Blessed Documentation

Release 1.17.12

Erik Rose, Jeff Quast, Avram Lubkin

Dec 05, 2020

CONTENTS

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, from our repository.

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

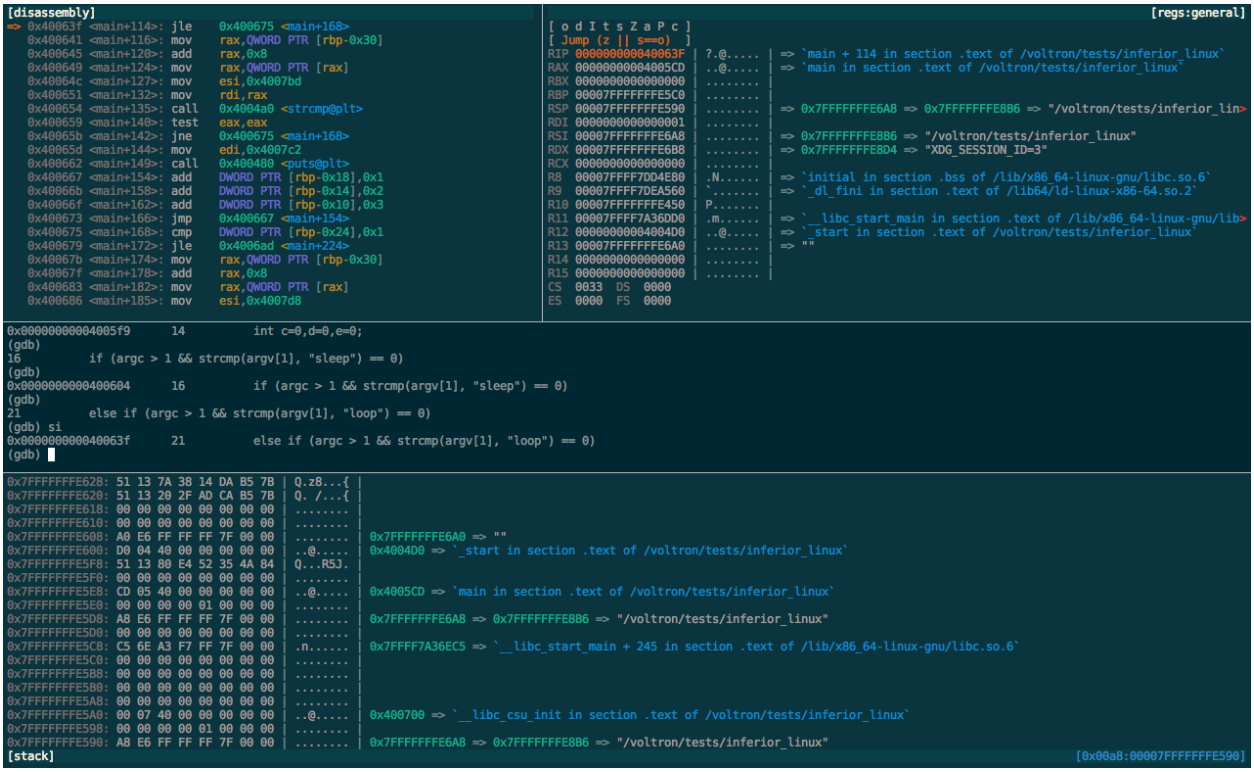


Fig. 2: Voltron is an extensible debugger UI toolkit written in Python

Fig. 3: cursewords is "graphical" command line program for solving crossword puzzles in the terminal.

Fig. 4: GitHeat builds an interactive heatmap of git history.

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.

Fig. 7: macht is a clone of the (briefly popular) puzzle game, 2048.

1.2 Requirements

Blessed works with Windows, Mac, Linux, and BSD's, on Python 2.7, 3.4, 3.5, 3.6, 3.7, and 3.8.

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().

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()):
    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('This is {under}underlined{normal}!'
      .format(under=underline, normal=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') + '!')
```

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 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!

2.3 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)
```

2.4 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.

2.5 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.

reverse Switch fore and background attributes.

normal Reset attributes to default.

underline Enable underline mode.

no_underline Disable underline mode.

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.

