystrokes may also represent mouse input, bracketed paste, or focus in/out events depending on enabled terminal modes.

The string *name* of the sequence is used to identify in code logic, such as `'KEY_LEFT'` to represent a common and human-readable form of the Keystroke this class instance represents.

Class constructor.

_name: `str | None = None`

_code: `int | None = None`

_mode: `int | None = None`

_match: `Any = None`

_modifiers: `int = 1`

static __new__(*cls: Type[_T]*, *ucs: str = ''*, *code: int* | *None = None*, *name: str* | *None = None*, *mode: int* | *None = None*, *match: Any = None*) → _T

Class constructor.

static _infer_modifiers(*ucs: str*, *mode: int* | *None*, *match: Any*) → int

Infer modifiers from keystroke data.

Returns modifiers in standard format: 1 + bitwise OR of modifier flags.

property is_sequence: `bool`

Whether the value represents a multibyte sequence (bool).

_get_modified_keycode_name() → str | None

Get base name for modern/legacy CSI sequence with modifiers.

Returns name like 'KEY_CTRL_ALT_F1' or 'KEY_SHIFT_UP' without event-type suffix. The suffix is applied by *name*.

_get_kitty_protocol_name() → str | None

Get base name for Kitty keyboard protocol letter/digit/symbol.

Returns name like 'KEY_CTRL_ALT_A', 'KEY_ALT_SHIFT_5', 'KEY_LEFT_SQUARE_BRACKET' without event-type suffix. The suffix is applied by *name*.

_get_control_char_name() → str | None

Get name for single-character control sequences.

Returns name like 'KEY_CTRL_A' or 'KEY_CTRL_SPACE'.

_get_control_symbol(*char_code: int*) → str

Get control symbol for a character code.

Returns symbol like 'A' for Ctrl+A, 'SPACE' for Ctrl+Space, 'BACKSPACE' for Ctrl+H, etc.

_get_alt_only_control_name(*char_code: int*) → str | None

Get name for Alt-only special control characters.

Returns names like 'KEY_ALT_ESCAPE', 'KEY_ALT_BACKSPACE', etc.

_get_meta_escape_name() → str | None

Get name for metaSendsEscape sequences (ESC + char).

Returns name like 'KEY_ALT_A', 'KEY_ALT_SHIFT_Z', 'KEY_CTRL_ALT_C', or 'KEY_ALT_ESCAPE'.

_get_mouse_event_name() → str | None

Get name for mouse events.

Returns name like 'MOUSE_LEFT', 'MOUSE_CTRL_LEFT', 'MOUSE_SCROLL_UP', 'MOUSE_LEFT_RELEASED', 'MOUSE_MOTION', 'MOUSE_RIGHT_MOTION', etc.

_get_focus_event_name() → str | None

Get name for focus events.

Returns 'FOCUS_IN' or 'FOCUS_OUT'.

_get_bracketed_paste_name() → str | None

Get name for bracketed paste events.

Returns 'BRACKETED_PASTE'.

_resolve_name() → str | None

Resolve key name without event-type suffix.

Shared by *name* and *key_name*. The two keyboard protocol helpers (_get_modified_keycode_name, _get_kitty_protocol_name) return the base name only; *name* appends the suffix.

property name: str | None

Special application key name.

This is the best equality attribute to use for special keys, as raw string value of the 'F1' key can be received in many different values.

The 'name' property will return a reliable constant, eg. `'KEY_F1'`.

The name supports "modifiers", such as `'KEY_CTRL_F1'`, `'KEY_CTRL_ALT_F1'`, `'KEY_CTRL_ALT_SHIFT_F1'`

When using a keyboard protocol with event reporting, names include an event-type suffix: `'KEY_CTRL_J_REPEATED'` for repeat and `'KEY_CTRL_J_RELEASED'` for release events. Press events have no suffix. See *key_name* for the same name without the suffix.

For mouse events, the name includes the `'MOUSE_'` prefix followed by the button/action name, such as `'MOUSE_LEFT'`, `'MOUSE_MOTION'`, `'MOUSE_RIGHT_MOTION'`, `'MOUSE_LEFT_RELEASED'`.

For other DEC events:

- Focus events: 'FOCUS_IN' or 'FOCUS_OUT'
- Bracketed paste: 'BRACKETED_PASTE'
- Resize events: 'RESIZE_EVENT'

For terminal query responses:

- Cursor position report: 'CPR_RESPONSE' (row >= 2 only; row 1 is ambiguous with F3+modifier)

When non-None, all phrases begin with either 'KEY', 'MOUSE', 'FOCUS_IN', 'FOCUS_OUT', 'BRACKETED_PASTE', or 'RESIZE_EVENT', with one exception: 'CSI' is returned for '\x1b[' in legacy (non-kitty-protocol) mode to indicate the beginning of a presumed unsupported input sequence. In kitty keyboard protocol mode, the '[' key uses the name 'KEY_LEFT_SQUARE_BRACKET' (with modifier and event-type suffixes as appropriate).

If this value is None, then it can probably be assumed that the value is an unsurprising textual character without any modifiers, like the letter `'a'`.

property key_name: str | None

Key identity without event-type suffix.

Like *name*, but without `_RELEASED` and `_REPEATED` suffixes, so that press, repeat, and release events for the same key all return the same value. Useful for key-map lookups when tracking press/release pairs.

Return type
str or None

property code: int | None

Legacy curses-alike keycode value (int).

property modifiers: int

Modifier flags in standard keyboard protocol format.

Return type
int

Returns
Standard-style modifiers value (1 means no modifiers)

The value is 1 + bitwise OR of modifier flags:

- shift: 0b1 (1)
- alt: 0b10 (2)
- ctrl: 0b100 (4)
- super: 0b1000 (8)
- hyper: 0b10000 (16)
- meta: 0b100000 (32)

- caps_lock: 0b1000000 (64)
- num_lock: 0b10000000 (128)

property modifiers_bits: int

Raw modifier bit flags without the +1 offset.

Return type
int

Returns
Raw bitwise OR of modifier flags (0 means no modifiers)

property _shift: bool

Whether the shift modifier is active.

property _alt: bool

Whether the alt modifier is active.

property _ctrl: bool

Whether the ctrl modifier is active.

property _super: bool

Whether the super (Windows/Cmd) modifier is active.

property _hyper: bool

Whether the hyper modifier is active.

property _meta: bool

Whether the meta modifier is active.

property _caps_lock: bool

Whether caps lock was known to be active during this sequence.

property _num_lock: bool

Whether num lock was known to be active during this sequence.

property uses_keyboard_protocol: bool

Whether this keystroke uses a special keyboard protocol mode.

Returns True for Kitty, ModifyOtherKeys, or LegacyCSIModifier protocols, which use negative mode values (SpecialInternalKitty=-1, SpecialInternalModifyOtherKeys=-2, SpecialInternalLegacyCSIModifier=-3).

Return type
bool

Returns
True if using special keyboard protocol mode

property pressed: bool

Whether this is a key press event.

Return type
bool

Returns
True if this is a key press event (event_type=1 or not specified), False for repeat or release events

property repeated: bool

Whether this is a key repeat event.

Return type
bool

Returns
True if this is a key repeat event (event_type=2), False otherwise

property released: bool

Whether this is a key release event.

Return type
bool

Returns
True if this is a key release event (event_type=3), False otherwise

_is_escape_sequence(*length: int = 2*) → bool

Check if keystroke is an escape sequence of given length.

Parameters
length (*int*) – Expected length of escape sequence (default 2)

Return type
bool

Returns
True if keystroke matches ESC + (length-1) chars pattern

static _make_expected_bits(*tokens_modifiers: List[str]*) → int

Build expected modifier bits from token list.

_make_effective_bits() → int

Returns modifier bits stripped of caps_lock and num_lock.

static _get_keycode_by_name(*key_name: str*) → int | None

Get keycode value for a given key name.

_build_appkeys_predicate(*tokens_modifiers: List[str]*, *key_name: str*, *event_type: str | None = None*) → Callable[[str | None, bool], bool]

Build a predicate function for application keys.

_build_alphanum_predicate(*tokens_modifiers: List[str]*) → Callable[[str | None, bool], bool]

Build a predicate function for modifier checking of alphanumeric input.

_get_plain_char_value() → str | None

Get value for plain printable characters.

Returns the character as-is if it's a single printable character.

_get_escape_sequence_value() → str | None

Get value for ESC+char sequences (Alt, Ctrl+Alt combinations).

Handles Alt+printable, Alt-only special keys, and Ctrl+Alt sequences. Returns the base character or empty string for application keys.

_get_ctrl_sequence_value() → str | None

Get value for Ctrl+char sequences.

Maps control characters back to their base characters.

_get_protocol_value() → str | None

Get value for Kitty or ModifyOtherKeys protocol sequences.

Extracts the character from modern keyboard protocols.

_get_ascii_value() → str | None

Get value for keys matched by curses-imitated keycodes.

property value: str

The textual character represented by this keystroke.

Return type
str

Returns
For text keys, returns the base character (ignoring modifiers). For application keys and sequences, returns empty string ‘’. For release events, always returns empty string.

Some Examples,

- Plain text: ‘a’, ‘A’, ‘1’, ‘;’, ‘ ‘, ‘’, emoji with ZWJ sequences
- Alt+printable: Alt+a -> ‘a’, Alt+A -> ‘A’
- Ctrl+letter: Ctrl+A -> ‘a’, Ctrl+Z -> ‘z’
- Ctrl+symbol: Ctrl+@ -> ‘@’, Ctrl+? -> ‘?’, Ctrl+[-> ‘[’
- Control chars: ‘t’, ‘n’, ‘x08’, ‘x1b’ (for Enter/Tab/Backspace/Escape keycodes)
- Application keys: KEY_UP, KEY_F1, etc. -> ‘’
- Release events: always -> ‘’

property key_value: str

Character for this key, even for release events.

Like *value*, but does not suppress the character for release events. For press and repeat events, identical to *value*.

Return type
str

property mode: *DecPrivateMode* | None

DEC Private Mode associated with this keystroke, if any.

Return type
blessed.dec_modes.DecPrivateMode or None

Returns
The *DecPrivateMode* enum value associated with this keystroke, or `None` if this is not a DEC mode event.

> **Note**
>
> Mode names beginning SpecialInternal with negative values (-1, -2, -3) are used to track how some kinds of application keys are matched.

property mouse_yx: Tuple[int, int]

Mouse position as (y, x) tuple for mouse events.

This is particularly useful with terminal movement functions: `term.move_yx(*keystroke.mouse_yx)`

Return type
tuple of (int, int)

Returns
(y, x) coordinate tuple (0-indexed) for mouse events, or `(-1, -1)` if not a mouse event

property mouse_xy: Tuple[int, int]

Mouse position as (x, y) tuple for mouse events.

Return type
tuple of (int, int)

Returns
(x, y) coordinate tuple (0-indexed) for mouse events, or `(-1, -1)` if not a mouse event

property cpr_yx: Tuple[int, int]

Cursor position as (y, x) tuple for Cursor Position Report.

Return type
tuple of (int, int)

Returns
(y, x) coordinate tuple (0-indexed) for cursor position report, or `(-1, -1)` if not a CPR_RESPONSE

property cpr_xy: Tuple[int, int]

Cursor position as (x, y) tuple for Cursor Position Report.

Return type
tuple of (int, int)

Returns
(x, y) coordinate tuple (0-indexed) for cursor position report, or `(-1, -1)` if not a CPR_RESPONSE

property xtgettcap: Dict[str, str]

Capabilities reported by an XTGETTCAP response.

Return type
dict

Returns
mapping of terminal capability name to value.

property text: str | None

Pasted text for bracketed paste events.

Return type
str or None

Returns
The pasted text for `BRACKETED_PASTE` events, or `None` if not a bracketed paste event

property _mode_values: *BracketedPasteEvent* | *MouseEvent* | *FocusEvent* | ResizeEvent | None

Return structured data for DEC private mode events (private API).

Returns a namedtuple with parsed event data for supported *DecPrivateMode* modes:

- `BRACKETED_PASTE`: *BracketedPasteEvent* with `text` field
- `MOUSE_EXTENDED_SGR`, `MOUSE_ALL_MOTION`, `MOUSE_REPORT_DRAG`, and `MOUSE_REPORT_CLICK` events: `MouseEvent` with button, coordinates, and modifier flags

- FOCUS_IN_OUT_EVENTS: *FocusEvent* with gained boolean field
- IN_BAND_WINDOW_RESIZE: ResizeEvent with dimension fields

Return type
namedtuple or None

Returns
Structured event data for this DEC mode event, or None if this keystroke is not a DEC mode event or the mode is not supported

_parse_focus() → *FocusEvent*

Parse focus event from stored regex match.

_parse_bracketed_paste() → *BracketedPasteEvent*

Parse bracketed paste event from stored regex match.

_parse_resize() → ResizeEvent

Parse resize event from stored regex match.

get_keyboard_codes() → Dict[int, str]

Return mapping of keycode integer values paired by their curses key name.

Return type
dict

Returns
Dictionary of (code, name) pairs for curses keyboard constant values and their mnemonic name. Such as key 260, with the value of its identity, 'KEY_LEFT'.

These keys are derived from the attributes by the same of the curses(jinxed) module, with the following exceptions:

- KEY_DELETE in place of KEY_DC
- KEY_INSERT in place of KEY_IC
- KEY_PGUP in place of KEY_PPAGE
- KEY_PGDOWN in place of KEY_NPAGE
- KEY_ESCAPE in place of KEY_EXIT
- KEY_SUP in place of KEY_SR
- KEY_SDOWN in place of KEY_SF

This function is the inverse of get_curses_keycodes(). With the given override "mixins" listed above, the keycode for the delete key will map to our imaginary KEY_DELETE mnemonic, effectively erasing the phrase KEY_DC from our code vocabulary for anyone that wishes to use the return value to determine the key-name by keycode.

get_keyboard_sequences(*term:* Terminal) → OrderedDict[str, int]

Return mapping of keyboard sequences paired by keycodes.

Parameters
term (*blessed.Terminal*) – *Terminal* instance.

Returns
mapping of keyboard unicode sequences paired by keycodes as integer. This is used as the argument mapper to the supporting function resolve_sequence().

Return type
OrderedDict

Initialize and return a keyboard map and sequence lookup table, (sequence, keycode) from *Terminal* instance `term`, where `sequence` is a multibyte input sequence of unicode characters, such as `'\x1b[D'`, and `keycode` is an integer value, matching curses constant such as term.KEY_LEFT.

The return value is an OrderedDict instance, with their keys sorted longest-first.

class KittyKeyEvent(*unicode_key*, *shifted_key*, *base_key*, *modifiers*, *event_type*, *int_codepoints*)

Create new instance of KittyKeyEvent(unicode_key, shifted_key, base_key, modifiers, event_type, int_codepoints)

static __new__(*_cls*, *unicode_key*, *shifted_key*, *base_key*, *modifiers*, *event_type*, *int_codepoints*)

Create new instance of KittyKeyEvent(unicode_key, shifted_key, base_key, modifiers, event_type, int_codepoints)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('unicode_key', 'shifted_key', 'base_key', 'modifiers', 'event_type', 'int_codepoints')

classmethod _make(*iterable*)

Make a new KittyKeyEvent object from a sequence or iterable

_replace(***kwds*)

Return a new KittyKeyEvent object replacing specified fields with new values

base_key

Alias for field number 2

event_type

Alias for field number 4

int_codepoints

Alias for field number 5

modifiers

Alias for field number 3

shifted_key

Alias for field number 1

unicode_key

Alias for field number 0

class ModifyOtherKeysEvent(*key*, *modifiers*)

Create new instance of ModifyOtherKeysEvent(key, modifiers)

static __new__(*_cls*, *key*, *modifiers*)

Create new instance of ModifyOtherKeysEvent(key, modifiers)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('key', 'modifiers')

classmethod _make(*iterable*)

Make a new ModifyOtherKeysEvent object from a sequence or iterable

_replace(**kwds*)

Return a new ModifyOtherKeysEvent object replacing specified fields with new values

key

Alias for field number 0

modifiers

Alias for field number 1

class LegacyCSIKeyEvent(*kind*, *key_id*, *modifiers*, *event_type*)

Create new instance of LegacyCSIKeyEvent(kind, key_id, modifiers, event_type)

static __new__(*_cls*, *kind*, *key_id*, *modifiers*, *event_type*)

Create new instance of LegacyCSIKeyEvent(kind, key_id, modifiers, event_type)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('kind', 'key_id', 'modifiers', 'event_type')

classmethod _make(*iterable*)

Make a new LegacyCSIKeyEvent object from a sequence or iterable

_replace(**kwds*)

Return a new LegacyCSIKeyEvent object replacing specified fields with new values

event_type

Alias for field number 3

key_id

Alias for field number 1

kind

Alias for field number 0

modifiers

Alias for field number 2

class KittyKeyboardProtocol(*value: int*)

Represents Kitty keyboard protocol flags.

Encapsulates the integer flag value returned by Kitty keyboard protocol queries and provides properties for individual flag bits and a method to convert back to enable_kitty_keyboard() arguments.

Initialize with raw integer flag value.

Parameters

value (*int*) – Raw integer flags value from Kitty keyboard protocol query

property disambiguate: bool

Whether disambiguated escape codes are enabled (bit 1).

property report_events: bool

Whether key repeat and release events are reported (bit 2).

property report_alternates: bool

Whether shifted and base layout keys are reported for shortcuts (bit 4).

property report_all_keys: bool

Whether all keys are reported as escape codes (bit 8).

property report_text: bool

Whether associated text is reported with key events (bit 16).

make_arguments() → Dict[str, bool]

Return dictionary of arguments suitable for enable_kitty_keyboard().

Return type
dict

Returns
Dictionary with boolean flags suitable for passing as keyword arguments to enable_kitty_keyboard()

class DeviceAttribute(*raw: str*, *service_class: int*, *extensions: List[int] | None*)

Represents a terminal's Device Attributes (DA1) response.

Device Attributes queries allow discovering terminal capabilities and type. The primary DA1 query sends CSI c and expects a response like:

```
CSI ? Psc ; Ps1 ; Ps2 ; ... ; Psn c
```

Where Psc is the service class (architectural class) and Ps1...Psn are supported extensions/capabilities.

Initialize DeviceAttribute instance.

Parameters

- **raw** (*str*) – Original response string from terminal
- **service_class** (*int*) – Service class number (first parameter)
- **extensions** (*list*) – List of extension numbers (remaining parameters)

RE_RESPONSE = re.compile('\\x1b\\[\\?([0-9]+)((?:;[0-9]+)*);*c')

RE_RESPONSE_CTERM = re.compile('\\x1b\\[=67;84;101;114;109(?:;[0-9]+)*;*c')

property supports_sixel: bool

Whether the terminal supports sixel graphics.

Return type
bool

Returns
True if extension 4 (sixel) is present in device attributes

property supports_osc52: bool

Whether the terminal advertises OSC 52 clipboard support.

Extension 52 in DA1 indicates the terminal supports writing to the system clipboard via the OSC 52 protocol.

Return type
bool

Returns
True if extension 52 is present in device attributes

classmethod from_match(*match: Match[str]*) → *DeviceAttribute*

Create DeviceAttribute from regex match object.

Parameters
match (*re.Match*) – Regex match object with groups for service_class and extensions

Return type
DeviceAttribute

Returns
DeviceAttribute instance parsed from match

class SoftwareVersion(*raw: str*, *name: str*, *version: str*)

Represents a terminal's software name and version from XTVERSION response.

Initialize SoftwareVersion instance.

Parameters

- **raw** (*str*) – Original response string from terminal
- **name** (*str*) – Software name (e.g., "kitty", "XTerm")
- **version** (*str*) – Version string (e.g., "0.24.2", "367") or empty string if no version

RE_RESPONSE = re.compile('\\x1bP>\\|(.+?)\\x1b\\\\')

classmethod from_match(*match: Match[str]*) → *SoftwareVersion*

Create SoftwareVersion from regex match object.

Parameters
match (*re.Match*) – Regex match object with group for software text

Return type
SoftwareVersion

Returns
SoftwareVersion instance parsed from match

static _parse_text(*text: str*) → Tuple[str, str]

Parse software name and version from text.

Parsing logic (in order): 1. Check for space-separated format: "tmux 3.2a" or "X.Org 7.7.0(370)" 2. Check for parentheses format: "kitty(0.24.2)" 3. Name-only format: "software" (version is empty string)

Parameters
text (*str*) – Text from XTVERSION response

Return type
tuple

Returns
Tuple of (name, version) where both are strings

class BracketedPasteEvent(*text*)

Create new instance of BracketedPasteEvent(text,)

static __new__(*_cls*, *text*)
Create new instance of BracketedPasteEvent(text,)

_asdict()
Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('text',)

classmethod _make(*iterable*)
Make a new BracketedPasteEvent object from a sequence or iterable

_replace(***kwds*)
Return a new BracketedPasteEvent object replacing specified fields with new values

text
Alias for field number 0

class FocusEvent(*gained*)
Create new instance of FocusEvent(gained,)

static __new__(*_cls*, *gained*)
Create new instance of FocusEvent(gained,)

_asdict()
Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('gained',)

classmethod _make(*iterable*)
Make a new FocusEvent object from a sequence or iterable

_replace(***kwds*)
Return a new FocusEvent object replacing specified fields with new values

gained
Alias for field number 0

class SyncEvent(*begin*)
Create new instance of SyncEvent(begin,)

static __new__(*_cls*, *begin*)
Create new instance of SyncEvent(begin,)

_asdict()
Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('begin',)

classmethod _make(*iterable*)
Make a new SyncEvent object from a sequence or iterable

_replace(***kwds*)
Return a new SyncEvent object replacing specified fields with new values

begin

Alias for field number 0

DEFAULT_SEQUENCE_MIXIN = (('\n', 343), ('\r', 343), ('\x08', 263), ('\t', 512), ('\x1b', 361), ('\x7f', 263), ('\x1b[A', 259), ('\x1b[B', 258), ('\x1b[C', 261), ('\x1b[D', 260), ('\x1b[E', 350), ('\x1b[1;2A', 337), ('\x1b[1;2B', 336), ('\x1b[1;2C', 402), ('\x1b[1;2D', 393), ('\x1b[F', 360), ('\x1b[H', 262), ('\x1b[K', 360), ('\x1b[U', 338), ('\x1b[V', 339), ('\x1bOM', 343), ('\x1bOj', 513), ('\x1bOk', 514), ('\x1bOl', 515), ('\x1bOm', 516), ('\x1bOn', 517), ('\x1bOo', 518), ('\x1bOX', 519), ('\x1bOp', 520), ('\x1bOq', 521), ('\x1bOr', 522), ('\x1bOs', 523), ('\x1bOt', 524), ('\x1bOu', 525), ('\x1bOv', 526), ('\x1bOw', 527), ('\x1bOx', 528), ('\x1bOy', 529), ('\x1b[1~', 262), ('\x1b[2~', 331), ('\x1b[3~', 330), ('\x1b[4~', 360), ('\x1b[5~', 339), ('\x1b[6~', 338), ('\x1b[7~', 262), ('\x1b[8~', 360), ('\x1b[OA', 259), ('\x1b[OB', 258), ('\x1b[OC', 261), ('\x1b[OD', 260), ('\x1b[OF', 360), ('\x1b[OH', 262), ('\x1bOP', 265), ('\x1bOQ', 266), ('\x1bOR', 267), ('\x1bOS', 268), ('\x1b[P', 265), ('\x1b[Q', 266), ('\x1b[13~', 267), ('\x1b[S', 268))

In a perfect world, terminal emulators would always send exactly what the terminfo(5) capability database plans for them, accordingly by the value of the `TERM` name they declare.

But this isn't a perfect world. Many vt220-derived terminals, such as those declaring 'xterm', will continue to send vt220 codes instead of their native-declared codes, for backwards-compatibility.

This goes for many: rxvt, putty, iTerm.

These "mixins" are used for *all* terminals, regardless of their type.

Furthermore, curses does not provide sequences sent by the keypad, at least, it does not provide a way to distinguish between keypad 0 and numeric 0.

CURSES_KEYCODE_OVERRIDE_MIXIN = (('KEY_DELETE', 330), ('KEY_INSERT', 331), ('KEY_PGUP', 339), ('KEY_PGDOWN', 338), ('KEY_ESCAPE', 361), ('KEY_SUP', 337), ('KEY_SDOWN', 336), ('KEY_UP_LEFT', 348), ('KEY_UP_RIGHT', 349), ('KEY_CENTER', 350), ('KEY_BEGIN', 354), ('KEY_DOWN_LEFT', 351), ('KEY_DOWN_RIGHT', 352))

Override mixins for a few curses constants with easier mnemonics: there may only be a 1:1 mapping when only a keycode (int) is given, where these phrases are preferred.