2.6 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()
```

2.7 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%"))
```

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.'))
```

3.1 24-bit Colors

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

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

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

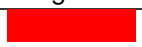
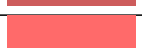
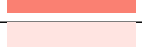
```
>>> term.number_of_colors = 1 << 24
>>> term.darkolivegreen
'\x1b[38;2;85;107;47m'
```

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

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

And finally, the direct `(r, g, b)` values of 0-255 can be used for *color_rgb()* and *on_color_rgb()* for foreground and background colors, to access each and every color!

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%
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%
webmaroon		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%
rosybrown1		100.0%	75.7%	75.7%	0.0%	24.3%	100.0%
rosybrown2		93.3%	70.6%	70.6%	0.0%	24.4%	93.3%
rosybrown3		80.4%	60.8%	60.8%	0.0%	24.4%	80.4%
rosybrown4		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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
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%
lightsalmon		100.0%	62.7%	47.8%	4.8%	52.2%	100.0%
lightsalmon2		93.3%	58.4%	44.7%	4.7%	52.1%	93.3%
lightsalmon3		80.4%	50.6%	38.4%	4.8%	52.2%	80.4%
lightsalmon4		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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
sandybrown		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%
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%
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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
darkseagreen1		75.7%	100.0%	75.7%	33.3%	24.3%	100.0%
darkseagreen2		70.6%	93.3%	70.6%	33.3%	24.4%	93.3%
darkseagreen3		60.8%	80.4%	60.8%	33.3%	24.4%	80.4%
darkseagreen4		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%
mediumseagreen		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%
springgreen		0.0%	100.0%	49.8%	41.6%	100.0%	100.0%
springgreen2		0.0%	93.3%	46.3%	41.6%	100.0%	93.3%
springgreen3		0.0%	80.4%	40.0%	41.6%	100.0%	80.4%
springgreen4		0.0%	54.5%	27.1%	41.6%	100.0%	54.5%
mediumspringgreen		0.0%	98.0%	60.4%	43.6%	100.0%	98.0%
aquamarine		49.8%	100.0%	83.1%	44.4%	50.2%	100.0%
aquamarine2		46.3%	93.3%	77.6%	44.4%	50.4%	93.3%
aquamarine3		40.0%	80.4%	66.7%	44.3%	50.2%	80.4%
aquamarine4		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%
lightseagreen		12.5%	69.8%	66.7%	49.1%	82.0%	69.8%
mediumturquoise		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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
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%
darkslategray		18.4%	31.0%	31.0%	50.0%	40.5%	31.0%
darkslategray1		59.2%	100.0%	100.0%	50.0%	40.8%	100.0%
darkslategray2		55.3%	93.3%	93.3%	50.0%	40.8%	93.3%
darkslategray3		47.5%	80.4%	80.4%	50.0%	41.0%	80.4%
darkslategray4		32.2%	54.5%	54.5%	50.0%	41.0%	54.5%
darkturquoise		0.0%	80.8%	82.0%	50.2%	100.0%	82.0%
paleturquoise		68.6%	93.3%	93.3%	50.0%	26.5%	93.3%
paleturquoise1		73.3%	100.0%	100.0%	50.0%	26.7%	100.0%
paleturquoise2		68.2%	93.3%	93.3%	50.0%	26.9%	93.3%
paleturquoise3		58.8%	80.4%	80.4%	50.0%	26.8%	80.4%
paleturquoise4		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%
powderblue		69.0%	87.8%	90.2%	51.9%	23.5%	90.2%
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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
lightblue3		60.4%	75.3%	80.4%	54.2%	24.9%	80.4%
deepskyblue		0.0%	74.9%	100.0%	54.2%	100.0%	100.0%
deepskyblue2		0.0%	69.8%	93.3%	54.2%	100.0%	93.3%
deepskyblue3		0.0%	60.4%	80.4%	54.1%	100.0%	80.4%
deepskyblue4		0.0%	40.8%	54.5%	54.2%	100.0%	54.5%
lightskyblue3		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%
lightskyblue		52.9%	80.8%	98.0%	56.4%	46.0%	98.0%
lightskyblue1		69.0%	88.6%	100.0%	56.1%	31.0%	100.0%
lightskyblue2		64.3%	82.7%	93.3%	56.1%	31.1%	93.3%
lightskyblue4		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%
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%
lightslategray		46.7%	53.3%	60.0%	58.3%	22.2%	60.0%
lightsteelblue		69.0%	76.9%	87.1%	59.4%	20.7%	87.1%
lightsteelblue1		79.2%	88.2%	100.0%	59.4%	20.8%	100.0%
lightsteelblue2		73.7%	82.4%	93.3%	59.3%	21.0%	93.3%
lightsteelblue3		63.5%	71.0%	80.4%	59.3%	21.0%	80.4%
lightsteelblue4		43.1%	48.2%	54.5%	59.2%	20.9%	54.5%
cornflowerblue		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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
midnightblue		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%
lightslateblue		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%
darkslateblue		28.2%	23.9%	54.5%	69.0%	56.1%	54.5%
mediumslateblue		48.2%	40.8%	93.3%	69.0%	56.3%	93.3%
mediumpurple		57.6%	43.9%	85.9%	72.1%	48.9%	85.9%
mediumpurple1		67.1%	51.0%	100.0%	72.1%	49.0%	100.0%
mediumpurple2		62.4%	47.5%	93.3%	72.1%	49.2%	93.3%
mediumpurple3		53.7%	40.8%	80.4%	72.1%	49.3%	80.4%
mediumpurple4		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%
rebeccapurple		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%
darkorchid1		74.9%	24.3%	100.0%	77.8%	75.7%	100.0%
darkorchid2		69.8%	22.7%	93.3%	77.8%	75.6%	93.3%
darkorchid3		60.4%	19.6%	80.4%	77.8%	75.6%	80.4%
darkorchid4		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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
mediumorchid1		87.8%	40.0%	100.0%	80.0%	60.0%	100.0%
mediumorchid2		82.0%	37.3%	93.3%	80.0%	60.1%	93.3%
mediumorchid3		70.6%	32.2%	80.4%	79.9%	60.0%	80.4%
mediumorchid4		47.8%	21.6%	54.5%	80.0%	60.4%	54.5%
mediumorchid		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%
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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
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%

continues on next page

Table 1 – continued from previous page

Name	Image	R	G	B	H	S	V
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%
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%

3.2 256 Colors

The built-in capability *color()* accepts a numeric index of any value between 0 and 254, I guess you could call this "Color by number…", it not recommended, there are many common cases where the colors do not match across terminals!

3.3 16 Colors

Recommended for common CLI applications.

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#3/4_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.

3.4 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.

CHAPTER

FOUR

KEYBOARD

The built-in function `input()` (or `raw_input()`) is pretty good for a basic game:

```
name = input("What is your name? ")
if sum(map(ord, name)) % 2:
    print(f"{name}?! What a beautiful name!")
else:
    print(f"How interesting, {name} you say?")
```

But it has drawbacks – it's no good for interactive apps! This function **will not return until the return key is pressed**, so we can't do any exciting animations, and we can't understand or detect arrow keys and others required to make awesome, interactive apps and games!

Blessed fixes this issue with a context manager, `cbreak()`, and a single function for all keyboard input, *`inkey()`*.

4.1 inkey()

Let's just dive right into a rich "event loop", that awaits a keypress for 3 seconds and tells us what key we pressed.

```
print(f"{term.home}{term.black_on_skyblue}{term.clear}")
print("press 'q' to quit.")
with term.cbreak():
    val = ''
    while val.lower() != 'q':
        val = term.inkey(timeout=3)
        if not val:
            print("It sure is quiet in here ...")
        elif val.is_sequence:
            print("got sequence: {0}.".format((str(val), val.name, val.code)))
        elif val:
            print("got {0}.".format(val))
    print(f'bye!{term.normal}')
```

`cbreak()` enters a special mode that ensures `os.read()` on an input stream will return as soon as input is available, as explained in cbreak(3). This mode is combined with *`inkey()`* to decode multibyte sequences, such as `\0x1bOA`, into a unicode-derived *`Keystroke`* instance.

The *`Keystroke`* returned by *`inkey()`* is unicode – it may be printed, joined with, or compared to any other unicode strings. It also has these special attributes:

- *`is_sequence`* (bool): Whether it is an "application" key.
- *`code`* (int): The keycode, for equality testing.

- *name* (str): a human-readable name of any "application" key.

4.2 Keycodes

When the *is_sequence* property tests *True*, the value of *code* represents a unique application key of the keyboard.

code may then be compared with attributes of *Terminal*, which are duplicated from those 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_C1	351	
KEY_C3	352	
KEY_CANCEL	355	
KEY_CATAB	342	
KEY_CENTER	350	
KEY_CLEAR	333	
KEY_CLOSE	356	
KEY_COMMAND	357	
KEY_COPY	358	
KEY_CREATE	359	
KEY_CTAB	341	
KEY_DELETE	330	'\x1b[3~'
KEY_DL	328	
KEY_DOWN	258	'\x1b[B', '\x1b[OB'
KEY_EIC	332	
KEY_END	360	'\x1b[F', '\x1b[K', '\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'
KEY_F10	274	
KEY_F11	275	
KEY_F12	276	
KEY_F13	277	
KEY_F14	278	
KEY_F15	279	
KEY_F16	280	
KEY_F17	281	
KEY_F18	282	
KEY_F19	283	
KEY_F2	266	'\x1bOQ'
KEY_F20	284	
KEY_F21	285	
KEY_F22	286	

continues on next page

Table 1 – continued from previous page

Name	Value	Example Sequence(s)
KEY_F23	287	
KEY_F3	267	‘\x1bOR’
KEY_F4	268	‘\x1bOS’
KEY_F5	269	
KEY_F6	270	
KEY_F7	271	
KEY_F8	272	
KEY_F9	273	
KEY_FIND	362	‘\x1b[1~’
KEY_HELP	363	
KEY_HOME	262	‘\x1b[H’, ‘\x1b[7~’, ‘\x1b[OH’
KEY_IL	329	
KEY_INSERT	331	‘\x1b[2~’
KEY_KP_0	520	‘\x1bOp’
KEY_KP_1	521	‘\x1bOq’
KEY_KP_2	522	‘\x1bOr’
KEY_KP_3	523	‘\x1bOs’
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_DECIMAL	517	‘\x1bOn’
KEY_KP_DIVIDE	518	‘\x1bOo’
KEY_KP_EQUAL	519	‘\x1bOX’
KEY_KP_MULTIPLY	513	‘\x1bOj’
KEY_KP_SEPARATOR	515	‘\x1bOl’
KEY_KP_SUBTRACT	516	‘\x1bOm’