15.6 line_editor.py

Headless line editor with history, auto-suggest, and grapheme-aware editing.

This module provides *LineEditor* for single-line input with readline-style editing and *LineHistory* for command recall.

class DisplayState(*text: str = ''*, *cursor: int = 0*, *suggestion: str = ''*, *overflow_left: bool = False*, *overflow_right: bool = False*, *text_sgr: str = ''*, *suggestion_sgr: str = ''*, *bg_sgr: str = ''*, *ellipsis_sgr: str = ''*)

Current visual state of the editor for rendering.

text: str = ''

Visible buffer text (masked in password mode, clipped when scrolling).

cursor: int = 0

Cursor column within the visible window.

suggestion: str = ''

Auto-suggest suffix (rendered dim/grey after text).

overflow_left: bool = False

True when content extends beyond the left edge.

overflow_right: bool = False

True when content extends beyond the right edge.

text_sgr: str = ''

SGR sequence applied to buffer text.

suggestion_sgr: str = ''

SGR sequence applied to suggestion text.

bg_sgr: str = ''

SGR sequence applied to fill the background.

ellipsis_sgr: str = ''

SGR sequence applied to the ellipsis indicator.

class LineEditResult(*line: str | None = None*, *eof: bool = False*, *interrupt: bool = False*, *changed: bool = False*, *bell: str = ''*)

Result of processing a keystroke.

line: str | None = None

Accepted line text when Enter was pressed, otherwise None.

eof: bool = False

True when Ctrl+D pressed on an empty line.

interrupt: bool = False

True when Ctrl+C pressed.

changed: bool = False

True when the display needs redrawing.

bell: str = ''

Bell string to emit (e.g. "\\a"), empty when silent.

class LineHistory(*max_entries: int = 5000*)

In-memory command history with navigation and prefix search.

Parameters

max_entries – Maximum number of history entries retained.

Initialize history with given maximum capacity.

entries: List[str]

History entries list (most recent last).

add(*line: str*) → None

Append *line* to history, skipping empty and consecutive duplicates.

search_prefix(*prefix: str*) → str | None

Return the most recent entry starting with *prefix*, or None.

nav_start(*current_line: str*) → None

Begin history navigation, saving *current_line*.

nav_up() → str | None

Navigate to the previous (older) history entry.

nav_down() → str | None

Navigate to the next (newer) history entry.

class LineEditor(*history:* LineHistory | *None* = *None*, *password: bool* = *False*, *password_char: str* = '', *max_width: int* = *0*, *ellipsis: str* = '…', *limit: int* = *2048*, *limit_bell: str* = '\x07', *scroll_jump: float* = *0.5*, *text_sgr: str* = '\x1b[38;2;230;225;220m', *suggestion_sgr: str* = '\x1b[30m', *bg_sgr: str* = '', *ellipsis_sgr: str* = '', *keymap: Dict[str, Callable[[...],* LineEditResult*]* | *None]* | *None* = *None*, *ambiguous_width: int* = *1*, *term_program: bool* | *str* = *False*)

Headless single-line editor with grapheme-aware cursor movement.

Feed keystrokes via *feed_key()*, read display state via *display*. Accepts blessed *Keystroke* objects directly (dispatching on `.name`), or plain strings for testing.

Custom keymap handlers must be callables accepting a single *LineEditor* argument and returning a *LineEditResult*:

```
def my_handler(editor: LineEditor) -> LineEditResult:
    ...
```

Initialize editor with optional history, display, and keymap settings.

password_char: str

max_width: int

ellipsis: str

limit: int

limit_bell: str

scroll_jump: float

text_sgr: str

suggestion_sgr: str

bg_sgr: str

ellipsis_sgr: str

ambiguous_width: int

term_program: bool | str

keymap: Dict[str, Callable[[...], *LineEditResult*] | None]

property history: *LineHistory*

Return the attached *LineHistory* instance.

property line: str

Return the current buffer contents as a string.

property password_mode: bool

Return whether password mode is currently active.

property display: *DisplayState*

Return the current *DisplayState* for rendering.

render(*term:* Terminal, *row:* *int*, *width:* *int*, *col:* *int* = *0*) → str

Build escape sequences to render the current display state.

Parameters

- **term**¶ – Blessed *Terminal* instance for cursor/SGR.
- **row**¶ – Terminal row for the input line.
- **width**¶ – Available columns.
- **col**¶ – Starting column offset (default 0).

Returns

Escape-sequence string; caller writes/encodes it.

render_insert(*term:* Terminal, *row:* *int*, *grapheme:* *str*) → str | None

Fast-path render for a single grapheme inserted at end of buffer.

Parameters

- **term**¶ – Blessed *Terminal* instance.
- **row**¶ – Terminal row for the input line.
- **grapheme**¶ – The grapheme cluster just inserted.

Returns

Escape-sequence string, or `None` if a full redraw is needed.

render_backspace(*term:* Terminal, *row:* *int*) → str | None

Fast-path render after a backspace at end of buffer.

Parameters

- **term**¶ – Blessed *Terminal* instance.
- **row**¶ – Terminal row for the input line.

Returns

Escape-sequence string, or `None` if a full redraw is needed.

feed_key(*key:* *'Keystroke'* | *str*) → *LineEditResult*

Process one keystroke and return a *LineEditResult*.

insert_text(*text:* *str*) → *LineEditResult*

Insert *text* at cursor position (for bracketed paste).

clear() → None

Clear the buffer and reset cursor to start.

set_password_mode(*enabled:* *bool*) → None

Toggle password masking on or off.

15.7 mouse.py

Sub-module providing mouse event handling.

class MouseEvent(*button_value:* *int*, *x:* *int*, *y:* *int*, *released:* *bool*, *shift:* *bool*, *meta:* *bool*, *ctrl:* *bool*, *is_motion:* *bool*, *is_wheel:* *bool*)

Mouse event with button, coordinates, and modifier information.

A unified mouse event structure that supports both legacy and SGR mouse protocols. Provides a dynamic button property that returns human-readable button names like "LEFT", "SCROLL_UP", "CTRL_LEFT", etc.

Variables

- **button_value** (*int*) – Raw button number with modifiers stripped (0=left, 1=middle, 2=right, 3=none, 64=scroll up, 65=scroll down, 66-67 for buttons 6-7, and 128-131 for buttons 8-11).
- **x** (*int*) – Horizontal position (0-indexed, in cells or pixels depending on mode).
- **y** (*int*) – Vertical position (0-indexed, in cells or pixels depending on mode).
- **released** (*bool*) – True if this is a button release event.
- **shift** (*bool*) – True if Shift modifier is pressed.
- **meta** (*bool*) – True if Meta/Alt modifier is pressed.
- **ctrl** (*bool*) – True if Ctrl modifier is pressed.
- **is_motion** (*bool*) – True if motion is being reported (drag or all- motion mode).
- **is_wheel** (*bool*) – True if this is a scroll wheel event.

Initialize a MouseEvent.

Parameters

- **button_value**¶ (*int*) – Raw button number.
- **x**¶ (*int*) – Horizontal position.
- **y**¶ (*int*) – Vertical position.
- **released**¶ (*bool*) – Whether this is a button release event.
- **shift**¶ (*bool*) – Whether Shift modifier is pressed.
- **meta**¶ (*bool*) – Whether Meta/Alt modifier is pressed.
- **ctrl**¶ (*bool*) – Whether Ctrl modifier is pressed.
- **is_motion**¶ (*bool*) – Whether motion is being reported.
- **is_wheel**¶ (*bool*) – Whether this is a scroll wheel event.

property button: str

Return human-readable button name.

Generates button names that include modifiers, button type, motion/release state:

- "LEFT", "MIDDLE", "RIGHT" for standard mouse buttons
- "LEFT_RELEASED", "MIDDLE_RELEASED", "RIGHT_RELEASED" for button releases
- "RELEASED" for a legacy release, which does not report which button was released
- "SCROLL_UP", "SCROLL_DOWN" for wheel events
- "MOTION" for mouse movement with no button pressed
- "LEFT_MOTION", "MIDDLE_MOTION", "RIGHT_MOTION" for drag events
- "CTRL_LEFT", "SHIFT_SCROLL_UP", "CTRL_SHIFT_META_MOTION" with modifiers
- "BUTTON_6" through "BUTTON_11" for extended mouse buttons

Return type
str

Returns
Button name with modifiers, button type, and motion/release state.

classmethod from_sgr_match(*match: Match[str]*) → *MouseEvent*

Parse SGR mouse event from regex match.

Handles both SGR (mode 1006) and SGR-Pixels (mode 1016) since they use identical wire formats: CSI < b;x;y m/M. The difference is semantic:

- Mode 1006: coordinates represent character cell positions
- Mode 1016: coordinates represent pixel positions

Applications must interpret x,y coordinates based on which mode was enabled.

The protocol sends 1-indexed coordinates (top-left is 1,1), but we convert to 0-indexed (top-left is 0,0) to match blessed's terminal movement functions.

Parameters
match¶ (*Match*) – Regex match object with groups 'b', 'x', 'y', 'type'.

Return type
MouseEvent

Returns
Parsed MouseEvent instance.

classmethod from_legacy_match(*match: Match[str]*) → *MouseEvent*

Parse legacy mouse event (X10/1000/1002/1003) from regex match.

The protocol sends 1-indexed coordinates (top-left is 1,1), but we convert to 0-indexed (top-left is 0,0) to match blessed's terminal movement functions.

Each coordinate is transmitted as a single byte of `value + 32`, so no position beyond 223 can be expressed. Windows Terminal and the console host (`conhost.exe`) share a VT input layer that *discards* any legacy mouse event where either coordinate exceeds 95, rather than transmit a byte above 127. Both are limitations of the terminal without workaround, the SGR protocol is the only remedy.

Parameters
match¶ (*Match*) – Regex match object with groups 'cb', 'cx', 'cy'.

Return type
MouseEvent

Returns
Parsed MouseEvent instance.

MouseSGREvent

alias of *MouseEvent*

MouseLegacyEvent

alias of *MouseEvent*

15.8 sequences.py

Module providing 'sequence awareness'.

class Sequence(*sequence_text: str*, *term:* Terminal)

A "sequence-aware" version of the base `str` class.

This unicode-derived class understands the effect of escape sequences of printable length, allowing a properly implemented *rjust()*, *ljust()*, *center()*, and *length()*.

> **Note**
>
> that other than *Sequence.padd()*, this is just a thin layer over `wcwidth`, kept by name for API compatibility.

Class constructor.

Parameters

- **sequence_text** (*str*) – A string that may contain sequences.
- **term** (*blessed.Terminal*) – *Terminal* instance.

ljust(*width: SupportsIndex*, *fillchar: str = ' '*) → str

Return string containing sequences, left-adjusted.

Parameters

- **width** (*int*) – Total width given to left-adjust `text`.
- **fillchar** (*str*) – String for padding right-of `text`.

Returns
String of `text`, left-aligned by `width`.

Return type
str

rjust(*width: SupportsIndex*, *fillchar: str = ' '*) → str

Return string containing sequences, right-adjusted.

Parameters

- **width** (*int*) – Total width given to right-adjust `text`.
- **fillchar** (*str*) – String for padding left-of `text`.

Returns
String of `text`, right-aligned by `width`.

Return type
str

center(*width: SupportsIndex*, *fillchar: str = ' '*) → str

Return string containing sequences, centered.

Parameters

- **width** (*int*) – Total width given to center `text`.
- **fillchar** (*str*) – String for padding left and right-of `text`.

Returns
String of `text`, centered by `width`.

Return type
str

clip(*start: SupportsIndex = 0*, *end: SupportsIndex = -1*, ** (Keyword-only parameters separator (PEP 3102))*, *fillchar: str = ' '*, *tabsize: int = 8*, *propagate_sgr: bool = True*, *control_codes: Literal['parse', 'strict', 'ignore'] = 'parse'*, *overtyping: bool | None = None*) → str

Return a window of this string spanning display columns `start` to `end`.

Like *clip()*, which documents the arguments. Cursor movement is interpreted by `wcwidth.clip()`, and the terminal's detected `ambiguous_width` and `term_program` are applied.

Return type
str

Returns
This string clipped to display columns (start, end)

truncate(*width: SupportsIndex*) → str

Truncate a string in a sequence-aware manner.

Any printable characters beyond `width` are removed, while all sequences remain in place. Horizontal cursor movement is interpreted by `wcwidth.clip()`.

Wide characters (such as CJK or emoji) that would partially exceed `width` are replaced with space padding to maintain exact width.

SGR (terminal styling) sequences are propagated: the result begins with any active style at the start position and ends with a reset sequence if styles were active.

Parameters
width (*int*) – The printable width to truncate the string to.

Return type
str

Returns
String truncated to exactly `width` printable characters.

length() → int

Return the printable length of string containing sequences.

Returns the maximum horizontal cursor extent reached while processing the string. Backspace and cursor-left movements do not reduce the length below the maximum position reached.

Some characters may consume more than one cell, mainly CJK (Chinese, Japanese, Korean) and Emojis and some kinds of symbols.

For example:

```
>>> from blessed import Terminal
>>> from blessed.sequences import Sequence
>>> term = Terminal()
>>> msg = term.clear + term.red('')
>>> Sequence(msg, term).length()
10
```

Note

Although accounted for, strings containing sequences such as `term.clear` will not give accurate returns, it is not considered lengthy (a length of 0).

strip(*chars: str | None = None*) → str

Return string of sequences, leading and trailing whitespace removed.

Parameters
chars (*str*) – Remove characters in chars instead of whitespace.

Return type
str

Returns
string of sequences with leading and trailing whitespace removed.

lstrip(*chars: str | None = None*) → str

Return string of all sequences and leading whitespace removed.

Parameters
chars (*str*) – Remove characters in chars instead of whitespace.

Return type
str

Returns
string of sequences with leading removed.

rstrip(*chars: str | None = None*) → str

Return string of all sequences and trailing whitespace removed.

Parameters
chars (*str*) – Remove characters in chars instead of whitespace.

Return type
str

Returns
string of sequences with trailing removed.

strip_seqs() → str

Return `text` stripped of only its terminal sequences.

Return type
str

Returns
Text with terminal sequences removed

padd(*strip: bool = False*) → str

Return non-destructive horizontal movement as destructive spacing.

Parameters
strip (*bool*) – Strip terminal sequences

Return type
str

Returns
Text adjusted for horizontal movement

iter_parse(*term: Terminal*, *text: str*) → Iterator[Tuple[str, Termcap | None]]

Generator yields (text, capability) for characters of `text`.

value for `capability` may be `None`, where `text` is `str` of length 1. Otherwise, `text` is a full matching sequence of given capability.

> **Note**
>
> Previously used by SequenceTextWrapper, sequence-aware text wrapping now exists in wcwidth as `wcwidth.wrap()`. This function is kept for API compatibility and used by *measure_length()*.

measure_length(*text: str*, *term:* Terminal) → int

Kept for API compatibility.

Return type
int

Returns
Length of the first sequence in the string

Deprecated since version 1.30.0.

15.9 terminal.py

Module containing *Terminal*, the primary API entry point.

class Terminal(*kind: str | None = None*, *stream: IO[str] | None = None*, *force_styling: bool | None = False*, *kind_fallback: str = 'xterm-256color'*, *_xtgettcap_data: TermcapResponse | None = None*)

An abstraction for color, style, positioning, and input in the terminal.

This keeps the endless calls to `tigetstr()` and `tparm()` out of your code, acts intelligently when somebody pipes your output to a non-terminal, and abstracts over the complexity of unbuffered keyboard input. It uses the terminfo database to remain portable across terminal types.

Initialize the terminal.

Parameters

- **kind** (*str*) – A terminal string used for capability database, defaults to the value of the `TERM` environment variable when undefined. If the terminal supports XTGETTCAP extension, the 'TN' capability name reported has precedence over 'kind'.
- **stream** (*file*) – A file-like object representing the Terminal output. Defaults to the original value of `sys.__stdout__`, like `curses.initscr()` does.

  If `stream` is not a tty, empty Unicode strings are returned for all capability values, so things like piping your program output to a pipe or file does not emit terminal sequences.
- **force_styling** (*bool*) – Whether to force the emission of capabilities even if `sys.__stdout__` does not seem to be connected to a terminal. If you want to force styling to not happen, use `force_styling=None`.

  This comes in handy if users are trying to pipe your output through something like `less -r` or build systems which support decoding of terminal sequences.

  When the OS Environment variable FORCE_COLOR or CLICOLOR_FORCE is *non-empty*, styling is used no matter the value specified by `force_styling`.

  Conversely, When OS Environment variable NO_COLOR is *non-empty*, styling is **not** used no matter the value specified by `force_styling`, FORCE_COLOR, or CLICOLOR_FORCE.
- **kind_fallback** (*str*) – Terminal kind to use as fallback when the requested `kind` is not found in jinxed virtual terminfo database. Defaults to `'xterm-256color'` which is well supported for all capabilities used by blessed on all modern terminals.

class CursorShape

DECSCUSR cursor shape constants accessible via Terminal.CursorShape or term.CursorShape

BLINKING_BAR = 5

BLINKING_BLOCK = 1

BLINKING_UNDERLINE = 3

COLOR_RESET_OSC = '\x1b]112\x07'

DEFAULT = 0

DEFAULT_STYLE = 2

STEADY_BAR = 6

STEADY_BLOCK = 2

STEADY_UNDERLINE = 4

STYLES: dict[str, int] = {'blinking_bar': 5, 'blinking_block': 1, 'blinking_underline': 3, 'default': 0, 'steady_bar': 6, 'steady_block': 2, 'steady_underline': 4}

static sequence(*style: int | str*) → str

Return the DECSCUSR escape sequence for the given cursor style.

Parameters

style – An integer constant (0-6) or a string name (e.g. `'blinking_bar'`).

Returns

The escape sequence string.

Raises

ValueError – If *style* is not a valid cursor shape.

class Decrqss

DECRQSS setting identifiers accessible via Terminal.Decrqss

DECSACE = '*x'

DECSASD = '$}'

DECSCA = '"q'

DECSCL = '"p'

DECSCP = '*u'

DECSCPP = '$|'

DECSCS = '*r'

DECSCUSR = ' q'

DECSDDT = '$q'

DECSDPT = '(p'

DECSFC = '*s'

DECSKCV = ' r'

DECSLCK = ' v'

DECSLPP = 't'

DECSLRM = 's'

DECSMBV = ' u'

DECSNLS = '*|'

DECSPP = '+w'

DECSPPCS = '*p'

DECSPRTT = '$s'

DECSRFR = '"t'

DECSSCLS = ' p'

DECSSDT = '$~'

DECSSL = 'p'

DECSTBM = 'r'

DECSTRL = '"u'

DECSWBV = ' t'

SGR = 'm'

class DecPrivateMode(*value: int*)

DEC Private Mode constants accessible via Terminal.DecPrivateMode or term.DecPrivateMode

AH1 = 24

ALT_MODIFIED_MOUSEWHEEL = 7765

ALT_SCREEN_AND_SAVE_CLEAR = 1049

ALT_SCREEN_BUFFER_SWITCH = 1046

ALT_SCREEN_BUFFER_XTERM = 1047

ALT_SCROLL_XTERM = 1007

ALT_SENDS_ESC = 1039

AMBIGUOUS_WIDTH_DOUBLE_WIDTH = 8840

AMBIGUOUS_WIDTH_REPORTING = 7700

APPLICATION_ESCAPE_KEY = 7727

APPLICATION_MOUSEWHEEL = 7787

BA1 = 21

```
BA2 = 22

BIDI_ARROW_KEY_SWAPPING = 1243

BIDI_AUTODETECTION = 2501

BIDI_CURRENT_LINE = 7796

BOLD_ITALIC_HIGH_INTENSITY = 1021

BRACKETED_PASTE = 2004

BRACKETED_PASTE_MIME = 5522

CHARACTER_MAPPING_SERVICE = 8800

COLOR_PALETTE_UPDATES = 2031

DEC131TM = 53

DECAAM = 100

DECANM = 2

DECARM = 8

DECARSM = 98

DECATCBM = 115

DECATCUM = 114

DECAWM = 7

DECBBSM = 116

DECBCMM = 65

DECBKM = 67

DECBPM = 55

DECBWRM = 51

DECCANSM = 101

DECCAPSLK = 109

DECCKM = 1

DECCOLM = 3

DECCRNLM = 40

DECCRTSM = 97

DECECM = 117

DECEDM = 10

DECEKEM = 16
```

```
DECESKM = 104

DECFPM = 70

DECFWM = 111

DECGEPM = 43

DECGPBM = 46

DECGPCM = 44

DECGPCS = 45

DECGRPM = 47

DECHCCM = 60

DECHDPXM = 103

DECHEBM = 35

DECHEM = 36

DECHWUM = 113

DECINLM = 9

DECIPEM = 58

DECKANAM = 12

DECKBUM = 68

DECKKDM = 59

DECKLHIM = 110

DECKPM = 81

DECLTM = 11

DECMCM = 99

DECNAKB = 57

DECNCSM = 95

DECNKM = 66

DECNRCM = 42

DECNULM = 102

DECNUMLK = 108

DECOM = 6

DECOPM = 52

DECOSCNM = 106
```

```
DECPCCM = 64

DECPEX = 19

DECPFF = 18

DECPSM = 29

DECPSP = 27

DECRLCM = 96

DECRLM = 34

DECRPL = 112

DECSCFDM = 13

DECSCLM = 4

DECSCNM = 5

DECSDM = 80

DECTCEM = 25

DECTEK = 38

DECTEM = 14

DECTHAICM = 50

DECTHAIM = 49

DECTHAISCM = 90

DECUPM = 41

DECVCCM = 61

DECVSSM = 69

DECXRLM = 73

ESC_KEY_SENDS_BACKSLASH = 7728

EXTENDED_REVERSE_WRAPAROUND = 1045

FAST_SCROLL = 1014

FOCUS_IN_OUT_EVENTS = 1004

FONT_CHANGE_REPORTING = 7767

GRAPHEME_CLUSTERING = 2027

GRAPHICS_POSITION = 7730

GRAPHICS_POSITION_2 = 7780

HP_FUNC_KEY_MODE = 1052
```

```
INPUT_METHOD_EDITOR = 737769

IN_BAND_WINDOW_RESIZE = 2048

ITERM2_REPORT_KEY_UP = 1337

KEEP_SELECTION_NO_HILITE = 1040

KITTY_HANDLE_CTRL_C_Z = 19997

KP_DELETE_SENDS_DEL = 1037

LEGACY_KBD_X11R6 = 1060

META_SENDS_ESC = 1036

META_SETS_EIGHTH_BIT = 1034

MINTTY_BIDI = 77096

MIRROR_BOX_DRAWING = 2500

MODIFIERS_ALT_NUMLOCK = 1035

MOUSEWHEEL_REPORTING = 7786

MOUSE_ALL_MOTION = 1003

MOUSE_EXTENDED_SGR = 1006

MOUSE_EXTENDED_UTF8 = 1005

MOUSE_HILITE_TRACKING = 1001

MOUSE_REPORT_CLICK = 1000

MOUSE_REPORT_DRAG = 1002

MOUSE_SGR_PIXELS = 1016

MOUSE_URXVT = 1015

OV1 = 20

PASSIVE_MOUSE_TRACKING = 2029

PK1 = 23

RAISE_ON_CTRL_G = 1043

READLINE_CHARACTER_QUOTING = 2005

READLINE_MOUSE_BUTTON_1 = 2001

READLINE_MOUSE_BUTTON_2 = 2002

READLINE_MOUSE_BUTTON_3 = 2003

READLINE_NEWLINE_PASTING = 2006

REPLACEMENT_CHAR_COLOR = 85
```

REPORT_GRID_CELL_SELECTION = 2030

REUSE_CLIPBOARD_DATA = 1044

REWRAP_ON_RESIZE_MINTTY = 7723

SAVE_CURSOR_DECSC = 1048

SCO_FUNC_KEY_MODE = 1053

SCROLL_MARKERS = 7711

SCROLL_ON_KEYPRESS_RXVT = 1011

SCROLL_ON_TTY_OUTPUT_RXVT = 1010

SHORTCUT_KEY_MODE = 7783

SHOW_HIDE_SCROLLBAR = 7766

SHOW_SCROLLBAR_RXVT = 30

SIXEL_PRIVATE_PALETTE = 1070

SIXEL_SCROLLING_LEAVES_CURSOR = 8452

SUN_FUNC_KEY_MODE = 1051

SYNCHRONIZED_OUTPUT = 2026

SpecialInternalKitty = -1

SpecialInternalLegacyCSIModifier = -3

SpecialInternalModifyOtherKeys = -2

TERMINFO_FUNC_KEY_MODE = 1050

TEXT_REFLOW = 2028

TTCTH = 8200

URGENCY_ON_CTRL_G = 1042

USE_CLIPBOARD_SELECTION = 1041

VT220_KBD_EMULATION = 1061

WIN32_INPUT_MODE = 9001

WYENAT_OFF = 84

WY_52_LINE = 83

property long_description: str

Full description of this DEC private mode's functionality.

Return type

str

Returns

Detailed description or "Unknown mode" for unrecognized modes

ambiguous_width

Width of East Asian ambiguous characters, always 1, may be 2 by env value, `AMBIGUOUS_WIDE` or until detected by call to *detect_ambiguous_width()*.

term_program: `str`

Normalized terminal software name for python wcwidth correction tables.

__getattr__(*attr: str*) → *NullCallableString* | *ParameterizingString* | *FormattingString*

Return a terminal capability as Unicode string.

For example, `term.bold` is a unicode string that may be prepended to text to set the video attribute for bold, which should also be terminated with the pairing *normal*. This capability returns a callable, so you can use `term.bold("hi")` which results in the joining of `(term.bold, "hi", term.normal)`.

Compound formatters may also be used. For example:

```
>>> term.bold_blink_red_on_green("merry x-mas!")
```

For a parameterized capability such as `move` (or `cup`), pass the parameters as positional arguments:

```
>>> term.move(line, column)
```

See the manual page terminfo(5) for a complete list of capabilities and their arguments.

property kind: `str`

Read-only property: Terminal kind determined on class initialization.

Return type
str

property does_styling: `bool`

Read-only property: Whether this class instance may emit sequences.

Return type
bool

property is_a_tty: `bool`

Read-only property: Whether *stream* is a terminal.

Return type
bool

property height: `int`

Read-only property: Height of the terminal (in number of lines).

Return type
int

property width: `int`

Read-only property: Width of the terminal (in number of columns).

Return type
int

property pixel_height: `int`

Read-only property: Height of the terminal (in pixels).

Return type
int

property pixel_width: int

Read-only property: Width of terminal (in pixels).

Return type
int

location(*x: int | None = None*, *y: int | None = None*) → Generator[None, None, None]

Context manager for temporarily moving the cursor.

Parameters

- **x** (*int*) – horizontal position, from left, *0*, to right edge of screen, *self.width - 1*.
- **y** (*int*) – vertical position, from top, *0*, to bottom of screen, *self.height - 1*.

Returns
a context manager.

Return type
Iterator

Move the cursor to a certain position on entry, do any kind of I/O, and upon exit let you print stuff there, then return the cursor to its original position:

```
term = Terminal()
with term.location(y=0, x=0):
    for row_num in range(term.height-1):
        print('Row #{row_num}')
print(term.clear_eol + 'Back to original location.')
```

Specify **x** to move to a certain column, **y** to move to a certain row, both, or neither. If you specify neither, only the saving and restoration of cursor position will happen. This can be useful if you simply want to restore your place after doing some manual cursor movement.

Calls cannot be nested: only one should be entered at a time.

Note

The argument order *(x, y)* differs from the return value order *(y, x)* of *get_location()*, or argument order *(y, x)* of `move()`. This is for API Compatibility with the blessings library, sorry for the trouble!

get_location(*timeout: float = 5.0*) → Tuple[int, int]

Return tuple (row, column) of cursor position.

When *is_a_tty* is False, no sequences are transmitted or response awaited, and the value `(-1, -1)` is returned without inquiry.

Parameters
timeout (*float*) – Return after time elapsed in seconds with value `(-1, -1)` indicating that the remote end did not respond.

Return type
tuple

Returns
cursor position as tuple in form of `(y, x)`. Always ensure the return value is checked for `(-1, -1)`.

The location of the cursor is determined by emitting the `u7` terminal capability for the VT100 Query Cursor Position. This elicits a response from a reply string described by capability `u6` for the VT100 response.

The `(y, x)` return value matches the parameter order of the *move_yx()* capability. The following sequence should cause the cursor to not move at all:

```
>>> term = Terminal()
>>> term.move_yx(*term.get_location())
```

And the following should assert True with a terminal:

```
>>> term = Terminal()
>>> given_y, given_x = 10, 20
>>> with term.location(y=given_y, x=given_x):
...     result_y, result_x = term.get_location()
...
>>> assert given_x == result_x, (given_x, result_x)
>>> assert given_y == result_y, (given_y, result_y)
```

get_fgcolor(*timeout: float = 5.0*, *bits: int = 16*) → Tuple[int, int, int]

Return tuple (r, g, b) of default foreground color.

This returns the terminal's default color for uncolored text, not any currently active color set by escape sequences.

When *is_a_tty* is False, no sequences are transmitted or response awaited, and the value `(-1, -1, -1)` is returned without inquiry.

Parameters

- **timeout** (*float*) – Return after time elapsed in seconds with value `(-1, -1, -1)` indicating that the remote end did not respond.
- **bits** (*int*) – Bits per channel: 16 (default, 0-65535, 48-bit total) or 8 (0-255, 24-bit total).

Return type
tuple

Returns
foreground color as tuple in form of `(r, g, b)`. When a timeout is specified, always ensure the return value is checked for `(-1, -1, -1)`.

Raises
ValueError – When bits is not 8 or 16.

The foreground color is determined by emitting an OSC 10 color query.

See also *get_bgcolor()* for querying the background color.

Note

The default return value is 16-bit, matching the XParseColor specification of the underlying protocol. For most uses, 8-bit values are preferred:

```
rgb = term.get_fgcolor(bits=8)
term.color_rgb(*rgb)  # reset foreground to default
```

get_bgcolor(*timeout: float = 5.0*, *bits: int = 16*) → Tuple[int, int, int]

Return tuple (r, g, b) of default background color.

This returns the terminal's default background color, not any currently active color set by escape sequences.

When *is_a_tty* is False, no sequences are transmitted or response awaited, and the value `(-1, -1, -1)` is returned without inquiry.

Parameters

- **timeout** (*float*) – Return after time elapsed in seconds with value `(-1, -1, -1)` indicating that the remote end did not respond.
- **bits** (*int*) – Bits per channel: 16 (default, 0-65535, 48-bit total) or 8 (0-255, 24-bit total).

Return type
tuple

Returns
background color as tuple in form of `(r, g, b)`. When a timeout is specified, always ensure the return value is checked for `(-1, -1, -1)`.

Raises
ValueError – If bits is not 8 or 16.

The background color is determined by emitting an OSC 11 color query.

See also *get_fgcolor()* for querying the foreground color.

Note

The default return value is 16-bit, matching the XParseColor specification of the underlying protocol. For most uses, 8-bit values are preferred:

```
rgb = term.get_bgcolor(bits=8)
term.on_color_rgb(*rgb)  # reset background to default
```

get_fgcolor_hex(*timeout: float = 5.0*, *maybe_short: bool = False*) → str

Return default foreground color as hex string.

Parameters

- **timeout** (*float*) – Return after time elapsed in seconds with empty string if the remote end did not respond.
- **maybe_short** (*bool*) – If True, return `#RGB` when possible.

Returns
Hex color string (`#RRGGBB` or `#RGB`), or empty string on timeout or when *is_a_tty* is False.

Convenience wrapper combining *get_fgcolor()* and *rgb_to_hex()*.

get_bgcolor_hex(*timeout: float = 5.0*, *maybe_short: bool = False*) → str

Return default background color as hex string.

Parameters

- **timeout** (*float*) – Return after time elapsed in seconds with empty string if the remote end did not respond.
- **maybe_short** (*bool*) – If True, return `#RGB` when possible.

Returns
Hex color string (`#RRGGBB` or `#RGB`), or empty string on timeout or when *`is_a_tty`* is False.

Convenience wrapper combining *`get_bgcolor()`* and *`rgb_to_hex()`*.

`get_device_attributes`(*timeout: float | None = 5.0, force: bool = False*) → *DeviceAttribute* | None

Query the terminal's Device Attributes (DA1).

When *`is_a_tty`* is False, no sequences are transmitted or response awaited, and `None` is returned without inquiry.

If a Device Attributes query fails to respond within the `timeout` specified, `None` is returned. If this was the first query for device attributes, all subsequent queries return `None` unless `force=True` is set (sticky failure).

Responses are cached indefinitely unless `force=True` is specified.

> **Note**
>
> A `timeout` value should be set to avoid blocking when the terminal does not respond to DA1 queries, which may happen with some kinds of "dumb" terminals.

Parameters

- **`timeout`** (*`float`*) – Timeout in seconds to await terminal response
- **`force`** (*`bool`*) – Force active terminal inquiry even if cached result exists or previous query failed

Return type
DeviceAttribute or None

Returns
DeviceAttribute instance with terminal capabilities, or None if unsupported/timeout

```
term = Terminal()

# Query device attributes
da = term.get_device_attributes(timeout=TERMINAL_QUERY_TIMEOUT_SECONDS)
if da is not None:
    print(f"Service class: {da.service_class}")
    print(f"Supports sixel: {da.supports_sixel}")
    print(f"Extensions: {sorted(da.extensions)}")
```

`get_software_version`(*timeout: float | None = 5.0, force: bool = False*) → *SoftwareVersion* | None

Return the terminal's software name and version.

A non-empty `TERM_PROGRAM` value takes precedence and no inquiry is made, otherwise an XTVERSION query is sent to the terminal and the response is returned as a `SoftwareVersion` instance. Returns `None` only if both methods fail.

Only Successful responses are cached indefinitely unless `force=True` is specified. Unlike other query methods, there is no "sticky failure", failed queries are not cached and are retried.

> **Note**
>
> A `timeout` value should be set to avoid blocking when the terminal does not respond to XTVERSION queries, which may happen with some kinds of "dumb" terminals.

Parameters

- **timeout** (*float*) – Timeout in seconds to await terminal response
- **force** (*bool*) – Force active terminal inquiry even if cached result exists

Return type
SoftwareVersion or None

Returns
SoftwareVersion instance with terminal name and version, or None if unsupported/timeout

```
term = Terminal()

# Query software version
sv = term.get_software_version(timeout=TERMINAL_QUERY_TIMEOUT_SECONDS)
if sv is not None:
    print(f"Terminal: {sv.name} {sv.version}")
```

does_sixel(*timeout: float | None = 5.0*, *force: bool = False*) → bool

Query whether the terminal supports sixel graphics.

Sixel is a bitmap graphics format supported by some modern terminal emulators, allowing applications to display inline images.

When *is_a_tty* is False, no sequences are transmitted or response awaited, and `False` is returned without inquiry.

This method calls *get_device_attributes()* to query the terminal's capabilities and checks for sixel support (extension 4 in the DA1 response). Results are cached, so subsequent calls are fast.

Parameters

- **timeout** (*float*) – Timeout in seconds to await terminal response. When `None` (default), the query may block indefinitely.
- **force** (*bool*) – Bypass cache and re-query the terminal

Return type
bool

Returns
`True` if terminal supports sixel graphics, `False` otherwise

detect_ambiguous_width(*timeout: float = 5.0*, *fallback: int = 1*, *force: bool = False*) → int

Detect whether the terminal renders ambiguous width characters as width 1 or 2.

Parameters

- **timeout** (*float*) – Timeout in seconds for any single cursor position response.
- **fallback** (*int*) – Value to return on timeout, invalid measurement, or not a tty.
- **force** (*bool*) – Measure again, bypassing the cache and the `AMBIGUOUS_WIDE` override.

Return type
int

Returns
1 for "Ambiguous width as narrow", 2 is "Ambiguous width as wide"

East Asian ambiguous width characters render as either single (1) or double width (2) depending on terminal settings. This measures the rendered width of U+00A7 (SECTION SIGN), an early Unicode character with East Asian Width property "Ambiguous", by drawing and erasing it at the current cursor position. Ambiguous width is used in "legacy" contexts, like with BIG5, gbk, or shift-jis encoding, and most especially in mathematics or artwork, like on terminal bulletin board systems, https://bbs.modem.xyz/encodings.html#big5, and sometimes simply as a user preference, for neat and tidy vertical alignment. configuration is usually required to enable "ambiguous width as wide".

The result is cached, assigned to *ambiguous_width*, and returned, and used automatically by subsequent alignment methods, *length()*, *wrap()*, *ljust()*, *rjust()*, and *center()*.

Example usage:

```
>>> term = Terminal()
>>> term.length('\u00a7')
1
>>> term.detect_ambiguous_width()
2
>>> term.length('\u00a7')
2
```

When *is_a_tty* is False, no sequences are transmitted or response awaited, and `fallback` is returned without inquiry.

detect_term_program(*timeout: float = 5.0, force: bool = False*) → str

Detect the terminal software name applied to python wcwidth correction tables.

Parameters

- **timeout** (*float*) – Timeout in seconds to await an XTVERSION response.
- **force** (*bool*) – Detect again, bypassing the cache.

Return type
str

Returns
Terminal name, wcwidth alias, or an empty string when unknown.

A non-empty `TERM_PROGRAM` environment value or a unique `TERM` that matches wcwidth other than 'xterm' returns that value. Otherwise, an in-band XTVERSION query is made and its software name is returned if responded.

The result is cached, assigned to *term_program*, and used automatically by width or alignment methods, *length()*, *wrap()*, *ljust()*, *rjust()*, and *center()*.

When *is_a_tty* or *does_styling* is False, or no keyboard is available, XTVERSION inquiries are not made and timeout does not apply, and the result is not cached.

get_dec_mode(*mode: int | DecPrivateMode, timeout: float = 5.0, force: bool = False*) → *DecModeResponse*

Query the state of a DEC Private Mode (DECRQM).

Sends a DECRQM query to the terminal and returns a `DecModeResponse` instance. Use the helper methods like `DecModeResponse.is_supported()` or `DecModeResponse.is_enabled()` and others to interpret the result.

When *is_a_tty* is False, no sequences are transmitted or response awaited, and the `DecModeResponse` value returned is always `DecModeResponse.NOT_QUERIED`.

In some cases a `timeout` value should be set, as it is possible for a terminal that succeeds *is_a_tty* to fail to respond to DEC Private Modes, such as in a CI Build Service or other "dumb" terminal, even a few popular modern ones such as Konsole. The first query may also make an XTVERSION query, *get_software_version()*, to avoid a DECRQM query to terminals with leaky output (Terminal.app), so the total elapsed time may exceed *timeout*.

If a DEC Private mode query fails to respond within the `timeout` specified, the `DecModeResponse` value returned is `DecModeResponse.NO_RESPONSE`. If this was the first DEC Private mode query, all subsequent queries return a `DecModeResponse` value of `DecModeResponse.NOT_QUERIED` unless `force=True` is set.

Repeat queries return the (cached) known state immediately without re-inquiry unless `force=True`. Although there are special cases where a user may re-configure their terminal settings after the state was requested by an application, the application is generally restarted to recognize the new settings rather than to repeatedly re-inquire about their latest value!

Parameters

- **mode** – DEC Private Mode to query
- **timeout** (*float*) – Timeout in seconds to await terminal response
- **force** (*bool*) – Force active terminal inquery in all cases

Return type
DecModeResponse

Returns
DecModeResponse instance

Raises
TypeError – If mode is not DecPrivateMode or int

```
term = Terminal()

# Query synchronized output support
response = term.get_dec_mode(DecPrivateMode.SYNCHRONIZED_OUTPUT)
if response.supported:
    print("Synchronized output is available")
```

dec_modes_enabled(**modes: int | DecPrivateMode, timeout: float | None = 5.0*) → Generator[None, None, None]

Context manager for temporarily enabling DEC Private Modes.

On entry, queries each mode's current state using get_dec_mode().

For modes that are supported but currently disabled, enables them and tracks them for restoration. On exit, disables all modes that were enabled by this context manager, restoring original state.

Unsupported modes are silently ignored.

Parameters

- **modes** – One or more DEC Private Mode numbers or enum members
- **timeout** (*float*) – Timeout in seconds for get_dec_mode calls

Raises
TypeError – If mode is not DecPrivateMode or int

```
term = Terminal()

# Enable synchronized output temporarily
with term.dec_modes_enabled(DecPrivateMode.SYNCHRONIZED_OUTPUT):
    # All output will be atomic
    print("Frame 1")
    print("Frame 2")
```

dec_modes_disabled(**modes: int | DecPrivateMode, timeout: float | None = 5.0*) → Generator[None, None, None]

Context manager for temporarily disabling DEC Private Modes.

Uses the same logic as dec_modes_enabled but inverted: disables supported modes that are currently enabled on entry, then restores them on exit.

Parameters

- **modes** – One or more DEC Private Mode numbers or enum members
- **timeout** (*float*) – Timeout in seconds for get_dec_mode calls

Raises

TypeError – If mode is not DecPrivateMode or int

does_mouse(**, clicks: bool = True, report_pixels: bool = False, report_drag: bool = False, report_motion: bool = False, timeout: float = 5.0*) → bool

Check if the terminal supports the specified mouse tracking features.

This method queries terminal support for the same DEC modes that *mouse_enabled()* would enable with the given parameters.

Parameters

- **clicks** (*bool*) – Check for basic click reporting (default True)
- **report_pixels** (*bool*) – Check for pixel coordinate reporting
- **report_drag** (*bool*) – Check for drag reporting
- **report_motion** (*bool*) – Check for motion reporting
- **timeout** (*float*) – Timeout for mode queries (default 1.0s)

Returns

True if all required modes are supported

Return type

bool

Example:

```
if term.does_mouse(report_drag=True, report_pixels=True):
    with term.mouse_enabled(report_drag=True, report_pixels=True):
        # Use mouse tracking with drag and pixel support
        pass
```

does_inband_resize(*timeout: float = 5.0*) → bool

Check if the terminal supports in-band window resize notifications.

This method queries whether the terminal supports DEC mode 2048 (IN_BAND_WINDOW_RESIZE), which allows receiving resize events as in-band sequences through *inkey()* instead of relying on SIGWINCH signals.

Parameters
timeout (*float*) – Timeout for mode query (default 1.0s)

Returns
True if in-band resize notifications are supported

Return type
bool

Example:

```
if term.does_inband_resize():
    with term.notify_on_resize():
        # Use in-band resize events
        pass
else:
    # Fall back to SIGWINCH or other methods
    pass
```

does_bracketed_paste(*timeout: float = 5.0*) → bool

Check if the terminal supports bracketed paste mode.

Bracketed paste mode (DEC mode 2004) wraps pasted text in special sequences so applications can distinguish pasted text from typed input.

See also
https://invisible-island.net/xterm/ctlseqs/ctlseqs.html#h3-Bracketed-Paste-Mode

Parameters
timeout (*float*) – Timeout for mode query (default 1.0s).

Returns
True if bracketed paste mode is supported.

Return type
bool

does_synchronized_output(*timeout: float = 5.0*) → bool

Check if the terminal supports synchronized output.

Synchronized output (DEC mode 2026) allows applications to batch screen updates, preventing tearing during rapid redraws.

See also
https://gist.github.com/christianparpart/d8a62cc1ab659194571ec44c8e0a7085

Parameters
timeout (*float*) – Timeout for mode query (default 1.0s).

Returns
True if synchronized output is supported.

Return type
bool

does_grapheme_clustering(*timeout: float = 5.0*) → bool

Check if the terminal supports grapheme clustering.

Grapheme clustering (DEC mode 2027) enables Unicode grapheme cluster aware cursor movement and display.

See also

https://mitchellh.com/writing/grapheme-clusters-in-terminals

Parameters
timeout (*float*) – Timeout for mode query (default 1.0s).

Returns
True if grapheme clustering is supported.

Return type
bool

does_focus_events(*timeout: float = 5.0*) → bool

Check if the terminal supports focus in/out event reporting.

Focus event reporting (DEC mode 1004) sends escape sequences when the terminal window gains or loses focus.

See also

https://invisible-island.net/xterm/ctlseqs/ctlseqs.html#h3-FocusIn_FocusOut

Parameters
timeout (*float*) – Timeout for mode query (default 1.0s).

Returns
True if focus event reporting is supported.

Return type
bool

get_xtgettcap(*timeout: float | None = 5.0*, *force: bool = False*, *caps: Iterable[str] | None = None*) → TermcapResponse | None

Query terminal capabilities via XTGETTCAP (DCS +q).

When *caps* is specified, only those capabilities are returned. When unspecified, all known XTGETTCAP capabilities are queried and returned. All results from XTGETTCAP are memoized permanently, unlike DECRQSS *get_decrqss()* which may change. Using `force=True` to re-retrieve any request.

Parameters

- **timeout** (*float*) – Timeout in seconds.

- **force** (*bool*) – Always re-query for latest values.
- **caps** – Capability names to query. When specified, only these additional capabilities may queried (incremental on cache). when unspecified, it as though *all=True* was used.

Return type
TermcapResponse

does_xtgettcap(*timeout: float | None = 5.0*, *force: bool = False*) → bool

Whether the terminal supports XTGETTCAP (DCS +q) queries.

Support is determined at class initialization and so this call is always non-blocking unless the *force* argument is used.

Parameters

- **timeout** (*float*) – Timeout in seconds.
- **force** (*bool*) – Bypass cached result.

Return type
bool

get_font_coverage(*text: str*, *timeout: float | None = 5.0*, *force: bool = False*) → FontCoverage

Query which codepoints of *text* the terminal font has a glyph for.

Parameters

- **text** (*str*) – Text whose codepoints to query. Each distinct codepoint is asked about once, in batches, however long *text* is and however often it repeats.
- **timeout** (*float*) – Seconds to wait for each round-trip. A long *text* takes several, so this bounds a single exchange and not the call.
- **force** (*bool*) – Discard the cached protocol probe and every cached codepoint result, and query the terminal again.

Return type
FontCoverage

Returns
Per-codepoint coverage. When *text* is empty, or when neither protocol answers, an empty result whose `supported` is False, which is falsey. A codepoint the terminal declined to answer for is reported through `unknown` and is not asked about again until `force`.

Two protocols are tried, mintty's *Font Glyph Coverage Enquiry* (`OSC 7771`) first, then the *Glyph Protocol* `'q'` verb (`APC 25a1`) of rio. The protocol that answers is remembered, as is the result for every codepoint queried, so repeated calls over overlapping text cost no further round-trips.

```
>>> coverage = term.get_font_coverage('—')
>>> if not coverage:
...     print('terminal cannot say')
... elif not coverage.covers('—'):
...     print('falling back to ASCII box drawing')
```

Warning

This reports *coverage of codepoints by a font*, and nothing more. Neither protocol can be asked about a grapheme cluster, and neither reports on the shaping that turns codepoints into drawn glyphs: ligatures, mark positioning, `ZWJ` joining, `VS-15`/`VS-16` presentation, contextual forms, or fallback to another font.

A cluster whose codepoints are all covered may still fail to draw, and one with uncovered codepoints may draw perfectly. The text presentation watch, `'⌚\uFE0E'`, renders correctly on terminals whose font has no glyph for `U+FE0E`, as no font does. Deciding what will render is a heuristic built on this answer, and blessed does not make it for you: see *detect-tofus.py* for one such heuristic.

See also

https://github.com/mintty/mintty/wiki/CtrlSeqs#font-glyph-coverage-enquiry and https://rapha.land/introducing-glyph-protocol-for-terminals/

does_kitty_graphics(*timeout: float | None = 5.0*, *force: bool = False*) → bool

Check if the terminal supports the Kitty graphics protocol.

Sends a minimal Kitty graphics query and checks for an `OK` response. Supported by kitty and WezTerm.

See also

https://sw.kovidgoyal.net/kitty/graphics-protocol/

Parameters

- **timeout** (*float*) – Timeout in seconds.
- **force** (*bool*) – Bypass cached result.

Return type
bool

get_iterm2_capabilities(*timeout: float | None = 5.0*, *force: bool = False*) → ITerm2Capabilities | None

Query iTerm2 capabilities via OSC 1337;Capabilities.

When *is_a_tty* is False, returns `None`.

Responses are cached unless *force* is True.

See also

https://iterm2.com/documentation-escape-codes.html

Parameters

- **timeout** (*float*) – Timeout in seconds for each sub-query. On the first call, an additional CPR probe may be performed to enable fast negative detection, so total elapsed time may exceed *timeout* for terminals that do not respond.
- **force** (*bool*) – Bypass cached result.

Return type
ITerm2Capabilities or None

does_iterm2(*timeout: float | None = 5.0*, *force: bool = False*) → bool

Check if the terminal answers the `OSC 1337` capabilities report.

Parameters

- **timeout** (*float*) – Timeout in seconds.
- **force** (*bool*) – Bypass cached result.