KEY_LEFT	260	‘\x1b[D’, ‘\x1b[OD’
KEY_LL	347	
KEY_MARK	364	
KEY_MAX	511	
KEY_MESSAGE	365	
KEY_MIN	257	
KEY_MOUSE	409	
KEY_MOVE	366	
KEY_NEXT	367	
KEY_OPEN	368	
KEY_OPTIONS	369	
KEY_PGDOWN	338	‘\x1b[U’, ‘\x1b[6~’
KEY_PGUP	339	‘\x1b[V’, ‘\x1b[5~’
KEY_PREVIOUS	370	
KEY_PRINT	346	
KEY_REDO	371	
KEY_REFERENCE	372	
KEY_REFRESH	373	
KEY_REPLACE	374	

continues on next page

Table 1 – continued from previous page

Name	Value	Example Sequence(s)
KEY_RESET	345	
KEY_RESIZE	410	
KEY_RESTART	375	
KEY_RESUME	376	
KEY_RIGHT	261	'\x1b[C', '\x1b[OC'
KEY_SAVE	377	
KEY_SBEG	378	
KEY_SCANCEL	379	
KEY_SCOMMAND	380	
KEY_SCOPY	381	
KEY_SCREATE	382	
KEY_SDC	383	
KEY_SDL	384	
KEY_SDOWN	336	'\x1b[1;2B'
KEY_SELECT	385	'\x1b[4~'
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	
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	

All such keystrokes can be decoded by blessed, there is a demonstration program, *keymatrix.py* that tests how many of them you can find !

4.3 delete

Typically, backspace is `^H` (8, or 0x08) and delete is ^? (127, or 0x7f).

On some systems however, the key for backspace is actually labeled and transmitted as "delete", though its function in the operating system behaves just as backspace. Blessed usually returns "backspace" in most situations.

It is highly recommend to accept **both** `KEY_DELETE` and `KEY_BACKSPACE` as having the same meaning except when implementing full screen editors, and provide a choice to enable the delete mode by configuration.

4.4 Alt/meta

Programs with GNU readline, like bash, have *Alt* combinators, such as *ALT+u* to uppercase the word after cursor. This is achieved by the configuration option altSendsEscape or metaSendsEscape in xterm.

The default for most terminals, however, is for this key to be bound by the operating system, or, used for inserting international keys, (where the combination *ALT+u, a* is used to insert the character `ä`).

It is therefore a recommendation to **avoid alt or meta keys entirely** in applications.

And instead prefer the ctrl-key combinations, maybe along with *raw()*, to avoid instructing users to custom-configure their terminal emulators to communicate *Alt* sequences.

If you still wish to optionall decode them, *ALT+z* becomes *Escape* + *z* (or, in raw form `\x1bz`). This is detected by blessings as two keystrokes, `KEY_ESCAPE` and `'z'`. Blessings currently provides no further assistance in detecting these key combinations.

CHAPTER

FIVE

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

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` or is often valued as *\n*, which additionally returns the carriage to column 0, and, depending on your terminal emulator, may also destroy any characters to end of line.

`move_down(1)` is always a safe non-destructive one-notch movement in the downward direction.

`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.

5.1 Example

The source code of *bounce.py* combines a small bit of *Keyboard* 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)
```

5.2 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.

5.3 Finding The Cursor

We can determine the cursor's current position at anytime using *get_location()*.