Return type
bool

does_iterm2_graphics(*timeout: float | None = 5.0*, *force: bool = False*) → bool

Check if the terminal supports the iTerm2 inline image protocol.

Draws a transparent image using iTerm2 protocol and detects for expected cursor position advance. Uses two CPR requests, before and after, for fast negative detection.

See also

https://iterm2.com/documentation-images.html

Parameters

- **timeout** (*float*) – Timeout in seconds for each cursor position query.
- **force** (*bool*) – Bypass cached result.

Return type
bool

Returns
True when the cursor advanced a single column, False when the terminal is 1-column width or did not answer a cursor position report within timeout.

does_kitty_notifications(*timeout: float | None = 5.0*, *force: bool = False*) → bool

Check if the terminal supports Kitty desktop notifications (OSC 99).

Sends an OSC 99 query with a CPR boundary guard for fast negative detection. Currently supported only by kitty.

See also

https://sw.kovidgoyal.net/kitty/desktop-notifications/

Parameters

- **timeout** (*float*) – Timeout in seconds for each sub-query. On the first call, an additional CPR probe may be performed to enable fast negative detection, so total elapsed time may exceed *timeout* for terminals that do not respond.
- **force** (*bool*) – Bypass cached result.

Return type
bool

does_kitty_clipboard(*timeout: float | None = 5.0*, *force: bool = False*) → bool

Check if the terminal supports the Kitty clipboard protocol (mode 5522).

Queries DEC private mode 5522 (Bracketed Paste MIME) by *get_dec_mode()*.

Parameters

- **timeout** (*float*) – Timeout in seconds.
- **force** (*bool*) – Bypass cached result.

Return type
bool

does_kitty_pointer_shapes(*timeout: float | None = 5.0*, *force: bool = False*) → str | None

Query Kitty mouse pointer shape support (OSC 22).

Returns the current pointer shape name if supported, or `None` if the terminal does not respond. Uses a CPR boundary guard for fast negative detection.

See also

https://sw.kovidgoyal.net/kitty/pointer-shapes/

Parameters

- **timeout** (*float*) – Timeout in seconds.
- **force** (*bool*) – Bypass cached result.

Return type
str or None

does_osc52_clipboard(*timeout: float | None = 5.0*, *force: bool = False*) → bool

Detect OSC 52 clipboard support without reading the clipboard.

This method uses non-intrusive detection to avoid triggering user-facing clipboard permission prompts, **DA1 extension 52** and **XTGETTCAP ``Ms``** fields. These methods are preferred over sending an actual OSC 52 read request (`\x1b]52;c;?\a`), which may trigger a clipboard permission dialog in many modern terminals.

Parameters

- **timeout** (*float*) – Timeout in seconds for each sub-query.
- **force** (*bool*) – Bypass cached result.

Return type
bool

clipboard_copy(*text: str*, *selection: str = 'c'*) → None

Copy text to the system clipboard via OSC 52.

Writes an OSC 52 set sequence to the terminal output stream. Most modern terminals accept clipboard writes without any user prompt.

The sequence is written unconditionally when *does_styling* is `True`. If the terminal does not support OSC 52, the sequence should be ignored by the terminal.

Parameters

- **text** (*str*) – The text to copy to the clipboard.
- **selection** (*str*) – The X11 selection target – `'c'` for clipboard (default), `'p'` for primary selection, or `'s'` for secondary. Most terminals only honor `'c'`.

clipboard_paste(*timeout: float | None = 10*, *selection: str = 'c'*) → str | None

Read the system clipboard via OSC 52.

Sends an OSC 52 query (`\x1b]52;c;?\a`) and waits for the terminal to respond with the clipboard contents.

> **Warning**
>
> Many modern terminals display a permission dialog when an application reads the clipboard. The user must approve the dialog before the terminal sends a response.
>
> The default timeout of 10 seconds allows time for human interaction. If the user denies the dialog, or no response is received within given timeout, this method returns `None`.

Parameters

- **timeout** (*float*) – Timeout in seconds. A generous timeout is recommended because the user may need to interact with a permission dialog.
- **selection** (*str*) – The X11 selection target – `'c'` for clipboard (default), `'p'` for primary.

Return type
str or None

Returns
The clipboard text, or `None` if denied or timeout is reached.

get_color_scheme(*timeout: float | None = 5.0*, *force: bool = False*) → str | None

Query the terminal's color scheme preference (dark or light mode).

Sends a `CSI ? 996 n` Device Status Report query. Terminals that support color-scheme reporting respond with `CSI ? 997 ; Ps n` where *Ps* is `1` for dark mode or `2` for light mode. Uses a CPR boundary guard for fast negative detection.

This relates to Mode 2031 (`COLOR_PALETTE_UPDATES`) (not implemented).

> **See also**
>
> Color palette update notifications

Parameters

- **timeout** (*float*) – Timeout in seconds.
- **force** (*bool*) – Bypass cached result.

Return type
str or None

Returns
`'dark'`, `'light'`, or `None` if unsupported.

does_kitty_query(*timeout: float | None = 5.0*, *force: bool = False*) → bool

Detect Kitty with XTGETTCAP query extensions.

This is a trifle convenience wrapper of *get_xtgettcap()*.

This method probes for `kitty-query-name` support. A successful response indicates that some set of Kitty query extensions are available.

To retrieve their values, follow up with *get_xtgettcap()* directly, for example:

```
>>> term = Terminal()
>>> term.does_kitty_query()
True
>>> term.get_xtgettcap(caps=[f"kitty-query-{key}" for key in [
    "font_family", "font_size", "dpi_x", "dpi_y"]])
TermcapResponse(supported=True, capabilities={
    'kitty-query-font_family': 'NotoSansMono-Regular',
    'kitty-query-font_size': '19',
    'kitty-query-dpi_x': '96',
    'kitty-query-dpi_y': '96'})
```

Parameters

- **timeout** (*float*) – Timeout in seconds.
- **force** (*bool*) – Bypass cached result.

Return type
bool

get_decrqss(*setting_id: str = 'm'*, *timeout: float | None = 5.0*) → str | None

Query terminal state via DECRQSS (Request Status String).

Sends `DCS $ q <setting_id> ST` and decodes the response. Returns the parameter value string on success, with the echoed setting identifier stripped. Returns `None` when the terminal does not support DECRQSS or the setting is invalid.

Results are not cached – DECRQSS queries runtime state that may change between calls (cursor style, margins, SGR, etc.).

Use `Decrqss` for setting identifiers:

```
term = Terminal()
term.get_decrqss(term.Decrqss.DECSCUSR)  # cursor style
term.get_decrqss(term.Decrqss.DECSTBM)   # scroll region
```

See also

DECRQSS specification

The first query may also make an XTVERSION query, *get_software_version()*, to avoid a DECRQSS query to terminals with leaky output (Terminal.app).

Parameters

- **setting_id** (*str*) – Setting identifier to query (default: SGR).
- **timeout** (*float*) – Timeout in seconds.

Return type
str or None

does_decrqss(*timeout: float | None = 5.0*, *force: bool = False*) → bool

Detect DECRQSS (Request Status String) support.

Sends a DECRQSS query for the current SGR state. A valid response indicates the terminal supports status string queries.

See also

DECRQSS specification

Parameters

- **timeout** (*float*) – Timeout in seconds.
- **force** (*bool*) – Bypass cached result.

Return type
bool

does_styled_underlines(*timeout: float | None = 5.0, force: bool = False*) → bool

Detect extended underline style support (curly, dotted, dashed).

Queries the terminal's Smulx terminfo capability via XTGETTCAP. When supported, the terminal can render underline styles beyond the standard single underline, such as CSI 4:3 m (curly).

Parameters

- **timeout** (*float*) – Timeout in seconds for XTGETTCAP query.
- **force** (*bool*) – Bypass cached result.

Return type
bool

does_colored_underlines(*timeout: float | None = 5.0, force: bool = False*) → bool

Detect colored underline support (CSI 58;2;r;g;b m).

Queries the terminal's Setulc terminfo capability via XTGETTCAP.

Parameters

- **timeout** (*float*) – Timeout in seconds for XTGETTCAP query.
- **force** (*bool*) – Bypass cached result.

Return type
bool

does_text_sizing(*timeout: float = 5.0, force: bool = False*) → TextSizingResult

Detect Kitty text sizing protocol support (OSC 66).

Tests width and scale text sizing by sending OSC 66 probes and measuring cursor position delta. Supported terminals may write up to 2 destructive spaces at the current cursor position, while unsupported terminals typically produce no output.

Responses are cached unless *force* is True.

Parameters

- **timeout** (*float*) – Timeout in seconds for each cursor position query.
- **force** (*bool*) – Bypass cached result.

Return type
TextSizingResult

Returns
Result with `.width` and `.scale` boolean attributes.

mouse_enabled(*, *clicks: bool = True*, *report_pixels: bool = False*, *report_drag: bool = False*, *report_motion: bool = False*, *timeout: float = 5.0*) → Generator[None, None, None]

Context manager for enabling mouse tracking with various reporting modes.

Enables mouse event reporting with sensible defaults, and always enables SGR extended mouse mode (1006).

Parameters

- **clicks** (*bool*) – Enable basic click reporting (default True)
- **report_pixels** (*bool*) – Report pixel coordinates instead of cell coordinates
- **report_drag** (*bool*) – Report mouse drag events (button held while moving)
- **report_motion** (*bool*) – Report all mouse motion events
- **timeout** (*float*) – Timeout for mode queries (default 1.0s)

The reporting modes have precedence: motion > drag > clicks. Enabling a higher-precedence mode automatically includes lower modes.

Example:

```
with term.mouse_enabled():
    # Basic click tracking
    inp = term.inkey()
    if inp.name and inp.name.startswith('MOUSE_'):
        print(f"Clicked at {inp.x}, {inp.y}")

with term.mouse_enabled(report_drag=True):
    # Track clicks and drags
    pass

with term.mouse_enabled(report_motion=True, report_pixels=True):
    # Track all motion with pixel coordinates
    pass
```

bracketed_paste(*timeout: float = 5.0*) → Generator[None, None, None]

Context manager for enabling bracketed paste mode.

When enabled, pasted text is wrapped with special escape sequences, allowing applications to distinguish pasted content from typed input.

Parameters
timeout (*float*) – Timeout for mode query (default 1.0s)

Example:

```
with term.bracketed_paste():
    inp = term.inkey()
    if inp.name == 'BRACKETED_PASTE':
        pasted_text = inp.text
        print(f"You pasted: {pasted_text}")
```

synchronized_output(*timeout: float = 5.0*) → Generator[None, None, None]

Context manager for enabling synchronized output mode.

Buffers all terminal output until the context exits, eliminating screen flicker during redraws. Perfect for animations and full-screen updates.

Parameters
timeout (*float*) – Timeout for mode query (default 1.0s)

focus_events(*timeout: float = 5.0*) → Generator[None, None, None]

Context manager for enabling focus event reporting.

Reports when the terminal window gains or loses focus, useful for pausing animations or updating status indicators.

Parameters
timeout (*float*) – Timeout for mode query (default 1.0s)

notify_on_resize(*timeout: float = 5.0*) → Generator[None, None, None]

Context manager for enabling in-band window resize notifications.

When enabled, the terminal will automatically send resize events as in-band sequences whenever the window size changes. These events are received by *inkey()* as keystroke events with *name* equal to `'RESIZE_EVENT'`.

The new dimensions are automatically cached and available immediately through the standard *height*, *width*, *pixel_height*, and *pixel_width* properties without additional ioctl system calls.

This is the preferred method for handling terminal resizes as it avoids the race conditions and signal handling complexity of SIGWINCH on Unix systems, and provides a consistent cross-platform API.

Parameters
timeout (*float*) – Timeout for mode query (default 1.0s)

Example:

```
with term.notify_on_resize():
    while True:
        inp = term.inkey(timeout=0.1)
        if inp.name == 'RESIZE_EVENT':
            # Dimensions updated automatically
            redraw_display(term.height, term.width)
        elif inp == 'q':
            break
```

get_sixel_height_and_width(*timeout: float | None = 5.0, force: bool = False*) → Tuple[int, int]

Query sixel graphics pixel dimensions.

Returns the maximum height and width in pixels for sixel graphics rendering. Detection order (from most to least reliable):

1. XTWINOPS 16t (CSI 16 t) - Character cell size, multiplied by rows/cols
2. XTWINOPS 14t (CSI 14 t) - Text area size in pixels
3. XTSMGRAPHICS - Sixel graphics query
4. TIOCSWINSZ / In-band resize - System ioctl / event pixel dimensions

The cell-based calculation (method 1) is preferred because it accounts for the actual drawable text area, excluding window margins and decorations.

When *is_a_tty* is False, no sequences are transmitted or response awaited, and `(-1, -1)` is returned without inquiry.

Parameters

- **timeout** (*float*) – Timeout in seconds for queries
- **force** (*bool*) – Bypass cache and re-query the terminal

Return type
tuple

Returns
`(height, width)` in pixels, or `(-1, -1)` if unsupported/timeout

get_sixel_colors(*timeout: float | None = 5.0*, *force: bool = False*) → int

Query number of sixel color registers (XTSMGRAPHICS).

Returns the maximum number of color registers available for sixel graphics rendering. If XTSMGRAPHICS query fails but the terminal advertises Sixel support via DA1 (Device Attributes), returns 256 as a sensible default.

When *is_a_tty* is False, no sequences are transmitted or response awaited, and `-1` is returned without inquiry.

Parameters

- **timeout** (*float*) – Timeout in seconds for both possible queries
- **force** (*bool*) – Bypass cache and re-query the terminal

Return type
int

Returns
Number of color registers, 256 if DA1 advertises sixel but XTSMGRAPHICS fails, or `-1` if unsupported/timeout

get_cell_height_and_width(*timeout: float | None = 5.0*, *force: bool = False*) → Tuple[int, int]

Query character cell pixel dimensions (XTWINOPS).

Returns the height and width in pixels of a single character cell.

When *is_a_tty* is False, no sequences are transmitted or response awaited, and `(-1, -1)` is returned without inquiry.

Parameters

- **timeout** (*float*) – Timeout in seconds for the query
- **force** (*bool*) – Bypass cache and re-query the terminal

Return type
tuple

Returns
`(height, width)` in pixels, or `(-1, -1)` if unsupported/timeout

get_kitty_keyboard_state(*timeout: float | None = 5.0*, *force: bool = False*) → *KittyKeyboardProtocol* | None

Query the current Kitty keyboard protocol flags.

Sends a Kitty keyboard protocol query to the terminal and returns a `KittyKeyboardProtocol` instance with the current flags. This method is not normally used directly, rather it is used by the

enable_kitty_keyboard() context manager on entrance to discover and restore the previous state on exit.

When *is_a_tty* is False, no sequences are transmitted or response awaited, and `None` is returned.

In many cases a `timeout` value (in seconds) should be set, as it is possible for a terminal that succeeds *is_a_tty* to fail to respond to either Kitty keyboard protocol state request, or the simple device attribute request query carried with it! And not just "dumb" terminals fail to respond, even some fairly modern terminals like Konsole.

If a Kitty keyboard protocol query fails to respond within the `timeout` specified, `None` is returned. If this was the first Kitty keyboard protocol query, all subsequent queries return `None` unless `force=True` is set.

No state caching is performed - each call re-queries the terminal unless the first query previously failed (sticky failure) and `force=False`.

Parameters

- **timeout** (*float*) – Timeout in seconds to await terminal response
- **force** (*bool*) – Force active terminal inquiry in all cases

Return type
KittyKeyboardProtocol or None

Returns
KittyKeyboardProtocol instance with current flags, or None if unsupported/timeout

enable_kitty_keyboard(*, *disambiguate: bool = True*, *report_events: bool = False*, *report_alternates: bool = False*, *report_all_keys: bool = False*, *report_text: bool = False*, *mode: int = 1*, *timeout: float | None = 5.0*, *force: bool = False*) → Generator[None, None, None]

Context manager that enables Kitty keyboard protocol features.

Parameters

- **disambiguate** (*bool*) – Enable disambiguated escape codes (fixes issues with Esc vs sequences)
- **report_events** (*bool*) – Report key repeat and release events
- **report_alternates** (*bool*) – Report shifted and base layout keys for shortcuts
- **report_all_keys** (*bool*) – Report all keys as escape codes (including text keys)
- **report_text** (*bool*) – Report associated text with key events (requires report_all_keys)
- **mode** (*int*) – Protocol mode (1=set/clear specified flags, 2=set only, 3=clear only)
- **timeout** (*float*) – Timeout for querying current flags before setting new ones
- **force** (*bool*) – Force sequences to be emitted even if timeout previously occurred

Always queries current state before setting new flags and restores previous state on exit.

Example:

```
with term.enable_kitty_keyboard(disambiguate=True):
    # Now Alt+C won't conflict with Ctrl+C
    key = term.inkey()
    if key.alt and key.is_alt('c'):
        print("Alt+C pressed")
```

> **Note**
>
> A `timeout` value should be set to avoid blocking when the terminal does not respond to DA1 or kitty protocol queries, which may happen with some kinds of "dumb" terminals, even some modern terminals like Konsole.

fullscreen() → Generator[None, None, None]

Context manager that switches to secondary screen, restoring on exit.

Under the hood, this switches between the primary screen buffer and the secondary one. The primary one is saved on entry and restored on exit. Likewise, the secondary contents are also stable and are faithfully restored on the next entry:

```
with term.fullscreen():
    main()
```

> **Note**
>
> There is only one primary and one secondary screen buffer. *fullscreen()* calls cannot be nested, only one should be entered at a time.

hidden_cursor() → Generator[None, None, None]

Context manager that hides the cursor, setting visibility on exit.

with term.hidden_cursor():
    main()

> **Note**
>
> *hidden_cursor()* calls cannot be nested: only one should be entered at a time.

cursor_shape(*style: int | str = 2*) → Generator[None, None, None]

Context manager that sets cursor shape, restoring default on exit.

Parameters
style – A *CursorShape* constant or string name (e.g. `'blinking_bar'`).

Uses DECSCUSR (DEC Set Cursor Style) escape sequences:

```
with term.cursor_shape(term.CursorShape.BLINKING_BAR):
    main()
```

String names are also accepted:

```
with term.cursor_shape('steady_underline'):
    main()
```

On exit, the cursor is reset to the terminal default (DECSCUSR 0).

> **Note**

DECSCUSR is supported by most modern terminals including xterm, VTE-based terminals, iTerm2, Windows Terminal, kitty, ghostty, alacritty, and WezTerm.

no_line_wrap() → Generator[None, None, None]

Context manager that disables line wrapping, enabling on exit.

Uses DEC Auto Wrap Mode (DECAWM) to control whether text reaching the right edge wraps to the next line:

```
with term.no_line_wrap():
    print(term.move_x(0) + 'x' * 200)  # Clips, doesn't wrap
```

On exit, line wrapping is unconditionally enabled as terminals should prefer line-wrapping mode for normal operation.

Note

no_line_wrap() calls cannot be nested: only one should be entered at a time.

scroll_region(*top: int = 0*, *height: int | None = None*) → Generator[None, None, None]

Context manager that sets a scrolling region, resetting on exit.

Parameters

- **top** (*int*) – Top row of the scrolling region (0-indexed). Defaults to 0.
- **height** (*int*) – Number of rows in the scrolling region. Defaults to `self.height - top` (extending to the bottom of the screen).

Returns
a context manager.

Return type
Iterator

A scrolling region restricts scrolling to a portion of the screen. When text scrolls within the region, content outside the region remains fixed. This is useful for creating interfaces with fixed headers, footers, or status bars:

```
with term.fullscreen(), term.scroll_region(top=1, height=term.height - 2):
    # row 0 and the last row remain fixed
    # scrolling happens only in the middle region
    for i in range(100):
        print(f'Line {i}')
```

The cursor position may be reset to the home position when the scroll region changes. Use *move_yx()* to reposition as needed after entering the context.

Note

scroll_region() calls cannot be nested: only one should be entered at a time.

move_xy(*x: int*, *y: int*) → str

A callable string that moves the cursor to the given `(x, y)` screen coordinates.

Parameters

- **x** (*int*) – horizontal position, from left, *0*, to right edge of screen, *self.width - 1*.
- **y** (*int*) – vertical position, from top, *0*, to bottom of screen, *self.height - 1*.

Return type
ParameterizingString

Returns
Callable string that moves the cursor to the given coordinates

move_yx(*y: int*, *x: int*) → str

A callable string that moves the cursor to the given `(y, x)` screen coordinates.

Parameters

- **y** (*int*) – vertical position, from top, *0*, to bottom of screen, *self.height - 1*.
- **x** (*int*) – horizontal position, from left, *0*, to right edge of screen, *self.width - 1*.

Return type
ParameterizingString

Returns
Callable string that moves the cursor to the given coordinates

property move_left: *FormattingOtherString*

Move cursor 1 cells to the left, or callable string for n>1 cells.

property move_right: *FormattingOtherString*

Move cursor 1 or more cells to the right, or callable string for n>1 cells.

property move_up: *FormattingOtherString*

Move cursor 1 or more cells upwards, or callable string for n>1 cells.

property move_down: *FormattingOtherString*

Move cursor 1 or more cells downwards, or callable string for n>1 cells.

property color: *NullCallableString* | *ParameterizingString*

A callable string that sets the foreground color.

Return type
ParameterizingString

The capability is unparameterized until called and passed a number, at which point it returns another string which represents a specific color change. This second string can further be called to color a piece of text and set everything back to normal afterward.

This should not be used directly, but rather a specific color by name or *color_rgb()* value.

color_rgb(*red: int*, *green: int*, *blue: int*) → *FormattingString*

Provides callable formatting string to set foreground color to the specified RGB color.

Parameters

- **red** (*int*) – 8-bit RGB value of Red (0-255).
- **green** (*int*) – 8-bit RGB value of Green (0-255).
- **blue** (*int*) – 8-bit RGB value of Blue (0-255).

Return type
FormattingString

Returns
Callable string that sets the foreground color.

If the terminal does not support RGB color, the nearest supported color will be determined using *color_distance_algorithm*.

color_hex(*hex_color: str*) → *FormattingString*

Provides callable formatting string to set foreground color from hex.

Parameters
hex_color (*str*) – Hex color in #RGB, #RRGGBB, or #RRRRGGGGBBBB format. The # prefix is optional.

Return type
FormattingString

Returns
Callable string that sets the foreground color.

If the terminal does not support RGB color, the nearest supported color will be determined using *color_distance_algorithm*.

See *hex_to_rgb()* for supported hex formats.

property on_color: *NullCallableString* | *ParameterizingString*

A callable capability that sets the background color.

Return type
ParameterizingString

on_color_rgb(*red: int*, *green: int*, *blue: int*) → *FormattingString*

Provides callable formatting string to set background color to the specified RGB color.

Parameters

- **red** (*int*) – 8-bit RGB value of Red (0-255).
- **green** (*int*) – 8-bit RGB value of Green (0-255).
- **blue** (*int*) – 8-bit RGB value of Blue (0-255).

Return type
FormattingString

Returns
Callable string that sets the background color.

If the terminal does not support RGB color, the nearest supported color will be determined using *color_distance_algorithm*.

on_color_hex(*hex_color: str*) → *FormattingString*

Provides callable formatting string to set background color from hex.

Parameters
hex_color (*str*) – Hex color in #RGB, #RRGGBB, or #RRRRGGGGBBBB format. The # prefix is optional.

Return type
FormattingString

Returns
Callable string that sets the background color.

If the terminal does not support RGB color, the nearest supported color will be determined using *color_distance_algorithm*.

See *hex_to_rgb()* for supported hex formats.

formatter(*value: str*) → *NullCallableString* | *FormattingString*

Provides callable formatting string to set color and other text formatting options.

Parameters
value (*str*) – Sugary, ordinary, or compound formatted terminal capability, such as "red_on_white", "normal", "red", or "bold_on_black".

Return type
FormattingString or NullCallableString

Returns
Callable string that sets color and other text formatting options

Calling `term.formatter('bold_on_red')` is equivalent to `term.bold_on_red`, but a string that is not a valid text formatter will return a `NullCallableString`. This is intended to allow validation of text formatters without the possibility of inadvertently returning another terminal capability.

rgb_downconvert(*red: int*, *green: int*, *blue: int*) → int

Translate an RGB color to a color code of the terminal's color depth.

This method is only be used to downconvert for terminals of 256 or fewer colors.

Parameters

- **red** (*int*) – RGB value of Red (0-255).
- **green** (*int*) – RGB value of Green (0-255).
- **blue** (*int*) – RGB value of Blue (0-255).

Return type
int

Returns
Color code of downconverted RGB color

property normal: str

A capability that resets all video attributes.