This uses a kind of "answer back" sequence that your terminal emulator responds to. Because the terminal may not respond, or may take some time to respond, the *timeout* keyword argument can be specified to return coordinates (-1, -1) after a blocking timeout:

```
>>> term.get_location(timeout=5)
(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!

CHAPTER

SIX

MEASURING

Any string containing sequences can be measured by blessed using the *length()* method. This means that blessed can measure, right-align, center, or word-wrap its own output!

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

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

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

```
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)))
```

6.1 Resizing

To detect when the size of the window changes, you can author a callback for SIGWINCH signals:

```
import signal
from blessed import Terminal

term = Terminal()

def on_resize(sig, action):
    print(f'height={term.height}, width={term.width}')
```

(continues on next page)

(continued from previous page)

```
signal.signal(signal.SIGWINCH, on_resize)

# wait for keypress
term.inkey()
```

Note: This is not compatible with Windows! We hope to make a cross-platform API for this in the future https://github.com/jquast/blessed/issues/131.

Sometimes it is necessary to make sense of sequences, and to distinguish them from plain text. The *split_seqs()* method can allow us 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']
```

Will display something like, `['\x1b[1m', 'b', 'b', 'q', '\x1b(B', '\x1b[m']`

Method *strip_seqs()* can remove all sequences from a string:

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

CHAPTER

SEVEN

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*.

7.1 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.

7.2 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.

7.3 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 of the connecting client is capable of rendering multibyte characters as a single cell.

7.4 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.

7.5 keymatrix.py

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

This program displays a "gameboard" of all known special KEY_NAME constants. When the key is depressed, it is highlighted, as well as displaying the unicode sequence, integer code, and friendly-name of any key pressed.

7.6 on_resize.py

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

This program installs a SIGWINCH signal handler, which detects screen resizes while also polling for input, displaying keypresses.

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

7.7 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

7.8 progress_bar.py

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

This program demonstrates a simple progress bar. All text is written to stderr, to avoid the need to "flush" or emit newlines, and makes use of the move_x (hpa) capability to "overstrike" the display a scrolling progress bar.

7.9 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.

7.10 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.

7.11 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.

7.12 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!

CHAPTER

EIGHT

API DOCUMENTATION

8.1 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*, *green*, *blue*)
Convert standard RGB color to XYZ color.

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

D65/2° standard illuminant

xyz_to_lab (*x_val*, *y_val*, *z_val*)
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*, *green*, *blue*)
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*, *rgb2*)

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*, *rgb2*)

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*, *rgb2*)

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_cie94(*rgb1*, *rgb2*)

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*, *rgb2*)
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

8.2 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.eu/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 = (RGBColor(red=0, green=0, blue=0), RGBColor(red=205, green=0, blue=0), RGBC
Curses color indices of 8, 16, and 256-color terminals

X11_COLORNAMES_TO_RGB = {'aliceblue': RGBColor(red=240, green=248, blue=255), 'antiquewhit
X11 Color names to (XTerm-defined) RGB values from xorg-rgb/rgb.txt

8.3 formatters.py

Sub-module providing sequence-formatting functions.

_make_colors()
Return set of valid colors and their derivatives.

Return type set

Returns Color names with prefixes

COLORS = {'aliceblue', 'antiquewhite', 'antiquewhite1', 'antiquewhite2', 'antiquewhite3',
Valid colors and their background (on), bright, and bright-background derivatives.

COMPOUNDABLES = {'blink', 'bold', 'italic', 'reverse', 'standout', 'underline'}
Attributes that may be compounded with colors, by underscore, such as 'reverse_indigo'.