Return type
str

`normal` is an alias for `sgr0` or `exit_attribute_mode`. Any styling attributes previously applied, such as foreground or background colors, reverse video, or bold are reset to defaults.

link(*url: str*, *text: str*, *url_id: str = ''*) → str

Display `text` that when touched or clicked, navigates to `url`.

Optional `url_id` may be specified, so that non-adjacent cells can reference a single target, all cells painted with the same "id" will highlight on hover, rather than any individual one, as described in "Hovering and underlining the id parameter" of gist https://gist.github.com/egmontkob/eb114294efbcd5adb1944c9f3cb5feda.

Parameters

- **url**¶ (*str*) – Hyperlink URL.
- **text**¶ (*str*) – Clickable text.

- **url_id** (*str*) – Optional 'id'.

Return type
str

Returns
String of `text` as a hyperlink to `url`.

set_window_title(*title: str*, *mode: int = 0*) → str

Return sequence to set the terminal title.

Uses xterm OSC (Operating System Command) sequences to set the window and/or icon title.

Parameters

- **title** (*str*) – Title text. Any embedded escape or BEL characters are stripped to prevent sequence injection.
- **mode** (*int*) – OSC mode – 0 sets both icon name and window title, 1 sets icon name only, 2 sets window title only.

Return type
str

Returns
Escape sequence string that sets the terminal title, or empty string when *does_styling* is `False`.

window_title(*title: str*, *mode: int = 0*) → Generator[None, None, None]

Context manager that sets terminal title, restoring on exit.

Uses the xterm title stack (XTWINOPS push/pop) to save and restore the previous title. Not all terminals support the title stack – those that do not will simply set the title without restoring it on exit.

Parameters

- **title** (*str*) – Title text.
- **mode** (*int*) – OSC mode – 0 sets both icon name and window title, 1 sets icon name only, 2 sets window title only.

progress_bar(*state: str | int*, *value: int | None = None*) → str

Return OSC 9;4 sequence for terminal progress bar support.

Support for 'OSC 9;4' cannot be detected. It is generally safe to write to unsupported terminals. Some terminals display a graphical progress indicator in the taskbar and window menu or status bar matching the given value (0-100). Supported by at least Windows Terminal.exe, ConEmu, Ghostty, kitty, and iTerm2.

Parameters

- **state** – Progress state. Accepts either an integer or a string name:
  - `0` or `'clear'`: Remove progress indicator
  - `1` or `'normal'`: Set progress to *value* (0 to 100)
  - `2` or `'error'`: Error state (typically red)
  - `3` or `'indeterminate'`: Indeterminate / pulsing
  - `4` or `'paused'`: Paused state (typically yellow)
- **value** (*int*) – Progress value as integer of percent (0 to 100). Required when state is `'normal'` or `1`. Terminals are expected to ignore 'value' for all other states.

Return type
str

Returns
OSC 9;4 escape sequence, or `''` when *does_styling* is `False`.

Raises
ValueError – on bad `state` identifier or invalid or out of bounds `value`.

See also

https://ghostty.org/docs/vt/osc/conemu#change-progress-state-(osc-94)

property stream: IO[str]

Read-only property: stream the terminal outputs to.

This is a convenience attribute. It is used internally for implied writes performed by context managers *hidden_cursor()*, *fullscreen()*, *location()*, and *keypad()*.

property number_of_colors: int

Number of colors supported by terminal.

Common return values are 0, 8, 16, 256, or 1 << 24.

This may be used to test whether the terminal supports colors, and at what depth, if that's a concern.

If this property is assigned a value of 88, the value 16 will be saved. This is due to the the rarity of 88 color support and the inconsistency of behavior between implementations.

Assigning this property to a value other than 0, 4, 8, 16, 88, 256, or 1 << 24 will raise an `AssertionError`.

property color_distance_algorithm: str

Color distance algorithm used by *rgb_downconvert()*.

The slowest, but most accurate, 'cie2000', is default. Other available options are 'rgb', 'rgb-weighted', 'cie76', and 'cie94'. This function is only be used to downconvert for terminals of 256 or fewer colors.

ljust(*text: str*, *width: SupportsIndex | None = None*, *fillchar: str = ' '*) → str

Left-align `text`, which may contain terminal sequences.

Parameters

- **text** (*str*) – String to be aligned
- **width** (*int*) – Total width to fill with aligned text. If unspecified, the whole width of the terminal is filled.
- **fillchar** (*str*) – String for padding the right of `text`

Return type
str

Returns
String of `text`, left-aligned by `width`.

rjust(*text: str*, *width: SupportsIndex | None = None*, *fillchar: str = ' '*) → str

Right-align `text`, which may contain terminal sequences.

Parameters

- **text** (*str*) – String to be aligned

- **width** (*int*) – Total width to fill with aligned text. If unspecified, the whole width of the terminal is used.
- **fillchar** (*str*) – String for padding the left of `text`

Return type
str

Returns
String of `text`, right-aligned by `width`.

center(*text: str*, *width: SupportsIndex | None = None*, *fillchar: str = ' '*) → str

Center `text`, which may contain terminal sequences.

Parameters

- **text** (*str*) – String to be centered
- **width** (*int*) – Total width in which to center text. If unspecified, the whole width of the terminal is used.
- **fillchar** (*str*) – String for padding the left and right of `text`

Return type
str

Returns
String of `text`, centered by `width`

text_sized(*text: str*, *scale: int = 1*, ***, *width: int = 0*, *numerator: int = 0*, *denominator: int = 0*, *vertical_align: int | str = 0*, *horizontal_align: int | str = 0*) → str

Wrap `text` in a text sizing escape sequence (OSC 66).

Returns `text` unchanged when the terminal does not support text sizing, providing graceful degradation.

Parameters

- **text** (*str*) – Text payload.
- **scale** (*int*) – Scale factor (1 to 7).
- **width** (*int*) – Width in cells (0 to 7). 0 means auto-calculate from the inner text.
- **numerator** (*int*) – Fractional scaling numerator (0 to 15).
- **denominator** (*int*) – Fractional scaling denominator (0 to 15).
- **vertical_align** – Vertical alignment: ('top', 'bottom', 'center'), or protocol code (0=top, 1=bottom, 2=center).
- **horizontal_align** – Horizontal alignment. ('left', 'right', 'center') or protocol code (0=left, 1=right, 2=center).

Return type
str

Returns
Text wrapped in an OSC 66 escape sequence, or plain `text` on unsupported terminals.

Raises

- **ValueError** – when the encoded `text` exceeds 4096 bytes.
- **ValueError** – when `vertical_align` or `horizontal_align` is an unrecognized string value.

See also

Kitty Text Sizing Protocol

Note

Because the first call to *Terminal.text_sized()* will cause the side effect of writing destructive spaces to the columns and row following the current cursor position as required for detection of kitty text sizing protocol, it is suggested to first call *Terminal.does_text_sizing()* during program initialization.

The result is permanently cached and all further calls to these methods return immediately without this side effect.

clip(*text: str*, *start: SupportsIndex = 0*, *end: SupportsIndex = -1*, ***kwargs: object*) → str

Return a window of `text` spanning display columns `start` to `end`.

Terminal sequences are retained, wide characters split at a boundary are replaced by `fillchar`, and horizontal cursor movement is interpreted by `wcwidth.clip()`. The detected *ambiguous_width* and *term_program* are applied.

Parameters

- **text** (*str*) – Text to clip, may contain terminal sequences
- **start** (*int*) – Absolute starting column, inclusive (default 0)
- **end** (*int*) – Absolute ending column, exclusive; -1 (default) means "to the end of the line"
- ****kwargs** – See `wcwidth.clip()`

Return type
str

Returns
`text` clipped to display columns (start, end)

```
>>> term.clip('hello world', 6)
'world'
```

truncate(*text: str*, *width: SupportsIndex | None = None*) → str

Truncate `text` to `width` printable characters, retaining terminal sequences.

Wide characters (such as CJK or emoji) that would partially exceed `width` are replaced with space padding to maintain exact width.

Parameters

- **text** (*str*) – Text to truncate
- **width** (*int*) – The width to truncate to. If unspecified, the whole width of the terminal is used.

Return type
str

Returns
`text` truncated to exactly `width` printable characters

```
>>> term.truncate('xyz\x1b[0;3m', 2)
'xy\x1b[0;3m'
```

length(*text: str*) → int

Return printable length of a string containing sequences.

Returns the maximum horizontal cursor extent reached while processing the string. Backspace and cursor-left movements do not reduce the length below the maximum position reached.

Parameters
text (*str*) – String to measure. May contain terminal sequences.

Return type
int

Returns
The maximum terminal character cell position reached

Wide characters that consume 2 character cells are supported:

```
>>> term = Terminal()
>>> term.length(term.clear + term.red(''))
10
```

Note

Sequences such as 'clear', which is considered as a "movement sequence" because it would move the cursor to (y, x)(0, 0), are evaluated as a printable length of *0*.

strip(*text: str*, *chars: str | None = None*) → str

Return `text` without sequences and leading or trailing whitespace.

Return type
str

Returns
Text with leading and trailing whitespace removed

```
>>> term.strip(' \x1b[0;3m xyz ')
'xyz'
```

rstrip(*text: str*, *chars: str | None = None*) → str

Return `text` without terminal sequences or trailing whitespace.

Return type
str

Returns
Text with terminal sequences and trailing whitespace removed

```
>>> term.rstrip(' \x1b[0;3m xyz ')
'  xyz'
```

lstrip(*text: str*, *chars: str | None = None*) → str

Return `text` without terminal sequences or leading whitespace.

Return type
str

Returns
Text with terminal sequences and leading whitespace removed

```
>>> term.lstrip(' \x1b[0;3m xyz ')
'xyz '
```

strip_seqs(*text: str*) → str

Return `text` stripped of only its terminal sequences.

Return type
str

Returns
Text with terminal sequences removed

```
>>> term.strip_seqs('\x1b[0;3mxyz')
'xyz'
>>> term.strip_seqs(term.cuf(5) + term.red('test'))
'     test'
```

Note

Non-destructive sequences that adjust horizontal distance (such as `\b` or `term.cuf(5)`) are replaced by destructive space or erasing.

split_seqs(*text: str*, *maxsplit: int = 0*) → List[str]

Return `text` split by individual character elements and sequences.

Parameters

- **text** (*str*) – String containing sequences
- **maxsplit** (*int*) – When maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list (same meaning is argument for `re.split()`).

Return type
list[str]

Returns
List of sequences and individual characters

```
>>> term.split_seqs(term.underline('xyz'))
['\x1b[4m', 'x', 'y', 'z', '\x1b(B', '\x1b[m']
```

```
>>> term.split_seqs(term.underline('xyz'), 1)
['\x1b[4m', r'xyz\x1b(B\x1b[m']
```

wrap(*text: str*, *width: int | None = None*, ***kwargs: object*) → List[str]

Text-wrap a string, returning a list of wrapped lines.

Parameters

- **text** (*str*) – Unlike `textwrap.wrap()`, `text` may contain terminal sequences, such as colors, bold, or underline. By default, tabs in `text` are expanded by `string.expandtabs()`.
- **width** (*int*) – Unlike `textwrap.wrap()`, `width` will default to the width of the attached terminal.
- ****kwargs** – See `textwrap.TextWrapper`

Return type
list

Returns
List of wrapped lines

Raises
ValueError – for non-positive values of `width`.

See `textwrap.TextWrapper` for keyword arguments that can customize wrapping behaviour.

getch(*decode_latin1: bool = False*) → str

Read, decode, and return the next byte from the keyboard stream.

Parameters
decode_latin1 (*bool*) – If True, decode byte as latin-1 (for legacy mouse sequences with 8-bit coordinates).

Return type
unicode

Returns
a single unicode character, or `''` if a multi-byte sequence has not yet been fully received.

Raises
EOFError – When the keyboard stream has reached end-of-file.

This method name and behavior mimics curses `getch(void)`, and it supports *inkey()*, reading only one byte from the keyboard string at a time. This method should always return without blocking if called after *kbhit()* has returned True.

Implementers of alternate input stream methods should override this method.

ungetch(*text: str*) → None

Buffer input data to be discovered by next call to *inkey()*.

Parameters
text (*str*) – String to be buffered as keyboard input.

kbhit(*timeout: float | None = None*) → bool

Return whether a keypress has been detected on the keyboard.

This method is used by *inkey()* to determine if a byte may be read using *getch()* without blocking. The standard implementation simply uses the `select.select()` call on stdin.

Parameters
timeout (*float*) – When `timeout` is 0, this call is non-blocking, otherwise blocking indefinitely until keypress is detected when None (default). When `timeout` is a positive number, returns after `timeout` seconds have elapsed (float).

Return type
bool

Returns
True if a keypress is awaiting to be read on the keyboard attached to this terminal. When input is not a terminal, False is always returned.

cbreak() → Generator[None, None, None]

Allow each keystroke to be read immediately after it is pressed.

This is a context manager for `tty.setcbreak()`.

This context manager activates 'rare' mode, the opposite of 'cooked' mode: On entry, `tty.setcbreak()` mode is activated disabling line-buffering of keyboard input and turning off automatic echo of input as output.

> **Note**
>
> You must explicitly print any user input you would like displayed. If you provide any kind of editing, you must handle backspace and other line-editing control functions in this mode as well!

Normally, characters received from the keyboard cannot be read by Python until the *Return* key is pressed. Also known as *cooked* or *canonical input* mode, it allows the tty driver to provide line-editing before shuttling the input to your program and is the (implicit) default terminal mode set by most unix shells before executing programs.

Technically, this context manager sets the `termios` attributes of the terminal attached to `sys.__stdin__`.

> **Note**
>
> `tty.setcbreak()` sets `VMIN = 1` and `VTIME = 0`, see http://www.unixwiz.net/techtips/termios-vmin-vtime.html

raw() → Generator[None, None, None]

A context manager for `tty.setraw()`.

Although both *cbreak()* and *raw()* modes allow each keystroke to be read immediately after it is pressed, Raw mode disables processing of input and output by the terminal driver.

In cbreak mode, special input characters such as `^C` or `^S` are interpreted by the terminal driver and excluded from the stdin stream. In raw mode these values are received by the *inkey()* method.

Because output processing is not done by the terminal driver, the newline `'\n'` is not enough, you must also print carriage return to ensure that the cursor is returned to the first column:

```
with term.raw():
    print("printing in raw mode", end="\r\n")
```

keypad() → Generator[None, None, None]

Context manager that enables directional keypad input.

On entry, this puts the terminal into "keyboard_transmit" mode by emitting the keypad_xmit (smkx) capability. On exit, it emits keypad_local (rmkx).

On an IBM-PC keyboard with numeric keypad of terminal-type *xterm*, with numlock off, the lower-left diagonal key transmits sequence `\\x1b[F`, translated to *Terminal* attribute `KEY_END`.

However, upon entering *keypad()*, `\\x1b[OF` is transmitted, translating to `KEY_LL` (lower-left key), allowing you to determine diagonal direction keys.

flushinp(*timeout: float = 0*) → str

Unbuffer and return all input available within `timeout`.

When legacy mouse sequence `'\x1b[M'` is detected in input stream, all remaining bytes are decoded as latin1 to handle 8-bit coordinates.

inkey(*timeout: float | None = None*, *esc_delay: float = 0.35*, *capture_cpr: bool = False*) → *Keystroke*

Read and return the next keyboard event within given timeout.

Generally, this should be used inside the *raw()* or *cbreak()* context manager.

Parameters

- **timeout** (*float*) – Number of seconds to wait for a keystroke before returning. When `None` (default), this method may block indefinitely.
- **esc_delay** (*float*) – Time in seconds to block after Escape key is received to await another key sequence beginning with escape such as *KEY_LEFT*, sequence `'\x1b[D'`, before returning a *Keystroke* instance for `KEY_ESCAPE`.

  Users may override the default value of `esc_delay` in seconds, using environment value of `ESCDELAY` as milliseconds, see *ncurses(3)* section labeled *ESCDELAY* for details. Setting the value as an argument to this function will override any such preference.
- **capture_cpr** (*bool*) – Prefer matches of `CPR_RESPONSE` over conflicting vt220 Legacy function keys (eg. `KEY_F3`, `KEY_SHIFT_F3`).

Return type
Keystroke.

Returns
Keystroke, which may be empty (`''`) if `timeout` is specified and keystroke is not received.

> **Note**
>
> When used without the context manager *cbreak()*, or *raw()*, `sys.__stdin__` remains line-buffered, and this function will block until the return key is pressed!

> **Note**
>
> On Windows, a 10 ms sleep is added to the key press detection loop to reduce CPU load. Due to the behavior of `time.sleep()` on Windows, this will actually result in a 15.6 ms delay when using the default time resolution. Decreasing the time resolution will reduce this to 10 ms, while increasing it, which is rarely done, will have a perceptable impact on the behavior.

ncurses(3): https://www.man7.org/linux/man-pages/man3/ncurses.3x.html

async async_inkey(*timeout: float | None = None*, *esc_delay: float = 0.35*, *capture_cpr: bool = False*) → *Keystroke*

Asynchronous version of *inkey()* for use with `asyncio`.

Read and return the next keyboard event within given timeout, yielding control to the asyncio event loop while waiting for input rather than blocking the thread.

Uses `asyncio.AbstractEventLoop.add_reader()` on the keyboard file descriptor, reusing the same `resolve_sequence()`, keymap, and *Keystroke* internals as the synchronous *inkey()*.

Must be called within a *cbreak()* or *raw()* context, just like *inkey()*.

Parameters

- **timeout** (*float*) – Number of seconds to wait for a keystroke before returning. When None (default), this method may block indefinitely.
- **esc_delay** (*float*) – Time in seconds to wait after Escape key is received to disambiguate bare Escape from escape sequences.
- **capture_cpr** (*bool*) – Prefer matches of CPR_RESPONSE over conflicting vt220 Legacy function keys (eg. KEY_F3, KEY_SHIFT_F3).

Return type
Keystroke

Returns
Keystroke, which may be empty ('') if timeout is specified and keystroke is not received.

class WINSZ(*ws_row*, *ws_col*, *ws_xpixel*, *ws_ypixel*)

Structure represents return value of termios.TIOCGWINSZ.

ws_row

rows, in characters

ws_col

columns, in characters

ws_xpixel

horizontal size, pixels

ws_ypixel

vertical size, pixels

Create new instance of WINSZ(ws_row, ws_col, ws_xpixel, ws_ypixel)

CHAPTER

SIXTEEN

PROJECT

Bugs or suggestions? Visit the issue tracker and file an issue. We welcome your bug reports and feature suggestions!

Are you stuck and need **support**? Give stackoverflow a try. If you're still having trouble, we'd like to hear about it! Open an issue in the issue tracker with a well-formed question.

Would you like to **contribute**? That's awesome! Pull Requests are always welcome!

16.1 Fork

Blessed is a fork of blessings. Apologies for the fork, I just couldn't get the *Keyboard Input*, and later *Location* or *Sizing & Alignment* code accepted upstream after two major initiatives, the effort was better spent in a fork, where the code is accepted.

Furthermore, a project in the node.js language of the same name, blessed, is **not** related, or a fork of each other in any way.

16.2 License

Blessed is under the MIT License. See the LICENSE file. Please enjoy!

16.3 Running Tests

Install and run tox:

```
pip install --upgrade tox
tox
```

Py.test is used as the test runner, and with the tox target supporting positional arguments:

```
tox -e py39 -- -x tests/test.py
```

16.4 Formatting

run:

```
tox -e format
```

16.5 Linting

ruj

tox -e lint

Warning

When you contribute a new feature, make sure it is covered by tests.

Likewise, some bug fixes include a test demonstrating the bug and is linted.

16.6 Further Reading

As a developer's API, blessed is often bundled with frameworks and toolsets that dive deeper into Terminal I/O programming than *Terminal* offers. Here are some recommended readings to help you along:

- The terminfo(5) manpage of your preferred posix-like operating system. The capabilities available as attributes of *Terminal* are directly mapped to those listed in the column **Cap-name**.
- The termios(3) of your preferred posix-like operating system.
- The TTY demystified by Linus Åkesson.
- A Brief Introduction to Termios by Nelson Elhage.
- Richard Steven's Advance Unix Programming ("AUP") provides two very good chapters, "Terminal I/O" and "Pseudo Terminals".
- GNU's The Termcap Manual by Richard M. Stallman.
- Chapter 4 of CUNY's course material for *Introduction to System Programming*, by Stewart Weiss
- Chapter 11 of the IEEE Open Group Base Specifications Issue 7, "General Terminal Interface"
- The GNU C Library documentation, section Low-Level Terminal Interface
- The source code of many popular terminal emulators. If there is ever any question of "the meaning of a terminal capability", or whether or not your preferred terminal emulator actually handles them, read the source! Many modern terminal emulators are now based on libvte.
- The source code of the tty(4), pty(7), and the given "console driver" for any posix-like operating system. If you search thoroughly enough, you will eventually discover a terminal sequence decoder, usually a `case` switch that translates `\x1b[0m` into a "reset color" action towards the video driver. Though `tty.c` linked here is probably not the most interesting, it can get you started:
  - FreeBSD
  - OpenBSD
  - Illumos (Solaris)
  - Minix
  - Linux
- Thomas E. Dickey has been maintaining xterm, as well as a primary maintainer of many related packages such as ncurses for quite a long while. His consistent, well-documented, long-term dedication to xterm, curses, and the many related projects is world-renown.
- termcap & terminfo (O'Reilly Nutshell) by Linda Mui, Tim O'Reilly, and John Strang.

- Note that System-V systems, also known as Unix98 (SunOS, HP-UX, AIX and others) use a Streams interface. On these systems, the ioctl(2) interface provides the `PUSH` and `POP` parameters to communicate with a Streams device driver, which differs significantly from Linux and BSD.

  Many of these systems provide compatible interfaces for Linux, but they may not always be as complete as the counterpart they emulate, most especially in regards to managing pseudo-terminals.

16.7 The misnomer of ANSI

When people say 'ANSI', they are discussing:

- Standard ECMA-48: Control Functions for Coded Character Sets
- ANSI X3.64 from 1981, when the American National Standards Institute adopted the ECMA-48 as standard, which was later withdrawn in 1997 (so in this sense it is *not* an ANSI standard).
- The ANSI.SYS driver provided in MS-DOS and clones. The popularity of the IBM Personal Computer and MS-DOS of the era, and its ability to display colored text further populated the idea that such text "is ANSI".
- The various code pages used in MS-DOS Personal Computers, providing "block art" characters in the 8th bit (int 127-255), paired with ECMA-48 sequences supported by the MS-DOS ANSI.SYS driver to create artwork, known as ANSI art.
- The ANSI terminal database entry and its many descendants in the terminfo database. This is mostly due to terminals compatible with SCO UNIX, which was the successor of Microsoft's Xenix, which brought some semblance of the Microsoft DOS ANSI.SYS driver capabilities.
- Select Graphics Rendition (SGR) on vt100 clones, which include many of the common sequences in ECMA-48.
- Any sequence started by the Control-Sequence-Inducer is often mistakenly termed as an "ANSI Escape Sequence" though not appearing in ECMA-48 or interpreted by the ANSI.SYS driver. The adjoining phrase "Escape Sequence" is so termed because it follows the ASCII character for the escape key (ESC, `\x1b`).

CHAPTER

SEVENTEEN

VERSION HISTORY

1.53

- change: type of *term_program* changed from `Union[str,bool]` to `str`, PR #433
- improve: performance of *clip()* and *truncate()*, PR #434.
- bugfix: small mathematics errors in *dist_cie2000()* and *dist_cie94()* now match the test data published by Sharma, Wu & Dalal (2005), #424.

1.52

- bugfix: *wrap()* discarded lines containing only spaces or tabs when `drop_whitespace=False`, they are now wrapped and retained, PR #423.

1.51

- introduced: *clip()* to extract a "window" of displayed columns from a string, a more precise version of *truncate()*, retaining terminal sequences, PR #420.
- improved: *length()*, *ljust()*, *rjust()*, *center()*, *truncate()*, and *wrap()* with terminal-specific Corrections by detecting the terminal software name at class-initialization, #420.
- change: East Asian ambiguous width is not measured at class-initialization. Call *detect_ambiguous_width()* to measure it, which assigns and caches *ambiguous_width*, or set `AMBIGUOUS_WIDE` to `1` or `2` to supply the value without measurement, PR #420.
- introduced: *detect_term_program()* to detect the terminal software name, or to detect it again after initialization, as needed by a terminal attached later, such as a remote stream, PR #420.
- change: *get_software_version()* returns the `TERM_PROGRAM` and `TERM_PROGRAM_VERSION` environment values when defined, without inquiry, and these values also override the Corrections detection and the XTVERSION query otherwise made for *term_program*, PR #420.
- deprecated: Python 3.8 and earlier no longer supported. PR #420.