class ParameterizingString(*cap*, *normal=''*, *name='<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')
u'\x1b[91mcolor #9\x1b(B\x1b[m'
```

Class constructor accepting 3 positional arguments.

Parameters

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

__call__(**args*)
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
- **curses.error** – `curses.tparm()` raised an exception

Return type *FormattingString* or *NullCallableString*

Returns Callable string for given parameters

class ParameterizingProxyString(*fmt_pair*, *normal=''*, *name='<not specified>'*)
A Unicode string which can be called to proxy missing termcap entries.

This class supports the function *get_proxy_string()*, and 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)
u''
>>> fmt = u'\x1b[{0}G'
>>> fmt_arg = lambda *arg: (arg[0] + 1,)
>>> hpa = ParameterizingProxyString((fmt, fmt_arg), term.normal, 'hpa')
>>> hpa(9)
u'\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*)

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*, *normal=''*)

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))
u'\x1b[94m'
>>> style('Big Blue')
u'\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*)

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*, *target*)
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))
u'\x1b[C'
>>> print(repr(move_right(666)))
u'\x1b[666C'
>>> print(repr(move_right()))
u'\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*)
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*)
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*, *attr*)
Proxy and return callable string for proxied attributes.

Parameters

- **term** (`Terminal`) – *Terminal* instance.
- **attr** (*str*) – terminal capability name that may be proxied.

Return type None or *ParameterizingProxyString*.

Returns *ParameterizingProxyString* for some attributes of some terminal types that support it, where the terminfo(5) database would otherwise come up empty, such as `move_x` attribute for `term.kind` of `screen`. Otherwise, None.

split_compound(*compound*)
Split compound formating 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 formating string segments

resolve_capability(*term*, *attr*)
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 (u") if not found or not supported by the given *kind*.

Return type str

resolve_color(*term*, *color*)
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*, *attr*)
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*

8.4 keyboard.py

Sub-module providing 'keyboard awareness'.

class Keystroke(*ucs=''*, *code=None*, *name=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`.

When the string is a known sequence, *code* matches terminal class attributes for comparison, such as `term.KEY_LEFT`.

The string-name of the sequence, such as `u'KEY_LEFT'` is accessed by property *name*, and is used by the `__repr__()` method to display a human-readable form of the Keystroke this class instance represents. It may otherwise by joined, split, or evaluated just as as any other unicode string.

Class constructor.

static __new__(*cls*, *ucs=''*, *code=None*, *name=None*)
Class constructor.

property is_sequence
Whether the value represents a multibyte sequence (bool).

property name
String-name of key sequence, such as `u'KEY_LEFT'` (str).

property code
Integer keycode value of multibyte sequence (int).

get_keyboard_codes()
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, `u'KEY_LEFT'`.

These keys are derived from the attributes by the same of the curses 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*)
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 `u'\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.

_alternative_left_right(*term*)

Determine and return mapping of left and right arrow keys sequences.

Parameters **term** (*blessed.Terminal*) – *Terminal* instance.

Return type dict

Returns Dictionary of sequences `term._cuf1`, and `term._cub1`, valued as `KEY_RIGHT`, `KEY_LEFT` (when appropriate).

This function supports `get_terminal_sequences()` to discover the preferred input sequence for the left and right application keys.

It is necessary to check the value of these sequences to ensure we do not use `u' '` and `u'\b'` for `KEY_RIGHT` and `KEY_LEFT`, preferring their true application key sequence, instead.

DEFAULT_SEQUENCE_MIXIN = (('\n', 343), ('\r', 343), ('\x08', 263), ('\t', 512), ('\x1b', 3

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', 33

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.

_CURSES_KEYCODE_ADDINS = ('TAB', 'KP_MULTIPLY', 'KP_ADD', 'KP_SEPARATOR', 'KP_SUBTRACT', 'K

Though we may determine *keynames* and codes for keyboard input that generate multibyte sequences, it is also especially useful to aliases a few basic ASCII characters such as `KEY_TAB` instead of `u'\t'` for uniformity.

Furthermore, many key-names for application keys enabled only by context manager *keypad()* are surprisingly absent. We inject them here directly into the curses module.

8.5 sequences.py

Module providing 'sequence awareness'.

class Sequence(*sequence_text*, *term*)

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()*.

Class constructor.

Parameters

- **sequence_text** (*str*) – A string that may contain sequences.
- **term** (*blessed.Terminal*) – *Terminal* instance.

ljust(*width*, *fillchar=' '*)

Return string containing sequences, left-adjusted.

Parameters

- **width** (*int*) – Total width given to left-adjust `text`. If unspecified, the width of the attached terminal is used (default).
- **fillchar** (*str*) – String for padding right-of `text`.

Returns String of `text`, left-aligned by `width`.

Return type str

rjust(*width*, *fillchar=' '*)

Return string containing sequences, right-adjusted.

Parameters

- **width** (*int*) – Total width given to right-adjust `text`. If unspecified, the width of the attached terminal is used (default).
- **fillchar** (*str*) – String for padding left-of `text`.

Returns String of `text`, right-aligned by `width`.

Return type str

center(*width*, *fillchar=' '*)

Return string containing sequences, centered.

Parameters

- **width** (*int*) – Total width given to center `text`. If unspecified, the width of the attached terminal is used (default).
- **fillchar** (*str*) – String for padding left and right-of `text`.

Returns String of `text`, centered by `width`.

Return type str

length()

Return the printable length of string containing sequences.

Strings containing `term.left` or `\b` will cause "overstrike", but a length less than 0 is not ever returned. So _\b+ is a length of 1 (displays as +), but `\b` alone is simply a length of 0.

Some characters may consume more than one cell, mainly those CJK Unified Ideographs (Chinese, Japanese, Korean) defined by Unicode as half or full-width characters.

For example:

```
>>> from blessed import Terminal
>>> from blessed.sequences import Sequence
>>> term = Terminal()
>>> msg = term.clear + term.red(u'')
>>> 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=None*)
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=None*)
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=None*)
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()
Return `text` stripped of only its terminal sequences.

Return type str

Returns Text with terminal sequences removed

padd(*strip=False*)
Return non-destructive horizontal movement as destructive spacing.

Parameters **strip** (*bool*) – Strip terminal sequences

Return type str

Returns Text adjusted for horizontal movement

class SequenceTextWrapper(*width*, *term*, ***kwargs*)
Object for wrapping/filling text. The public interface consists of the wrap() and fill() methods; the other methods are just there for subclasses to override in order to tweak the default behaviour. If you want to completely replace the main wrapping algorithm, you'll probably have to override _wrap_chunks().

Several instance attributes control various aspects of wrapping:

width (default: 70) the maximum width of wrapped lines (unless break_long_words is false)

initial_indent (default: “”) string that will be prepended to the first line of wrapped output. Counts towards the line’s width.

subsequent_indent (default: “”) string that will be prepended to all lines save the first of wrapped output; also counts towards each line’s width.

expand_tabs (default: true) Expand tabs in input text to spaces before further processing. Each tab will become 0 .. ‘tabsize’ spaces, depending on its position in its line. If false, each tab is treated as a single character.

tabsize (default: 8) Expand tabs in input text to 0 .. ‘tabsize’ spaces, unless ‘expand_tabs’ is false.

replace_whitespace (default: true) Replace all whitespace characters in the input text by spaces after tab expansion. Note that if expand_tabs is false and replace_whitespace is true, every tab will be converted to a single space!

fix_sentence_endings (default: false) Ensure that sentence-ending punctuation is always followed by two spaces. Off by default because the algorithm is (unavoidably) imperfect.

break_long_words (default: true) Break words longer than ‘width’. If false, those words will not be broken, and some lines might be longer than ‘width’.

break_on_hyphens (default: true) Allow breaking hyphenated words. If true, wrapping will occur preferably on whitespaces and right after hyphens part of compound words.

drop_whitespace (default: true) Drop leading and trailing whitespace from lines.

max_lines (default: None) Truncate wrapped lines.

placeholder (default: ‘ [...]’) Append to the last line of truncated text.

Class initializer.

This class supports the *wrap()* method.

_wrap_chunks(*chunks*)
Sequence-aware variant of `textwrap.TextWrapper._wrap_chunks()`.

Raises **ValueError** – `self.width` is not a positive integer

Return type list

Returns text chunks adjusted for width

This simply ensures that word boundaries are not broken mid-sequence, as standard python textwrap would incorrectly determine the length of a string containing sequences, and may also break consider sequences part of a “word” that may be broken by hyphen (-), where this implementation corrects both.

_handle_long_word(*reversed_chunks*, *cur_line*, *cur_len*, *width*)
Sequence-aware `textwrap.TextWrapper._handle_long_word()`.

This simply ensures that word boundaries are not broken mid-sequence, as standard python textwrap would incorrectly determine the length of a string containing sequences, and may also break consider sequences part of a “word” that may be broken by hyphen (-), where this implementation corrects both.

iter_parse(*term*, *text*)
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.

measure_length(*text*, *term*)
Deprecated since version 1.12.0..

Return type int

Returns Length of the first sequence in the string

8.6 terminal.py

Module containing *Terminal*, the primary API entry point.

class Terminal(*kind=None*, *stream=None*, *force_styling=False*)

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 as taken by `curses.setupterm()`. Defaults to the value of the `TERM` environment variable.

  Note: Terminals withing a single process must share a common `kind`. See *_CUR_TERM*.

- **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.

__getattr__(*attr*)

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

Read-only property: Terminal kind determined on class initialization.

Return type str

property does_styling
Read-only property: Whether this class instance may emit sequences.

Return type bool

property is_a_tty
Read-only property: Whether *stream* is a terminal.

Return type bool

property height
Read-only property: Height of the terminal (in number of lines).

Return type int

property width
Read-only property: Width of the terminal (in number of columns).

Return type int

property pixel_height
Read-only property: Height ofthe terminal (in pixels).

Return type int

property pixel_width
Read-only property: Width of terminal (in pixels).

Return type int

location(*x=None*, *y=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 Compaibility with the blessings library, sorry for the trouble!

---

get_location(*timeout=None*)
Return tuple (row, column) of cursor position.

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)`. When a timeout is specified, always ensure the return value is checked for `(-1, -1)`.

The location of the cursor is determined by emitting the `u7` terminal capability, or VT100 Query Cursor Position when such capability is undefined, which elicits a response from a reply string described by capability `u6`, or again VT100's definition of `\x1b[%i%d;%dR` when undefined.