1.50

- change: default timeout of automatic terminal queries from 1 to 5 seconds with environment variable `BLESSED_QUERY_TIMEOUT_SECONDS` override, PR #414.
- bugfix: Do not automatic query XTGETTCAP for older CONPTY builds (Windows Server 2022), which displays control codes as visible text PR #411.
- bugfix: *mouse_enabled()* failed to yield `MOUSE_*` keystrokes on older CONPTY builds (Windows Server 2022), where the console's "QuickEdit mode" blocks transmission, PR #412.
- bugfix: an XTGETTCAP received after query time out was returned by *inkey()* as errant keystrokes. It is no longer yielded by *inkey()*, PR #413.

- bugfix: *does_iterm2_graphics()* results are now more accurate, by drawing a transparent pixel, PR #416.
- improve: performance of *does_xtgettcap()*, PR #415.
- improve: skip some automatic queries for Apple's Terminal.app that leak VT100 codes as output PR #417.
- change: default timeout of automatic terminal queries from 1 to 5 seconds with environment variable `BLESSED_QUERY_TIMEOUT_SECONDS` override, PR #414.

1.49

- introduce: *get_font_coverage()* reports codepoints the terminal font has a glyph for, by mintty's `OSC 7771` Font Glyph Coverage Enquiry or the `APC 25a1` Glyph Protocol, PR #407.
- bugfix: *Legacy Mouse Protocols* event names and button values, mouse release events returned by *inkey()* were always `MOUSE_LEFT_RELEASED` for all buttons, they now report `MOUSE_RELEASED`, PR #406.
- bugfix: Mouse buttons 6 through 11 were erroneously decoded as left, middle, or right instead of `MOUSE_BUTTON_6` through `MOUSE_BUTTON_11`, PR #406.
- improve: performance of automatic terminal query replies, PR #405.
- improve: all parameterized capabilities are now memorized, about 50x faster, PR #404.
- bugfix: *truncate()* by bump of dependency `wcwidth>=0.8.3`, #402.

1.48

- bugfix: *async_inkey()* dropped keystrokes while another task is busy, #401.

1.47

- bugfix: *does_sixel()* now returns `True` for SyncTERM using special-case matching of its illegal DA1 response.
- bugfix: legacy SGR mouse decoder reported no-button motion event (mode 1003) as `LEFT_MOTION` instead of `MOTION`, PR #398.
- bugfix: match keyboard input `\x1b\n` as `KEY_ALT_ENTER` instead of `KEY_CTRL_ALT_J`.

1.46

- bugfix: *does_sixel()* failed to detect DA1 and caused the response to "leak" into next call to *inkey()* for some terminals of unmatched patterns (e.g., kitty).
- improve: add new sugar for extended cap-defined styles, like 'clear_scrollback' (common), 'strikethrough', and 'overline' (not common).
- improve: Truecolor support by unique TERM, improves remote sessions like SSH that do not forward `COLORTERM`, and, for the few terminals that do not respond to XTGETTCAP request for `RGB`: (kitty).

1.45

- bugfix: name consistency of kitty keyboard modifiers PR #390.

1.44

- improve: reduce errant XTGETTCAP output for Terminal.app and ConEmu.exe PR #385.

1.43

- bugfix: regression of XTGETTCAP responses leaking into first call for empty/non-response terminals (libvte/Gnome Terminal), in versions 1.40 to 1.42 PR #383.

1.42

- bugfix: regression in *cbreak()* and *raw()* were not thread-safe broken in versions 1.40 and 1.41, remove signal ignore of SIGTTOU #380.

1.41

- bugfix: *get_location()* broken in 1.40, returned a generator instead of a tuple. #378.

1.40

- improved: jinxed is **now required on all platforms**, providing a curses-free and singleton-free implementation of the subset of curses used by blessed. The jinxed 1.5.0 release provides a terminal *capability database <https://jinxed.readthedocs.io/en/stable/capabilities.html#database>* of 45 terminals and their common aliases.
- improved: Class initialization of *Terminal()* now uses XTGETTCAP to determine preferred terminal name `TN`, 24-bit color support `RGB`, number of colors `Co`, *italic*, and *blink* capabilities.

  This improves detection of Terminal *kind* and *number_of_colors* over protocols like serial that cannot forward any environment variables or ssh that do not forward `COLORTERM`.
- introduced: A `UserWarning` is emitted when *__getattr__()* resolves an unknown terminal capability name, helping developers catch typos like `term.bld` (missing `bold`). The warning can be suppressed by setting the environment variable `BLESSED_NOWARN_UNKNOWN_CAPS`.
- bugfix: Fixed internal typo `susimpleript` to the correct terminfo name `ssubm` for the `enter_susimpleript_mode` capability. This was previously masked by curses returning an empty string for unknown capabilities.

1.39

- introduced: *progress_bar()* for OSC 9;4 sequence.
- introduced: *text_sized()* – wrap text in Kitty text sizing protocol (OSC 66) escape sequences, with graceful fallback to plain text when the terminal does not support the protocol.
- introduced: *Keystroke* of name `CPR_RESPONSE` for asynchronous capture of Cursor Position Report responses via *inkey()*. New argument `capture_cpr=True` resolves the legacy F3 key ambiguity and matches against `CPR_RESPONSE`. New properties *cpr_yx* and *cpr_xy* return the decoded cursor coordinates. PR #369.
- improved: *inkey()* raises `EOFError` when keyboard fd is at EOF, rather than returning an empty *Keystroke*. PR #371.
- improved: *ljust()*, *rjust()*, and *center()* now measure text containing hyperlinks, Kitty text sizing protocol sequences, and overtyping (backspace/cursor-left with painter's algorithm), introduced by wcwidth 0.7.0.

1.38

- introduced: *does_osc52_clipboard()*, *clipboard_copy()*, and *clipboard_paste()* to detect, copy, and read from clipboard.
- introduced: *get_color_scheme()*.
- introduced: *does_kitty_query()* for Kitty's XTGETTCAP query extensions.
- introduced: *does_decrqss()* for DECRQSS (Request Status String).
- introduced: *does_styled_underlines()* and *does_colored_underlines()* – detect extended underline styles (`Smulx`) and colored underlines (`Setulc`) via XTGETTCAP.
- introduced: *set_window_title()* and *window_title()* – set the terminal window and/or icon title via xterm OSC sequences, with a context manager that pushes and pops the title stack (XTWINOPS).

- introduced: `DecModeResponse.recognized` and `DecModeResponse.supported` properties – distinguish modes the terminal acknowledges from those it can actually use.
- improved: `Smulx` and `Setulc` added to XTGETTCAP capability list.
- bugfix: `EOF` when stdin is connected to a Pipe (eg. pytest capture) caused infinite loop PR #366.
- bugfix: Background SGR not applying to text/suggestion content in *blessed.line_editor.LineEditor*, now prepend bg_sgr before each content SGR sequence.
- bugfix: OSC responses for 'ST' in addition to 'BEL' terminated iTerm2 capabilities, Kitty notifications, Kitty pointer shapes responses

1.37

- bugfix: legacy CSI letter-form sequences with explicit modifiers and event type (e.g., `\x1b[1;1:1A` for arrow key press) were not resolved to key names, this affected only some terminals, such as Ghostty. PR #363.

1.36

- bugfix: `[` key returned *name* of value `CSI` in Kitty keyboard protocol mode, it now returns `KEY_LEFT_SQUARE_BRACKET`.
- improved: Kitty keyboard protocol now synthesizes *name* for all ASCII alphanumeric and punctuation keys, including unmodified press events (e.g., `KEY_A`, `KEY_PERIOD`, `KEY_LEFT_SQUARE_BRACKET`). Previously, only modified or released/repeated keys received synthesized names.

1.35

- introduced: *cursor_shape()* context manager and *CursorShape* constants for DECSCUSR cursor shape control (block, underline, bar – steady or blinking).
- introduced: methods `key_name()` and `key_value()` allow for better key and value tracking when using kitty keyboard protocol with pressed, released, and repeated events.
- improved: Synthesize *name* for "released" and "repeated" key states, when using the kitty keyboard protocol.
- bugfix: *get_fgcolor()* and *get_bgcolor()* now correctly interprets ST-terminated OSC responses in addition to BEL-terminated.

1.34

- improved: Windows now supports *does_mouse()*, *notify_on_resize()*, and *mouse_enabled()*. More than 100x improvement of response times to keyboard input and automatic response sequences on the Windows platform by using `WaitForSingleObject` initially proposed by cr5451, PR #354.
- bugfix: *async_inkey()* no longer raises NotImplementedError on Windows.

1.33

- bugfix: *blessed.line_editor.LineEditor* exceed limit when using Yank (Ctrl+Y).

1.32

- bugfix: *get_kitty_keyboard_state()* should not check for *does_styling* as a requirement.
- bugfix: *get_fgcolor()* and *get_bgcolor()* now return "no support" value, `(-1, -1, -1)` when *does_styling* is False.
- introduced: *does_kitty_clipboard()*, *does_kitty_pointer_shapes()*, and *does_text_sizing()*
- introduced: *to_dict()* and `DecPrivateMode.BRACKETED_PASTE_MIME` constant (mode 5522).

1.31

- bugfix: *cbreak()* and *raw()* should use `TCSADRAIN` to preserve keystrokes buffered during mode switches, previously `TCSAFLUSH` was used which discarded unread input, dropping keystrokes.
- improved: all terminal query methods now use a CPR boundary guard for fast negative detection of unsupported features: *get_fgcolor()*, *get_bgcolor()*, *get_device_attributes()*, and *get_software_version()*.
- introduced: *async_inkey()* – asyncio-compatible version of *inkey()*.
- introduced: *blessed.line_editor.LineEditor*, a "headless" single-line editor.
- introduced: *get_xtgettcap()* – query terminal capabilities via XTGETTCAP and *get_iterm2_capabilities()* for iTerm2 protocol features.
- introduced: terminal feature detection methods *does_bracketed_paste()*, *does_focus_events()*, *does_grapheme_clustering()* *does_iterm2()*, *does_iterm2_graphics()*, *does_kitty_graphics()*, *does_kitty_notifications()*, *does_synchronized_output()*, and *does_xtgettcap()*.

1.30

- bugfix: Terminal.wrap drops color escape sequences, PR #352.
- bugfix: Prevent rare feedback loops in automatic responses, PR #350.

1.29

- bugfix: *Terminal.get_kitty_keyboard_state()* failed to match PR #348.

1.28

- improved: upgrade to wcwidth 0.5, improving performance and correctness of *Terminal.wrap()*, *Terminal.ljust()*, and related functions, PR #344.
- deprecated: Python 3.7 and earlier no longer supported. PR #344.

1.27

- bugfix missing tests, bin, and docs folder in 1.26 release, PR #341.

1.26

- introduced: *Terminal.detect_ambiguous_width()*, PR #339.
- introduced: *Terminal.no_line_wrap()*, context manager for attributes `disable_line_wrap` and `enable_line_wrap` PR #337.
- introduced: New methods *Terminal.color_hex()* and *Terminal.on_color_hex()* and bugfix parsing of uncommon responses in *Terminal.get_fgcolor()* and bgcolor, PR #335.
- improved: performance of *Terminal.wrap()*, PR #331 by grayjk.
- improved: *Terminal.wrap()* gets break_on_hyphen support, matching behavior of `textwrap.wrap()` by ps06756 in PR #330.
- bugfix: for missing leading sequences in *Terminal.center()*, *Terminal.wrap()*, add support for emojis containing Zero Width Joiner (*u200D*) or Variation Selector-16 (*uFE0F*), PR #338

1.25

- bugfix: The "Copy globals" fix in 1.20 got reverted in release in 1.23 by PR #306, this release re-applies the fix, PR #326.

1.24

- bugfix: `blessed.Terminal`get_sixel_height_and_width()` detection order and improve documentation for window resize event handling PR #320
- bugfix: change default argument `timeout=None` to `timeout=1` in methods requiring automatic response, preventing stalls on "dumb" terminals PR #317

1.23

- introduced: Environment values FORCE_COLOR, CLICOLOR_FORCE, and NO_COLOR overrides `blessed.Terminal` argument value, `force_styling` PR #295
- improved: performance of *rgb_downconvert()* for 256-color terminals, PR #298.
- deprecated: Python 2.7 through 3.6 no longer supported. PR #299.
- improved: keyboard support for modifier keys, PR #306.
- introduced: kitty keyboard protocol support, PR #309.
- introduced: mouse support, Synchronized Output, and support for DEC Private Modes, PR #310
- introduced: methods to determine Sixel, dimensions and color depth PR #312
- introduced: in-band resize support (DEC Private Mode 2048) PR #313
- introduced: XTVersion identification: PR #314
- bugfix: default timeout for get_bgcolor, get_fgcolor PR #315

1.22

- improved: performance of *length()* and *wrap()* via PR #286, PR #287, PR #289, and PR #291
- spelling fixes, PR #278, PR #293

1.21

- bugfix: infinite loop in *wrap()* when "Wide" characters of width 2 (East-Asian or Emoji) are used with a wrap width of 1, and a small performance enhancement, #273 and PR #274 by grayjk merged as PR #275.

1.20

- introduced: *get_fgcolor()* and *get_bgcolor()* to query the terminal for the currently set colors. #237 by stefanholek
- bugfix: Copy globals dict before iterating to avoid RuntimeError in multithreaded applications, #248 by adamnovak

1.19

- introduced: *truncate()* to truncate a string while retaining the sequences, #211 by fishermans-friend
- enhancement: Add small sleep in *kbhit()* on Windows to reduce CPU load #209 by numerlor
- enhancement: *rjust()*, *ljust()*, and *center()* now accept `SupportsIndex` types, #234
- enhancement: When 88 colors is detected, it is now interpreted as 16 colors rather than raising an `AssertionError`, #235
- bugfix: Fix global variable declarations in type annotations, #230 by mwchase

1.18

- bugfix: *split_seqs()* for some sequences like `term.move_left(3)`, #197.
- introduced: type annotations, #192 by dlax.
- bugfix: do not fail when `sys.stdin` is unset, #195 by Olen

- docfix: correct "Bottom of the screen" example to use `end=''` and document about it in location.rst, #188 by pyfisch

1.17

- introduced: *Hyperlinks*, method *link()*, #116.
- introduced: 24-bit color support, detected by `term.number_of_colors == 1 << 24`, and 24-bit color foreground method *color_rgb()* and background method *on_color_rgb()*, as well as 676 common X11 color attribute names are now possible, such as `term.aquamarine_on_wheat`, #60.
- introduced: `term.move_xy`, recommended over built-in `move` capability, as the argument order, `(x, y)` matches the return value of *get_location()*, and all other common graphics library calls, #65.
- introduced: *move_up()*, *move_down()*, *Terminal.move_left()*, *move_right()* which are strings that move the cursor one cell in the respective direction, are now **also** callables for moving *n* cells to the given direction, such as `term.move_right(9)`.
- introduced: *pixel_width* and *pixel_height* for libsixel support or general curiosity.
- introduced: *formatter()* which returns callable formatters for valid text formatters such as 'red' or 'bold_on_red', returning a *NullCallableString* if passed an invalid text formatter.
- bugfix: prevent `ValueError: I/O operation on closed file` on `sys.stdin` in multiprocessing environments, where the keyboard wouldn't work, anyway.
- bugfix: prevent error condition, `ValueError: underlying buffer has been detached` in rare conditions where sys.__stdout__ has been detached in test frameworks. #126.
- bugfix: off-by-one error in *get_location()*, now accounts for `%i` in cursor_report, #94.
- bugfix *split_seqs()* and related functions failed to match when the color index was greater than 15, #101.
- bugfix: Context Managers, *fullscreen()*, *hidden_cursor()*, and *keypad()* now flush the stream after writing their sequences.
- bugfix: `chr(127)`, `\x7f` has changed from keycode `term.DELETE` to the more common match, `term.BACKSPACE`, #115 by jwezel.
- bugfix: ensure *FormattingOtherString* may be pickled.
- bugfix: Use `UTF-8` for keyboard if input encoding cannot be determined.
- deprecated: the curses `move()` capability is no longer recommended, suggest to use *move_xy()*, which matches the return value of *get_location()*.
- deprecated: `superscript`, `subscript`, `shadow`, and `dim` are no longer "compoundable" with colors, such as in phrase `Terminal.blue_subscript('a')`. These attributes are not typically supported, anyway. Use Unicode text or 256 or 24-bit color codes instead.
- deprecated: additional key names, such as `KEY_TAB`, are no longer "injected" into the curses module namespace.
- bugfix: briefly tried calling `curses.setupterm()` with `os.devnull` as the file descriptor, reverted. #59.
- deprecated: *inkey()* no longer raises RuntimeError when *stream* is not a terminal, programs using *inkey()* to block indefinitely if a keyboard is not attached. #69.
- deprecated: using argument `_intr_continue` to method *kbhit()*, behavior is as though such value is always True since 1.9.
- bugfix: Now imports on 3.10+
- bugfix: Fix detection of shift+arrow keys when using tmux. #178.

- enhancement: Instantiate SequenceTextWrapper only once in *wrap()*. #184.

1.16

- introduced: Windows support?! PR #110 by avylove.

1.15

- enhancement: disable timing integration tests for keyboard routines.
- enhancement: Support python 3.7. PR #102.
- enhancement: Various fixes to test automation PR #108

1.14

- bugfix: *wrap()* misbehaved for text containing newlines, #74.
- bugfix: TypeError when using `PYTHONOPTIMIZE=2` environment variable, #84.
- bugfix: define `blessed.__version__` value, #92.
- bugfix: detect sequences `\x1b[0K` and `\x1b2K`, #95.

1.13

- enhancement: *split_seqs()* introduced, and 4x cost reduction in related sequence-aware functions, #29.
- deprecated: `blessed.sequences.measure_length` function superseded by *iter_parse()* if necessary.
- deprecated: warnings about "binary-packed capabilities" are no longer emitted on strange terminal types, making best effort.

1.12

- enhancement: *get_location()* returns the `(row, col)` position of the cursor at the time of call for attached terminal.
- enhancement: a keyboard now detected as *stdin* when `stream` is `sys.stderr`.

1.11

- enhancement: *inkey()* can return more quickly for combinations such as `Alt + Z` when `MetaSendsEscape` is enabled, #30.
- enhancement: *FormattingString* may now be nested, such as `t.red('red', t.underline('rum'))`, #61

1.10

- workaround: provide `sc` and `rc` for Terminals of `kind='ansi'`, repairing *location()* #44.
- bugfix: length of simple SGR reset sequence `\x1b[m` was not correctly determined on all terminal types, #45.
- deprecated: `_intr_continue` arguments introduced in 1.8 are now marked deprecated in 1.10: beginning with python 3.5, the default behavior is as though this argument is always True, PEP-475, blessed does the same.

1.9

- enhancement: *break_long_words* now supported by *Terminal.wrap()*
- Ignore `curses.error` message `'tparm() returned NULL'`: this occurs on win32 or other platforms using a limited curses implementation, such as PDCurses, where `curses.tparm()` is not implemented, or no terminal capability database is available.

- Context manager *keypad()* emits sequences that enable "application keys" such as the diagonal keys on the numpad. This is equivalent to `curses.window.keypad()`.
- bugfix: translate keypad application keys correctly.
- enhancement: no longer depend on the '2to3' tool for python 3 support.
- enhancement: allow `civis` and `cnorm` (*hide_cursor*, *normal_hide*) to work with terminal-type *ansi* by emulating support by proxy.
- enhancement: new public attribute: *kind*: the very same as given `Terminal.__init__.kind` keyword argument. Or, when not given, determined by and equivalent to the `TERM` Environment variable.

1.8

- enhancement: export keyboard-read function as public method `getch()`, so that it may be overridden by custom terminal implementers.
- enhancement: allow *inkey()* and *kbhit()* to return early when interrupted by signal by passing argument `_intr_continue=False`.
- enhancement: allow `hpa` and `vpa` (*move_x*, *move_y*) to work on tmux(1) or screen(1) by emulating support by proxy.
- enhancement: add *rstrip()* and *lstrip()*, strips both sequences and trailing or leading whitespace, respectively.
- enhancement: include wcwidth library support for *length()*: the printable width of many kinds of CJK (Chinese, Japanese, Korean) ideographs and various combining characters may now be determined.
- enhancement: better support for detecting the length or sequences of externally-generated *ecma-48* codes when using `xterm` or `aixterm`.
- bugfix: when `locale.getpreferredencoding()` returns empty string or an encoding that is not valid for `codecs.getincrementaldecoder`, fallback to ASCII and emit a warning.
- bugfix: ensure *FormattingString* and *ParameterizingString* may be pickled.
- bugfix: allow *~.inkey* and related to be called without a keyboard.
- **change**: `term.keyboard_fd` is set `None` if `stream` or `sys.stdout` is not a tty, making `term.inkey()`, `term.cbreak()`, `term.raw()`, no-op.
- bugfix: `\x1bOH` (KEY_HOME) was incorrectly mapped as KEY_LEFT.

1.7

- Forked github project erikrose/blessings to jquast/blessed, this project was previously known as **blessings** version 1.6 and prior.
- introduced: context manager *cbreak()*, which is equivalent to entering terminal state by `tty.setcbreak()` and returning on exit, as well as the lesser recommended *raw()*, pairing from `tty.setraw()`.
- introduced: *inkey()*, which will return one or more characters received by the keyboard as a unicode sequence, with additional attributes *code* and *name*. This allows application keys (such as the up arrow, or home key) to be detected. Optional value *timeout* allows for timed poll.
- introduced: *center()*, *rjust()*, *ljust()*, allowing text containing sequences to be aligned to detected horizontal screen width, or by *width* specified.
- introduced: *wrap()* method. Allows text containing sequences to be word-wrapped without breaking mid-sequence, honoring their printable width.
- introduced: *strip()*, strips all sequences *and* whitespace.

- introduced: *strip_seqs()* strip only sequences.
- introduced: *rstrip()* and *lstrip()* strips both sequences and trailing or leading whitespace, respectively.
- bugfix: cannot call curses.setupterm() more than once per process (from Terminal.__init__()): Previously, blessed pretended to support several instances of different Terminal *kind*, but was actually using the *kind* specified by the first instantiation of *Terminal*. A warning is now issued. Although this misbehavior is still allowed, a warnings.WarningMessage is now emitted to notify about subsequent terminal misbehavior.
- bugfix: resolved issue where *number_of_colors* fails when *does_styling* is False. Resolves issue where piping tests output would fail.
- bugfix: warn and set *does_styling* to False when the given *kind* is not found in the terminal capability database.
- bugfix: allow unsupported terminal capabilities to be callable just as supported capabilities, so that the return value of *color*(n) may be called on terminals without color capabilities.
- bugfix: for terminals without underline, such as vt220, term.underline('text') would emit 'text' + term.normal. Now it emits only 'text'.
- enhancement: some attributes are now properties, raise exceptions when assigned.
- enhancement: pypy is now a supported python platform implementation.
- enhancement: removed pokemon curses.error exceptions.
- enhancement: do not ignore curses.error exceptions, unhandled curses errors are legitimate errors and should be reported as a bug.
- enhancement: converted nose tests to pytest, merged travis and tox.
- enhancement: pytest fixtures, paired with a new @as_subprocess decorator are used to test a multitude of terminal types.
- enhancement: test accessories @as_subprocess resolves various issues with different terminal types that previously went untested.
- deprecation: python2.5 is no longer supported (as tox does not supported).

1.6

- Add *does_styling*. This takes force_styling into account and should replace most uses of *is_a_tty*.
- Make *is_a_tty* a read-only property like *does_styling*. Writing to it never would have done anything constructive.
- Add fullscreen`() and hidden_cursor() to the auto-generated docs.

1.5.1

- Clean up fabfile, removing the redundant test command.
- Add Travis support.
- Make python setup.py test work without spurious errors on 2.6.
- Work around a tox parsing bug in its config file.
- Make context managers clean up after themselves even if there's an exception (*Vitja Makarov #29 <https://github.com/erikrose/blessings/pull/29>*).
- Parameterizing a capability no longer crashes when there is no tty (<*Vitja Makarov #31 <https://github.com/erikrose/blessings/pull/31>*)

1.5

- Add syntactic sugar and documentation for `enter_fullscreen` and `exit_fullscreen`.
- Add context managers *fullscreen()* and *hidden_cursor()*.
- Now you can force a *Terminal* to never to emit styles by passing keyword argument `force_styling=None`.

1.4

- Add syntactic sugar for cursor visibility control and single-space-movement capabilities.
- Endorse the *location()* context manager for restoring cursor position after a series of manual movements.
- Fix a bug in which *location()* that wouldn't do anything when passed zeros.
- Allow tests to be run with `python setup.py test`.