The `(y, x)` return value matches the parameter order of the *move_xy()* 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)
```

fullscreen()
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()
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.

move_xy(*x*, *y*)
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*, *x*)
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
Move cursor 1 cells to the left, or callable string for n>1 cells.

property move_right
Move cursor 1 or more cells to the right, or callable string for n>1 cells.

property move_up
Move cursor 1 or more cells upwards, or callable string for n>1 cells.

property move_down
Move cursor 1 or more cells downwards, or callable string for n>1 cells.

property color
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*, *green*, *blue*)
Provides callable formatting string to set foreground color to the specified RGB color.

Parameters

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

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*.

property on_color
A callable capability that sets the background color.

Return type *ParameterizingString*

on_color_rgb(*red*, *green*, *blue*)
Provides callable formatting string to set background color to the specified RGB color.

Parameters

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

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*.

formatter(*value*)

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*, *green*, *blue*)

Translate an RGB color to a color code of the terminal's color depth.

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

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*, *text*, *url_id=''*)

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`.

property stream
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
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.

property color_distance_algorithm
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'.

ljust (*text*, *width=None*, *fillchar=' '*)
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*, *width=None*, *fillchar=' '*)
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*, *width=None*, *fillchar=' '*)
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`

length(*text*)
Return printable length of a string containing sequences.

Parameters **text** (*str*) – String to measure. May contain terminal sequences.

Return type int

Returns The number of terminal character cells the string will occupy when printed

Wide characters that consume 2 character cells are supported:

```
>>> term = Terminal()
>>> term.length(term.clear + term.red(u''))
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*, *chars=None*)
Return `text` without sequences and leading or trailing whitespace.

Return type str

Returns Text with leading and trailing whitespace removed

```
>>> term.strip(u' \x1b[0;3m xyz ')
u'xyz'
```

rstrip(*text*, *chars=None*)
Return `text` without terminal sequences or trailing whitespace.

Return type str

Returns Text with terminal sequences and trailing whitespace removed

```
>>> term.rstrip(u' \x1b[0;3m xyz ')
u'  xyz'
```

lstrip(*text*, *chars=None*)
Return `text` without terminal sequences or leading whitespace.

Return type str

Returns Text with terminal sequences and leading whitespace removed

```
>>> term.lstrip(u' \x1b[0;3m xyz ')
u'xyz '
```

strip_seqs(*text*)
Return `text` stripped of only its terminal sequences.

Return type str

Returns Text with terminal sequences removed

```
>>> term.strip_seqs(u'\x1b[0;3mxyz')
u'xyz'
>>> term.strip_seqs(term.cuf(5) + term.red(u'test'))
u'     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*, ***kwds*)
Return `text` split by individual character elements and sequences.

Parameters

- **text** (*str*) – String containing sequences
- **kwds** – remaining keyword arguments for `re.split()`.

Return type list[str]

Returns List of sequences and individual characters

```
>>> term.split_seqs(term.underline(u'xyz'))
['\x1b[4m', 'x', 'y', 'z', '\x1b(B', '\x1b[m']
```

wrap(*text*, *width=None*, ***kwargs*)
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

See `textwrap.TextWrapper` for keyword arguments that can customize wrapping behaviour.

getch()
Read, decode, and return the next byte from the keyboard stream.

Return type unicode

Returns a single unicode character, or `u''` if a multi-byte sequence has not yet been fully received.

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.

Implementors of alternate input stream methods should override this method.

ungetch(*text*)
Buffer input data to be discovered by next call to *inkey()*.

Parameters **text** (*str*) – String to be buffered as keyboard input.

kbhit(*timeout=None*)
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()

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()

A context manager for `tty.setraw()`.

Although both `break()` and *raw()* modes allow each keystroke to be read immediately after it is pressed, Raw mode disables processing of input and output.

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 receive by the *inkey()* method.

Because output processing is not done, 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()

Context manager that enables directional keypad input.

On entrying, 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.

inkey(*timeout=None*, *esc_delay=0.35*)
Read and return the next keyboard event within given timeout.

Generally, this should be used inside the *raw()* 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*) – To distinguish between the keystroke of KEY_ESCAPE, and sequences beginning with escape, the parameter esc_delay specifies the amount of time after receiving escape (chr(27)) to seek for the completion of an application key before returning a *Keystroke* instance for KEY_ESCAPE.

Return type *Keystroke*.

Returns *Keystroke*, which may be empty (u'') 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!

---

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)

_CUR_TERM = None

CHAPTER

NINE

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!

9.1 Fork

Blessed is a fork of blessings. Apologies for the fork, I just couldn't get the *Keyboard*, and later *Location* or *Measuring* 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.

9.2 License

Blessed is under the MIT License. See the LICENSE file. Please enjoy!

9.3 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, you may for example use looponfailing with python 3.7, stopping at the first failing test case, and looping (retrying) after a filesystem save is detected:

```
tox -epy37 -- -fx
```

The test runner (`tox`) ensures all code and documentation complies with standard python style guides, pep8 and pep257, as well as various static analysis tools.

Warning: When you contribute a new feature, make sure it is covered by tests.

Likewise, some bug fixes should include a test demonstrating the bug.

9.4 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.

9.5 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

TEN

VERSION HISTORY

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 is 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

ELEVEN

INDEXES

- genindex
- modindex

PYTHON MODULE INDEX

b

INDEX

Symbols

B

C

D

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W

X