1.3

- Added *number_of_colors*, which tells you how many colors the terminal supports.
- Made *color*(n) and *on_color*(n) callable to wrap a string, like the named colors can. Also, make them both fall back to the `setf` and `setb` capabilities (like the named colors do) if the termcap entries for `setaf` and `setab` are not available.
- Allowed *color* to act as an unparametrized string, not just a callable.
- Made *height* and *width* examine any passed-in stream before falling back to stdout (This rarely if ever affects actual behavior; it's mostly philosophical).
- Made caching simpler and slightly more efficient.
- Got rid of a reference cycle between *Terminal* and *FormattingString*.
- Updated docs to reflect that terminal addressing (as in `location()`) is 0-based.

1.2

- Added support for Python 3! We need 3.2.3 or greater, because the curses library couldn't decide whether to accept strs or bytes before that (https://bugs.python.org/issue10570).
- Everything that comes out of the library is now unicode. This lets us support Python 3 without making a mess of the code, and Python 2 should continue to work unless you were testing types (and badly). Please file a bug if this causes trouble for you.
- Changed to the MIT License for better world domination.
- Added Sphinx docs.

1.1

- Added nicely named attributes for colors.
- Introduced compound formatting.
- Added wrapper behavior for styling and colors.
- Let you force capabilities to be non-empty, even if the output stream is not a terminal.
- Added *is_a_tty* to determine whether the output stream is a terminal.
- Sugared the remaining interesting string capabilities.
- Allow *location()* to operate on just an x *or* y coordinate.

1.0

- Extracted Blessed from nose-progressive.

CHAPTER

EIGHTEEN

INDEXES

- genindex
- modindex

PYTHON MODULE INDEX

b

blessed.color, 101
blessed.colorspace, 104
blessed.dec_modes, 108
blessed.formatters, 116
blessed.keyboard, 122
blessed.line_editor, 135
blessed.mouse, 138
blessed.sequences, 140
blessed.terminal, 144

INDEX

Symbols

__call__() (*FormattingOtherString method*), 120
__call__() (*FormattingString method*), 120
__call__() (*NullCallableString method*), 120
__call__() (*ParameterizingProxyString method*), 119
__call__() (*ParameterizingString method*), 118
__getattr__() (*Terminal method*), 152
__new__() (*BracketedPasteEvent static method*), 133
__new__() (*FocusEvent static method*), 134
__new__() (*Keystroke static method*), 122
__new__() (*KittyKeyEvent static method*), 130
__new__() (*LegacyCSIKeyEvent static method*), 131
__new__() (*ModifyOtherKeysEvent static method*), 130
__new__() (*SyncEvent static method*), 134
_alt (*Keystroke property*), 125
_asdict() (*BracketedPasteEvent method*), 134
_asdict() (*FocusEvent method*), 134
_asdict() (*KittyKeyEvent method*), 130
_asdict() (*LegacyCSIKeyEvent method*), 131
_asdict() (*ModifyOtherKeysEvent method*), 130
_asdict() (*SyncEvent method*), 134
_build_alphanum_predicate() (*Keystroke method*), 126
_build_appkeys_predicate() (*Keystroke method*), 126
_build_known_capability_names() (*in module blessed.formatters*), 116
_caps_lock (*Keystroke property*), 125
_code (*Keystroke attribute*), 122
_ctrl (*Keystroke property*), 125
_field_defaults (*BracketedPasteEvent attribute*), 134
_field_defaults (*FocusEvent attribute*), 134
_field_defaults (*KittyKeyEvent attribute*), 130
_field_defaults (*LegacyCSIKeyEvent attribute*), 131
_field_defaults (*ModifyOtherKeysEvent attribute*), 130
_field_defaults (*SyncEvent attribute*), 134
_fields (*BracketedPasteEvent attribute*), 134
_fields (*FocusEvent attribute*), 134
_fields (*KittyKeyEvent attribute*), 130
_fields (*LegacyCSIKeyEvent attribute*), 131
_fields (*ModifyOtherKeysEvent attribute*), 130
_fields (*SyncEvent attribute*), 134
_get_alt_only_control_name() (*Keystroke method*), 123
_get_ascii_value() (*Keystroke method*), 127
_get_bracketed_paste_name() (*Keystroke method*), 123
_get_control_char_name() (*Keystroke method*), 123
_get_control_symbol() (*Keystroke method*), 123
_get_ctrl_sequence_value() (*Keystroke method*), 126
_get_escape_sequence_value() (*Keystroke method*), 126
_get_focus_event_name() (*Keystroke method*), 123
_get_keycode_by_name() (*Keystroke static method*), 126
_get_kitty_protocol_name() (*Keystroke method*), 123
_get_meta_escape_name() (*Keystroke method*), 123
_get_modified_keycode_name() (*Keystroke method*), 122
_get_mouse_event_name() (*Keystroke method*), 123
_get_plain_char_value() (*Keystroke method*), 126
_get_protocol_value() (*Keystroke method*), 126
_hyper (*Keystroke property*), 125
_infer_modifiers() (*Keystroke static method*), 122
_is_escape_sequence() (*Keystroke method*), 126
_make() (*BracketedPasteEvent class method*), 134
_make() (*FocusEvent class method*), 134
_make() (*KittyKeyEvent class method*), 130
_make() (*LegacyCSIKeyEvent class method*), 131
_make() (*ModifyOtherKeysEvent class method*), 131
_make() (*SyncEvent class method*), 134
_make_colors() (*in module blessed.formatters*), 116
_make_effective_bits() (*Keystroke method*), 126
_make_expected_bits() (*Keystroke static method*), 126
_match (*Keystroke attribute*), 122
_meta (*Keystroke property*), 125
_mode (*Keystroke attribute*), 122
_mode_values (*Keystroke property*), 128
_modifiers (*Keystroke attribute*), 122
_name (*Keystroke attribute*), 122

_num_lock (*Keystroke property*), 125
_parse_bracketed_paste() (*Keystroke method*), 129
_parse_focus() (*Keystroke method*), 129
_parse_resize() (*Keystroke method*), 129
_parse_text() (*SoftwareVersion static method*), 133
_replace() (*BracketedPasteEvent method*), 134
_replace() (*FocusEvent method*), 134
_replace() (*KittyKeyEvent method*), 130
_replace() (*LegacyCSIKeyEvent method*), 131
_replace() (*ModifyOtherKeysEvent method*), 131
_replace() (*SyncEvent method*), 134
_resolve_name() (*Keystroke method*), 123
_shift (*Keystroke property*), 125
_super (*Keystroke property*), 125
_tparm() (*in module blessed.formatters*), 118
_tparm_cached() (*in module blessed.formatters*), 118

A

add() (*LineHistory method*), 136
AH1 (*DecPrivateMode attribute*), 111
AH1 (*Terminal.DecPrivateMode attribute*), 146
ALT_MODIFIED_MOUSEWHEEL (*DecPrivateMode attribute*), 115
ALT_MODIFIED_MOUSEWHEEL (*Terminal.DecPrivateMode attribute*), 146
ALT_SCREEN_AND_SAVE_CLEAR (*DecPrivateMode attribute*), 114
ALT_SCREEN_AND_SAVE_CLEAR (*Terminal.DecPrivateMode attribute*), 146
ALT_SCREEN_BUFFER_SWITCH (*DecPrivateMode attribute*), 114
ALT_SCREEN_BUFFER_SWITCH (*Terminal.DecPrivateMode attribute*), 146
ALT_SCREEN_BUFFER_XTERM (*DecPrivateMode attribute*), 114
ALT_SCREEN_BUFFER_XTERM (*Terminal.DecPrivateMode attribute*), 146
ALT_SCROLL_XTERM (*DecPrivateMode attribute*), 113
ALT_SCROLL_XTERM (*Terminal.DecPrivateMode attribute*), 146
ALT_SENDS_ESC (*DecPrivateMode attribute*), 113
ALT_SENDS_ESC (*Terminal.DecPrivateMode attribute*), 146
ambiguous_width (*LineEditor attribute*), 137
ambiguous_width (*Terminal attribute*), 151
AMBIGUOUS_WIDTH_DOUBLE_WIDTH (*DecPrivateMode attribute*), 115
AMBIGUOUS_WIDTH_DOUBLE_WIDTH (*Terminal.DecPrivateMode attribute*), 146
AMBIGUOUS_WIDTH_REPORTING (*DecPrivateMode attribute*), 115
AMBIGUOUS_WIDTH_REPORTING (*Terminal.DecPrivateMode attribute*), 146
APPLICATION_ESCAPE_KEY (*DecPrivateMode attribute*), 115
APPLICATION_ESCAPE_KEY (*Terminal.DecPrivateMode attribute*), 146
APPLICATION_MOUSEWHEEL (*DecPrivateMode attribute*), 115
APPLICATION_MOUSEWHEEL (*Terminal.DecPrivateMode attribute*), 146
async_inkey() (*Terminal method*), 187

B

BA1 (*DecPrivateMode attribute*), 111
BA1 (*Terminal.DecPrivateMode attribute*), 146
BA2 (*DecPrivateMode attribute*), 111
BA2 (*Terminal.DecPrivateMode attribute*), 146
base_key (*KittyKeyEvent attribute*), 130
begin (*SyncEvent attribute*), 134
bell (*LineEditResult attribute*), 136
bg_sgr (*DisplayState attribute*), 136
bg_sgr (*LineEditor attribute*), 137
BIDI_ARROW_KEY_SWAPPING (*DecPrivateMode attribute*), 114
BIDI_ARROW_KEY_SWAPPING (*Terminal.DecPrivateMode attribute*), 147
BIDI_AUTODETECTION (*DecPrivateMode attribute*), 115
BIDI_AUTODETECTION (*Terminal.DecPrivateMode attribute*), 147
BIDI_CURRENT_LINE (*DecPrivateMode attribute*), 115
BIDI_CURRENT_LINE (*Terminal.DecPrivateMode attribute*), 147
blessed.color
    module, 101
blessed.colorspace
    module, 104
blessed.dec_modes
    module, 108
blessed.formatters
    module, 116
blessed.keyboard
    module, 122
blessed.line_editor
    module, 135
blessed.mouse
    module, 138
blessed.sequences
    module, 140
blessed.terminal
    module, 144
BLINKING_BAR (*Terminal.CursorShape attribute*), 145
BLINKING_BLOCK (*Terminal.CursorShape attribute*), 145
BLINKING_UNDERLINE (*Terminal.CursorShape attribute*), 145
BOLD_ITALIC_HIGH_INTENSITY (*DecPrivateMode attribute*), 113

BOLD_ITALIC_HIGH_INTENSITY (*Terminal.DecPrivateMode attribute*), 147
BRACKETED_PASTE (*DecPrivateMode attribute*), 114
BRACKETED_PASTE (*Terminal.DecPrivateMode attribute*), 147
bracketed_paste() (*Terminal method*), 170
BRACKETED_PASTE_MIME (*DecPrivateMode attribute*), 114
BRACKETED_PASTE_MIME (*Terminal.DecPrivateMode attribute*), 147
BracketedPasteEvent (*class in blessed.keyboard*), 133
button (*MouseEvent property*), 139

C

cbreak() (*Terminal method*), 186
center() (*Sequence method*), 141
center() (*Terminal method*), 181
changeable (*DecModeResponse property*), 109
changed (*LineEditResult attribute*), 136
CHARACTER_MAPPING_SERVICE (*DecPrivateMode attribute*), 115
CHARACTER_MAPPING_SERVICE (*Terminal.DecPrivateMode attribute*), 147
clear() (*LineEditor method*), 138
clip() (*Sequence method*), 141
clip() (*Terminal method*), 182
clipboard_copy() (*Terminal method*), 166
clipboard_paste() (*Terminal method*), 166
code (*Keystroke property*), 124
color (*Terminal property*), 176
color_distance_algorithm (*Terminal property*), 180
color_hex() (*Terminal method*), 177
COLOR_PALETTE_UPDATES (*DecPrivateMode attribute*), 114
COLOR_PALETTE_UPDATES (*Terminal.DecPrivateMode attribute*), 147
COLOR_RESET_OSC (*Terminal.CursorShape attribute*), 145
color_rgb() (*Terminal method*), 176
COLORS (*in module blessed.formatters*), 116
COMPOUNDABLES (*in module blessed.formatters*), 118
cpr_xy (*Keystroke property*), 128
cpr_yx (*Keystroke property*), 128
CURSES_KEYCODE_OVERRIDE_MIXIN (*in module blessed.keyboard*), 135
cursor (*DisplayState attribute*), 135
cursor_shape() (*Terminal method*), 174

D

DEC131TM (*DecPrivateMode attribute*), 111
DEC131TM (*Terminal.DecPrivateMode attribute*), 147
dec_modes_disabled() (*Terminal method*), 160
dec_modes_enabled() (*Terminal method*), 159
DECAAM (*DecPrivateMode attribute*), 112
DECAAM (*Terminal.DecPrivateMode attribute*), 147
DECANM (*DecPrivateMode attribute*), 110
DECANM (*Terminal.DecPrivateMode attribute*), 147
DECARM (*DecPrivateMode attribute*), 110
DECARM (*Terminal.DecPrivateMode attribute*), 147
DECARSM (*DecPrivateMode attribute*), 112
DECARSM (*Terminal.DecPrivateMode attribute*), 147
DECATCBM (*DecPrivateMode attribute*), 113
DECATCBM (*Terminal.DecPrivateMode attribute*), 147
DECATCUM (*DecPrivateMode attribute*), 113
DECATCUM (*Terminal.DecPrivateMode attribute*), 147
DECAWM (*DecPrivateMode attribute*), 110
DECAWM (*Terminal.DecPrivateMode attribute*), 147
DECBBSM (*DecPrivateMode attribute*), 113
DECBBSM (*Terminal.DecPrivateMode attribute*), 147
DECBCMM (*DecPrivateMode attribute*), 112
DECBCMM (*Terminal.DecPrivateMode attribute*), 147
DECBKM (*DecPrivateMode attribute*), 112
DECBKM (*Terminal.DecPrivateMode attribute*), 147
DECBPM (*DecPrivateMode attribute*), 112
DECBPM (*Terminal.DecPrivateMode attribute*), 147
DECBWRM (*DecPrivateMode attribute*), 111
DECBWRM (*Terminal.DecPrivateMode attribute*), 147
DECCANSM (*DecPrivateMode attribute*), 112
DECCANSM (*Terminal.DecPrivateMode attribute*), 147
DECCAPSLK (*DecPrivateMode attribute*), 113
DECCAPSLK (*Terminal.DecPrivateMode attribute*), 147
DECCKM (*DecPrivateMode attribute*), 110
DECCKM (*Terminal.DecPrivateMode attribute*), 147
DECCOLM (*DecPrivateMode attribute*), 110
DECCOLM (*Terminal.DecPrivateMode attribute*), 147
DECCRNLM (*DecPrivateMode attribute*), 111
DECCRNLM (*Terminal.DecPrivateMode attribute*), 147
DECCRTSM (*DecPrivateMode attribute*), 112
DECCRTSM (*Terminal.DecPrivateMode attribute*), 147
DECECM (*DecPrivateMode attribute*), 113
DECECM (*Terminal.DecPrivateMode attribute*), 147
DECEDM (*DecPrivateMode attribute*), 110
DECEDM (*Terminal.DecPrivateMode attribute*), 147
DECEKEM (*DecPrivateMode attribute*), 111
DECEKEM (*Terminal.DecPrivateMode attribute*), 147
DECESKM (*DecPrivateMode attribute*), 112
DECESKM (*Terminal.DecPrivateMode attribute*), 147
DECFPM (*DecPrivateMode attribute*), 112
DECFPM (*Terminal.DecPrivateMode attribute*), 148
DECFWM (*DecPrivateMode attribute*), 113
DECFWM (*Terminal.DecPrivateMode attribute*), 148
DECGEPM (*DecPrivateMode attribute*), 111
DECGEPM (*Terminal.DecPrivateMode attribute*), 148
DECGPBM (*DecPrivateMode attribute*), 111
DECGPBM (*Terminal.DecPrivateMode attribute*), 148
DECGPCM (*DecPrivateMode attribute*), 111
DECGPCM (*Terminal.DecPrivateMode attribute*), 148
DECGPCS (*DecPrivateMode attribute*), 111

DECGPCS (*Terminal.DecPrivateMode attribute*), 148
DECGRPM (*DecPrivateMode attribute*), 111
DECGRPM (*Terminal.DecPrivateMode attribute*), 148
DECHCCM (*DecPrivateMode attribute*), 112
DECHCCM (*Terminal.DecPrivateMode attribute*), 148
DECHDPXM (*DecPrivateMode attribute*), 112
DECHDPXM (*Terminal.DecPrivateMode attribute*), 148
DECHEBM (*DecPrivateMode attribute*), 111
DECHEBM (*Terminal.DecPrivateMode attribute*), 148
DECHEM (*DecPrivateMode attribute*), 111
DECHEM (*Terminal.DecPrivateMode attribute*), 148
DECHWUM (*DecPrivateMode attribute*), 113
DECHWUM (*Terminal.DecPrivateMode attribute*), 148
DECINLM (*DecPrivateMode attribute*), 110
DECINLM (*Terminal.DecPrivateMode attribute*), 148
DECIPEM (*DecPrivateMode attribute*), 112
DECIPEM (*Terminal.DecPrivateMode attribute*), 148
DECKANAM (*DecPrivateMode attribute*), 110
DECKANAM (*Terminal.DecPrivateMode attribute*), 148
DECKBUM (*DecPrivateMode attribute*), 112
DECKBUM (*Terminal.DecPrivateMode attribute*), 148
DECKKDM (*DecPrivateMode attribute*), 112
DECKKDM (*Terminal.DecPrivateMode attribute*), 148
DECKLHIM (*DecPrivateMode attribute*), 113
DECKLHIM (*Terminal.DecPrivateMode attribute*), 148
DECKPM (*DecPrivateMode attribute*), 112
DECKPM (*Terminal.DecPrivateMode attribute*), 148
DECLTM (*DecPrivateMode attribute*), 110
DECLTM (*Terminal.DecPrivateMode attribute*), 148
DECMCM (*DecPrivateMode attribute*), 112
DECMCM (*Terminal.DecPrivateMode attribute*), 148
DecModeResponse (*class in blessed.dec_modes*), 108
DECNAKB (*DecPrivateMode attribute*), 112
DECNAKB (*Terminal.DecPrivateMode attribute*), 148
DECNCSM (*DecPrivateMode attribute*), 112
DECNCSM (*Terminal.DecPrivateMode attribute*), 148
DECNKM (*DecPrivateMode attribute*), 112
DECNKM (*Terminal.DecPrivateMode attribute*), 148
DECNRCM (*DecPrivateMode attribute*), 111
DECNRCM (*Terminal.DecPrivateMode attribute*), 148
DECNULM (*DecPrivateMode attribute*), 112
DECNULM (*Terminal.DecPrivateMode attribute*), 148
DECNUMLK (*DecPrivateMode attribute*), 113
DECNUMLK (*Terminal.DecPrivateMode attribute*), 148
DECOM (*DecPrivateMode attribute*), 110
DECOM (*Terminal.DecPrivateMode attribute*), 148
DECOPM (*DecPrivateMode attribute*), 111
DECOPM (*Terminal.DecPrivateMode attribute*), 148
DECOSCNM (*DecPrivateMode attribute*), 112
DECOSCNM (*Terminal.DecPrivateMode attribute*), 148
DECPCCM (*DecPrivateMode attribute*), 112
DECPCCM (*Terminal.DecPrivateMode attribute*), 148
DECPEX (*DecPrivateMode attribute*), 111
DECPEX (*Terminal.DecPrivateMode attribute*), 149
DECPFF (*DecPrivateMode attribute*), 111
DECPFF (*Terminal.DecPrivateMode attribute*), 149
DecPrivateMode (*class in blessed.dec_modes*), 110
DECPSM (*DecPrivateMode attribute*), 111
DECPSM (*Terminal.DecPrivateMode attribute*), 149
DECPSP (*DecPrivateMode attribute*), 111
DECPSP (*Terminal.DecPrivateMode attribute*), 149
DECRLCM (*DecPrivateMode attribute*), 112
DECRLCM (*Terminal.DecPrivateMode attribute*), 149
DECRLM (*DecPrivateMode attribute*), 111
DECRLM (*Terminal.DecPrivateMode attribute*), 149
DECRPL (*DecPrivateMode attribute*), 113
DECRPL (*Terminal.DecPrivateMode attribute*), 149
DECSACE (*Terminal.Decrqss attribute*), 145
DECSASD (*Terminal.Decrqss attribute*), 145
DECSCA (*Terminal.Decrqss attribute*), 145
DECSCFDM (*DecPrivateMode attribute*), 111
DECSCFDM (*Terminal.DecPrivateMode attribute*), 149
DECSCL (*Terminal.Decrqss attribute*), 145
DECSCLM (*DecPrivateMode attribute*), 110
DECSCLM (*Terminal.DecPrivateMode attribute*), 149
DECSCNM (*DecPrivateMode attribute*), 110
DECSCNM (*Terminal.DecPrivateMode attribute*), 149
DECSCP (*Terminal.Decrqss attribute*), 145
DECSCPP (*Terminal.Decrqss attribute*), 145
DECSCS (*Terminal.Decrqss attribute*), 145
DECSCUSR (*Terminal.Decrqss attribute*), 145
DECSDDT (*Terminal.Decrqss attribute*), 145
DECSDM (*DecPrivateMode attribute*), 112
DECSDM (*Terminal.DecPrivateMode attribute*), 149
DECSDPT (*Terminal.Decrqss attribute*), 145
DECSFC (*Terminal.Decrqss attribute*), 145
DECSKCV (*Terminal.Decrqss attribute*), 145
DECSLCK (*Terminal.Decrqss attribute*), 146
DECSLPP (*Terminal.Decrqss attribute*), 146
DECSLRM (*Terminal.Decrqss attribute*), 146
DECSMBV (*Terminal.Decrqss attribute*), 146
DECSNLS (*Terminal.Decrqss attribute*), 146
DECSPP (*Terminal.Decrqss attribute*), 146
DECSPPCS (*Terminal.Decrqss attribute*), 146
DECSPRTT (*Terminal.Decrqss attribute*), 146
DECSRFR (*Terminal.Decrqss attribute*), 146
DECSSCLS (*Terminal.Decrqss attribute*), 146
DECSSDT (*Terminal.Decrqss attribute*), 146
DECSSL (*Terminal.Decrqss attribute*), 146
DECSTBM (*Terminal.Decrqss attribute*), 146
DECSTRL (*Terminal.Decrqss attribute*), 146
DECSWBV (*Terminal.Decrqss attribute*), 146
DECTCEM (*DecPrivateMode attribute*), 111
DECTCEM (*Terminal.DecPrivateMode attribute*), 149
DECTEK (*DecPrivateMode attribute*), 111
DECTEK (*Terminal.DecPrivateMode attribute*), 149
DECTEM (*DecPrivateMode attribute*), 111
DECTEM (*Terminal.DecPrivateMode attribute*), 149

DECTHAICM (*DecPrivateMode attribute*), 111
DECTHAICM (*Terminal.DecPrivateMode attribute*), 149
DECTHAIM (*DecPrivateMode attribute*), 111
DECTHAIM (*Terminal.DecPrivateMode attribute*), 149
DECTHAISCM (*DecPrivateMode attribute*), 112
DECTHAISCM (*Terminal.DecPrivateMode attribute*), 149
DECUPM (*DecPrivateMode attribute*), 111
DECUPM (*Terminal.DecPrivateMode attribute*), 149
DECVCCM (*DecPrivateMode attribute*), 112
DECVCCM (*Terminal.DecPrivateMode attribute*), 149
DECVSSM (*DecPrivateMode attribute*), 112
DECVSSM (*Terminal.DecPrivateMode attribute*), 149
DECXRLM (*DecPrivateMode attribute*), 112
DECXRLM (*Terminal.DecPrivateMode attribute*), 149
DEFAULT (*Terminal.CursorShape attribute*), 145
DEFAULT_SEQUENCE_MIXIN (*in module blessed.keyboard*), 135
DEFAULT_STYLE (*Terminal.CursorShape attribute*), 145
description (*DecModeResponse property*), 108
detect_ambiguous_width() (*Terminal method*), 157
detect_term_program() (*Terminal method*), 158
DeviceAttribute (*class in blessed.keyboard*), 132
disabled (*DecModeResponse property*), 109
disambiguate (*KittyKeyboardProtocol property*), 131
display (*LineEditor property*), 137
DisplayState (*class in blessed.line_editor*), 135
dist_cie2000() (*in module blessed.color*), 103
dist_cie76() (*in module blessed.color*), 102
dist_cie94() (*in module blessed.color*), 103
dist_rgb() (*in module blessed.color*), 102
dist_rgb_weighted() (*in module blessed.color*), 102
does_bracketed_paste() (*Terminal method*), 161
does_colored_underlines() (*Terminal method*), 169
does_decrqss() (*Terminal method*), 168
does_focus_events() (*Terminal method*), 162
does_grapheme_clustering() (*Terminal method*), 162
does_inband_resize() (*Terminal method*), 160
does_iterm2() (*Terminal method*), 164
does_iterm2_graphics() (*Terminal method*), 165
does_kitty_clipboard() (*Terminal method*), 165
does_kitty_graphics() (*Terminal method*), 164
does_kitty_notifications() (*Terminal method*), 165
does_kitty_pointer_shapes() (*Terminal method*), 166
does_kitty_query() (*Terminal method*), 167
does_mouse() (*Terminal method*), 160
does_osc52_clipboard() (*Terminal method*), 166
does_sixel() (*Terminal method*), 157
does_styled_underlines() (*Terminal method*), 169
does_styling (*Terminal property*), 152
does_synchronized_output() (*Terminal method*), 161
does_text_sizing() (*Terminal method*), 169
does_xtgettcap() (*Terminal method*), 163

E

ellipsis (*LineEditor attribute*), 137
ellipsis_sgr (*DisplayState attribute*), 136
ellipsis_sgr (*LineEditor attribute*), 137
enable_kitty_keyboard() (*Terminal method*), 173
enabled (*DecModeResponse property*), 109
entries (*LineHistory attribute*), 136
eof (*LineEditResult attribute*), 136
ESC_KEY_SENDS_BACKSLASH (*DecPrivateMode attribute*), 115
ESC_KEY_SENDS_BACKSLASH (*Terminal.DecPrivateMode attribute*), 149
event_type (*KittyKeyEvent attribute*), 130
event_type (*LegacyCSIKeyEvent attribute*), 131
EXTENDED_REVERSE_WRAPAROUND (*DecPrivateMode attribute*), 114
EXTENDED_REVERSE_WRAPAROUND (*Terminal.DecPrivateMode attribute*), 149

F

failed (*DecModeResponse property*), 109
FAST_SCROLL (*DecPrivateMode attribute*), 113
FAST_SCROLL (*Terminal.DecPrivateMode attribute*), 149
feed_key() (*LineEditor method*), 138
flushinp() (*Terminal method*), 186
focus_events() (*Terminal method*), 171
FOCUS_IN_OUT_EVENTS (*DecPrivateMode attribute*), 113
FOCUS_IN_OUT_EVENTS (*Terminal.DecPrivateMode attribute*), 149
FocusEvent (*class in blessed.keyboard*), 134
FONT_CHANGE_REPORTING (*DecPrivateMode attribute*), 115
FONT_CHANGE_REPORTING (*Terminal.DecPrivateMode attribute*), 149
formatter() (*Terminal method*), 178
FormattingOtherString (*class in blessed.formatters*), 120
FormattingString (*class in blessed.formatters*), 119
from_legacy_match() (*MouseEvent class method*), 140
from_match() (*DeviceAttribute class method*), 133
from_match() (*SoftwareVersion class method*), 133
from_sgr_match() (*MouseEvent class method*), 140
fullscreen() (*Terminal method*), 174

G

gained (*FocusEvent attribute*), 134
get_bgcolor() (*Terminal method*), 154
get_bgcolor_hex() (*Terminal method*), 155
get_cell_height_and_width() (*Terminal method*), 172

get_color_scheme() (*Terminal method*), 167
get_dec_mode() (*Terminal method*), 158
get_decrqss() (*Terminal method*), 168
get_device_attributes() (*Terminal method*), 156
get_fgcolor() (*Terminal method*), 154
get_fgcolor_hex() (*Terminal method*), 155
get_font_coverage() (*Terminal method*), 163
get_iterm2_capabilities() (*Terminal method*), 164
get_keyboard_codes() (*in module blessed.keyboard*), 129
get_keyboard_sequences() (*in module blessed.keyboard*), 129
get_kitty_keyboard_state() (*Terminal method*), 172
get_location() (*Terminal method*), 153
get_proxy_string() (*in module blessed.formatters*), 121
get_sixel_colors() (*Terminal method*), 172
get_sixel_height_and_width() (*Terminal method*), 171
get_software_version() (*Terminal method*), 156
get_xtgettcap() (*Terminal method*), 162
getch() (*Terminal method*), 185
GRAPHEME_CLUSTERING (*DecPrivateMode attribute*), 114
GRAPHEME_CLUSTERING (*Terminal.DecPrivateMode attribute*), 149
GRAPHICS_POSITION (*DecPrivateMode attribute*), 115
GRAPHICS_POSITION (*Terminal.DecPrivateMode attribute*), 149
GRAPHICS_POSITION_2 (*DecPrivateMode attribute*), 115
GRAPHICS_POSITION_2 (*Terminal.DecPrivateMode attribute*), 149

H

height (*Terminal property*), 152
hex_to_rgb() (*in module blessed.colorspace*), 107
hidden_cursor() (*Terminal method*), 174
history (*LineEditor property*), 137
HP_FUNC_KEY_MODE (*DecPrivateMode attribute*), 114
HP_FUNC_KEY_MODE (*Terminal.DecPrivateMode attribute*), 149

I

IN_BAND_WINDOW_RESIZE (*DecPrivateMode attribute*), 114
IN_BAND_WINDOW_RESIZE (*Terminal.DecPrivateMode attribute*), 150
inkey() (*Terminal method*), 187
INPUT_METHOD_EDITOR (*DecPrivateMode attribute*), 115
INPUT_METHOD_EDITOR (*Terminal.DecPrivateMode attribute*), 149
insert_text() (*LineEditor method*), 138
int_codepoints (*KittyKeyEvent attribute*), 130
interrupt (*LineEditResult attribute*), 136
is_a_tty (*Terminal property*), 152
is_sequence (*Keystroke property*), 122
iter_parse() (*in module blessed.sequences*), 143
ITERM2_REPORT_KEY_UP (*DecPrivateMode attribute*), 114
ITERM2_REPORT_KEY_UP (*Terminal.DecPrivateMode attribute*), 150

K

kbhit() (*Terminal method*), 185
KEEP_SELECTION_NO_HILITE (*DecPrivateMode attribute*), 113
KEEP_SELECTION_NO_HILITE (*Terminal.DecPrivateMode attribute*), 150
key (*ModifyOtherKeysEvent attribute*), 131
key_id (*LegacyCSIKeyEvent attribute*), 131
key_name (*Keystroke property*), 124
key_value (*Keystroke property*), 127
keymap (*LineEditor attribute*), 137
keypad() (*Terminal method*), 186
Keystroke (*class in blessed.keyboard*), 122
kind (*LegacyCSIKeyEvent attribute*), 131
kind (*Terminal property*), 152
KITTY_HANDLE_CTRL_C_Z (*DecPrivateMode attribute*), 115
KITTY_HANDLE_CTRL_C_Z (*Terminal.DecPrivateMode attribute*), 150
KittyKeyboardProtocol (*class in blessed.keyboard*), 131
KittyKeyEvent (*class in blessed.keyboard*), 130
KP_DELETE_SENDS_DEL (*DecPrivateMode attribute*), 113
KP_DELETE_SENDS_DEL (*Terminal.DecPrivateMode attribute*), 150

L

LEGACY_KBD_X11R6 (*DecPrivateMode attribute*), 114
LEGACY_KBD_X11R6 (*Terminal.DecPrivateMode attribute*), 150
LegacyCSIKeyEvent (*class in blessed.keyboard*), 131
length() (*Sequence method*), 142
length() (*Terminal method*), 183
limit (*LineEditor attribute*), 137
limit_bell (*LineEditor attribute*), 137
line (*LineEditor property*), 137
line (*LineEditResult attribute*), 136
LineEditor (*class in blessed.line_editor*), 137
LineEditResult (*class in blessed.line_editor*), 136
LineHistory (*class in blessed.line_editor*), 136
link() (*Terminal method*), 178
ljust() (*Sequence method*), 141

ljust() (*Terminal method*), 180
location() (*Terminal method*), 153
long_description (*DecPrivateMode property*), 115
long_description (*Terminal.DecPrivateMode property*), 151
lstrip() (*Sequence method*), 143
lstrip() (*Terminal method*), 183

M

make_arguments() (*KittyKeyboardProtocol method*), 132
max_width (*LineEditor attribute*), 137
measure_length() (*in module blessed.sequences*), 144
META_SENDS_ESC (*DecPrivateMode attribute*), 113
META_SENDS_ESC (*Terminal.DecPrivateMode attribute*), 150
META_SETS_EIGHTH_BIT (*DecPrivateMode attribute*), 113
META_SETS_EIGHTH_BIT (*Terminal.DecPrivateMode attribute*), 150
MINTTY_BIDI (*DecPrivateMode attribute*), 115
MINTTY_BIDI (*Terminal.DecPrivateMode attribute*), 150
MIRROR_BOX_DRAWING (*DecPrivateMode attribute*), 115
MIRROR_BOX_DRAWING (*Terminal.DecPrivateMode attribute*), 150
mode (*DecModeResponse property*), 108
mode (*Keystroke property*), 127
modifiers (*Keystroke property*), 124
modifiers (*KittyKeyEvent attribute*), 130
modifiers (*LegacyCSIKeyEvent attribute*), 131
modifiers (*ModifyOtherKeysEvent attribute*), 131
MODIFIERS_ALT_NUMLOCK (*DecPrivateMode attribute*), 113
MODIFIERS_ALT_NUMLOCK (*Terminal.DecPrivateMode attribute*), 150
modifiers_bits (*Keystroke property*), 125
ModifyOtherKeysEvent (*class in blessed.keyboard*), 130
module
    blessed.color, 101
    blessed.colorspace, 104
    blessed.dec_modes, 108
    blessed.formatters, 116
    blessed.keyboard, 122
    blessed.line_editor, 135
    blessed.mouse, 138
    blessed.sequences, 140
    blessed.terminal, 144
MOUSE_ALL_MOTION (*DecPrivateMode attribute*), 113
MOUSE_ALL_MOTION (*Terminal.DecPrivateMode attribute*), 150
mouse_enabled() (*Terminal method*), 170
MOUSE_EXTENDED_SGR (*DecPrivateMode attribute*), 113
MOUSE_EXTENDED_SGR (*Terminal.DecPrivateMode attribute*), 150
MOUSE_EXTENDED_UTF8 (*DecPrivateMode attribute*), 113
MOUSE_EXTENDED_UTF8 (*Terminal.DecPrivateMode attribute*), 150
MOUSE_HILITE_TRACKING (*DecPrivateMode attribute*), 113
MOUSE_HILITE_TRACKING (*Terminal.DecPrivateMode attribute*), 150
MOUSE_REPORT_CLICK (*DecPrivateMode attribute*), 113
MOUSE_REPORT_CLICK (*Terminal.DecPrivateMode attribute*), 150
MOUSE_REPORT_DRAG (*DecPrivateMode attribute*), 113
MOUSE_REPORT_DRAG (*Terminal.DecPrivateMode attribute*), 150
MOUSE_SGR_PIXELS (*DecPrivateMode attribute*), 113
MOUSE_SGR_PIXELS (*Terminal.DecPrivateMode attribute*), 150
MOUSE_URXVT (*DecPrivateMode attribute*), 113
MOUSE_URXVT (*Terminal.DecPrivateMode attribute*), 150
mouse_xy (*Keystroke property*), 128
mouse_yx (*Keystroke property*), 127
MouseEvent (*class in blessed.mouse*), 138
MouseLegacyEvent (*in module blessed.mouse*), 140
MouseSGREvent (*in module blessed.mouse*), 140
MOUSEWHEEL_REPORTING (*DecPrivateMode attribute*), 115
MOUSEWHEEL_REPORTING (*Terminal.DecPrivateMode attribute*), 150
move_down (*Terminal property*), 176
move_left (*Terminal property*), 176
move_right (*Terminal property*), 176
move_up (*Terminal property*), 176
move_xy() (*Terminal method*), 175
move_yx() (*Terminal method*), 176

N

name (*Keystroke property*), 123
nav_down() (*LineHistory method*), 136
nav_start() (*LineHistory method*), 136
nav_up() (*LineHistory method*), 136
no_line_wrap() (*Terminal method*), 175
NO_RESPONSE (*DecModeResponse attribute*), 108
normal (*Terminal property*), 178
NOT_QUERIED (*DecModeResponse attribute*), 108
NOT_RECOGNIZED (*DecModeResponse attribute*), 108
notify_on_resize() (*Terminal method*), 171
NullCallableString (*class in blessed.formatters*), 120
number_of_colors (*Terminal property*), 180

O

on_color (*Terminal property*), 177
on_color_hex() (*Terminal method*), 177

on_color_rgb() (*Terminal method*), 177
OV1 (*DecPrivateMode attribute*), 111
OV1 (*Terminal.DecPrivateMode attribute*), 150
overflow_left (*DisplayState attribute*), 135
overflow_right (*DisplayState attribute*), 136

P

padd() (*Sequence method*), 143
ParameterizingProxyString (*class in blessed.formatters*), 119
ParameterizingString (*class in blessed.formatters*), 118
PASSIVE_MOUSE_TRACKING (*DecPrivateMode attribute*), 114
PASSIVE_MOUSE_TRACKING (*Terminal.DecPrivateMode attribute*), 150
password_char (*LineEditor attribute*), 137
password_mode (*LineEditor property*), 137
permanent (*DecModeResponse property*), 109
PERMANENTLY_RESET (*DecModeResponse attribute*), 108
PERMANENTLY_SET (*DecModeResponse attribute*), 108
pixel_height (*Terminal property*), 152
pixel_width (*Terminal property*), 152
PK1 (*DecPrivateMode attribute*), 111
PK1 (*Terminal.DecPrivateMode attribute*), 150
pressed (*Keystroke property*), 125
progress_bar() (*Terminal method*), 179

R

RAISE_ON_CTRL_G (*DecPrivateMode attribute*), 114
RAISE_ON_CTRL_G (*Terminal.DecPrivateMode attribute*), 150
raw() (*Terminal method*), 186
RE_RESPONSE (*DeviceAttribute attribute*), 132
RE_RESPONSE (*SoftwareVersion attribute*), 133
RE_RESPONSE_CTERM (*DeviceAttribute attribute*), 132
READLINE_CHARACTER_QUOTING (*DecPrivateMode attribute*), 114
READLINE_CHARACTER_QUOTING (*Terminal.DecPrivateMode attribute*), 150
READLINE_MOUSE_BUTTON_1 (*DecPrivateMode attribute*), 114
READLINE_MOUSE_BUTTON_1 (*Terminal.DecPrivateMode attribute*), 150
READLINE_MOUSE_BUTTON_2 (*DecPrivateMode attribute*), 114
READLINE_MOUSE_BUTTON_2 (*Terminal.DecPrivateMode attribute*), 150
READLINE_MOUSE_BUTTON_3 (*DecPrivateMode attribute*), 114
READLINE_MOUSE_BUTTON_3 (*Terminal.DecPrivateMode attribute*), 150
READLINE_NEWLINE_PASTING (*DecPrivateMode attribute*), 114
READLINE_NEWLINE_PASTING (*Terminal.DecPrivateMode attribute*), 150
recognized (*DecModeResponse property*), 108
released (*Keystroke property*), 126
render() (*LineEditor method*), 137
render_backspace() (*LineEditor method*), 138
render_insert() (*LineEditor method*), 138
repeated (*Keystroke property*), 125
REPLACEMENT_CHAR_COLOR (*DecPrivateMode attribute*), 112
REPLACEMENT_CHAR_COLOR (*Terminal.DecPrivateMode attribute*), 150
report_all_keys (*KittyKeyboardProtocol property*), 132
report_alternates (*KittyKeyboardProtocol property*), 132
report_events (*KittyKeyboardProtocol property*), 131
REPORT_GRID_CELL_SELECTION (*DecPrivateMode attribute*), 114
REPORT_GRID_CELL_SELECTION (*Terminal.DecPrivateMode attribute*), 150
report_text (*KittyKeyboardProtocol property*), 132
RESET (*DecModeResponse attribute*), 108
resolve_attribute() (*in module blessed.formatters*), 121
resolve_capability() (*in module blessed.formatters*), 121
resolve_color() (*in module blessed.formatters*), 121
REUSE_CLIPBOARD_DATA (*DecPrivateMode attribute*), 114
REUSE_CLIPBOARD_DATA (*Terminal.DecPrivateMode attribute*), 151
REWRAP_ON_RESIZE_MINTTY (*DecPrivateMode attribute*), 115
REWRAP_ON_RESIZE_MINTTY (*Terminal.DecPrivateMode attribute*), 151
RGB_256TABLE (*in module blessed.colorspace*), 104
rgb_downconvert() (*Terminal method*), 178
rgb_to_hex() (*in module blessed.colorspace*), 107
rgb_to_lab() (*in module blessed.color*), 101
rgb_to_xyz() (*in module blessed.color*), 101
RGBColor (*class in blessed.colorspace*), 104
rjust() (*Sequence method*), 141
rjust() (*Terminal method*), 180
rstrip() (*Sequence method*), 143
rstrip() (*Terminal method*), 183

S

SAVE_CURSOR_DECSC (*DecPrivateMode attribute*), 114
SAVE_CURSOR_DECSC (*Terminal.DecPrivateMode attribute*), 151
SCO_FUNC_KEY_MODE (*DecPrivateMode attribute*), 114
SCO_FUNC_KEY_MODE (*Terminal.DecPrivateMode attribute*), 151

scroll_jump (*LineEditor attribute*), 137
SCROLL_MARKERS (*DecPrivateMode attribute*), 115
SCROLL_MARKERS (*Terminal.DecPrivateMode attribute*), 151
SCROLL_ON_KEYPRESS_RXVT (*DecPrivateMode attribute*), 113
SCROLL_ON_KEYPRESS_RXVT (*Terminal.DecPrivateMode attribute*), 151
SCROLL_ON_TTY_OUTPUT_RXVT (*DecPrivateMode attribute*), 113
SCROLL_ON_TTY_OUTPUT_RXVT (*Terminal.DecPrivateMode attribute*), 151
scroll_region() (*Terminal method*), 175
search_prefix() (*LineHistory method*), 136
Sequence (*class in blessed.sequences*), 140
sequence() (*Terminal.CursorShape static method*), 145
SET (*DecModeResponse attribute*), 108
set_password_mode() (*LineEditor method*), 138
set_window_title() (*Terminal method*), 179
SGR (*Terminal.Decrqss attribute*), 146
shifted_key (*KittyKeyEvent attribute*), 130
SHORTCUT_KEY_MODE (*DecPrivateMode attribute*), 115
SHORTCUT_KEY_MODE (*Terminal.DecPrivateMode attribute*), 151
SHOW_HIDE_SCROLLBAR (*DecPrivateMode attribute*), 115
SHOW_HIDE_SCROLLBAR (*Terminal.DecPrivateMode attribute*), 151
SHOW_SCROLLBAR_RXVT (*DecPrivateMode attribute*), 111
SHOW_SCROLLBAR_RXVT (*Terminal.DecPrivateMode attribute*), 151
SIXEL_PRIVATE_PALETTE (*DecPrivateMode attribute*), 114
SIXEL_PRIVATE_PALETTE (*Terminal.DecPrivateMode attribute*), 151
SIXEL_SCROLLING_LEAVES_CURSOR (*DecPrivateMode attribute*), 115
SIXEL_SCROLLING_LEAVES_CURSOR (*Terminal.DecPrivateMode attribute*), 151
SoftwareVersion (*class in blessed.keyboard*), 133
SpecialInternalKitty (*DecPrivateMode attribute*), 110
SpecialInternalKitty (*Terminal.DecPrivateMode attribute*), 151
SpecialInternalLegacyCSIModifier (*DecPrivateMode attribute*), 110
SpecialInternalLegacyCSIModifier (*Terminal.DecPrivateMode attribute*), 151
SpecialInternalModifyOtherKeys (*DecPrivateMode attribute*), 110
SpecialInternalModifyOtherKeys (*Terminal.DecPrivateMode attribute*), 151
split_compound() (*in module blessed.formatters*), 121
split_seqs() (*Terminal method*), 184
STEADY_BAR (*Terminal.CursorShape attribute*), 145
STEADY_BLOCK (*Terminal.CursorShape attribute*), 145
STEADY_UNDERLINE (*Terminal.CursorShape attribute*), 145
stream (*Terminal property*), 180
strip() (*Sequence method*), 142
strip() (*Terminal method*), 183
strip_seqs() (*Sequence method*), 143
strip_seqs() (*Terminal method*), 184
STYLES (*Terminal.CursorShape attribute*), 145
suggestion (*DisplayState attribute*), 135
suggestion_sgr (*DisplayState attribute*), 136
suggestion_sgr (*LineEditor attribute*), 137
SUN_FUNC_KEY_MODE (*DecPrivateMode attribute*), 114
SUN_FUNC_KEY_MODE (*Terminal.DecPrivateMode attribute*), 151
supported (*DecModeResponse property*), 109
supports_osc52 (*DeviceAttribute property*), 132
supports_sixel (*DeviceAttribute property*), 132
SyncEvent (*class in blessed.keyboard*), 134
SYNCHRONIZED_OUTPUT (*DecPrivateMode attribute*), 114
SYNCHRONIZED_OUTPUT (*Terminal.DecPrivateMode attribute*), 151
synchronized_output() (*Terminal method*), 170

T

term_program (*LineEditor attribute*), 137
term_program (*Terminal attribute*), 152
Terminal (*class in blessed.terminal*), 144
Terminal.CursorShape (*class in blessed.terminal*), 144
Terminal.DecPrivateMode (*class in blessed.terminal*), 146
Terminal.Decrqss (*class in blessed.terminal*), 145
TERMINFO_FUNC_KEY_MODE (*DecPrivateMode attribute*), 114
TERMINFO_FUNC_KEY_MODE (*Terminal.DecPrivateMode attribute*), 151
text (*BracketedPasteEvent attribute*), 134
text (*DisplayState attribute*), 135
text (*Keystroke property*), 128
TEXT_REFLOW (*DecPrivateMode attribute*), 114
TEXT_REFLOW (*Terminal.DecPrivateMode attribute*), 151
text_sgr (*DisplayState attribute*), 136
text_sgr (*LineEditor attribute*), 137
text_sized() (*Terminal method*), 181
to_dict() (*DecModeResponse method*), 110
truncate() (*Sequence method*), 142
truncate() (*Terminal method*), 182
TTCTH (*DecPrivateMode attribute*), 115
TTCTH (*Terminal.DecPrivateMode attribute*), 151

U

ungetch() (*Terminal method*), 185
unicode_key (*KittyKeyEvent attribute*), 130
URGENCY_ON_CTRL_G (*DecPrivateMode attribute*), 114
URGENCY_ON_CTRL_G (*Terminal.DecPrivateMode attribute*), 151
USE_CLIPBOARD_SELECTION (*DecPrivateMode attribute*), 113
USE_CLIPBOARD_SELECTION (*Terminal.DecPrivateMode attribute*), 151
uses_keyboard_protocol (*Keystroke property*), 125

V

value (*DecModeResponse property*), 108
value (*Keystroke property*), 127
VT220_KBD_EMULATION (*DecPrivateMode attribute*), 114
VT220_KBD_EMULATION (*Terminal.DecPrivateMode attribute*), 151

W

width (*Terminal property*), 152
WIN32_INPUT_MODE (*DecPrivateMode attribute*), 115
WIN32_INPUT_MODE (*Terminal.DecPrivateMode attribute*), 151
window_title() (*Terminal method*), 179
WINSZ (*class in blessed.terminal*), 188
wrap() (*Terminal method*), 184
ws_col (*WINSZ attribute*), 188
ws_row (*WINSZ attribute*), 188
ws_xpixel (*WINSZ attribute*), 188
ws_ypixel (*WINSZ attribute*), 188
WY_52_LINE (*DecPrivateMode attribute*), 112
WY_52_LINE (*Terminal.DecPrivateMode attribute*), 151
WYENAT_OFF (*DecPrivateMode attribute*), 112
WYENAT_OFF (*Terminal.DecPrivateMode attribute*), 151

X

X11_COLORNAMES_TO_RGB (*in module blessed.colorspace*), 105
xparse_color() (*in module blessed.colorspace*), 107
xterm256color_from_rgb() (*in module blessed.color*), 103
xterm256gray_from_rgb() (*in module blessed.color*), 103
xtgettcap (*Keystroke property*), 128
xyz_to_lab() (*in module blessed.color*), 101