### Introduction
1.1 Brief Overview ........................................... 3
1.2 Before And After ....................................... 4
1.3 Requirements ............................................. 5
1.4 Further Documentation .................................. 5
1.5 Bugs, Contributing, Support ........................... 5
1.6 License .................................................. 5
1.7 Forked ................................................... 5

### Overview
2.1 Styling and Formatting ................................... 7
2.2 Moving The Cursor ....................................... 10
2.3 Height And Width ....................................... 11
2.4 Clearing The Screen ................................... 12
2.5 Full-Screen Mode ....................................... 12
2.6 Pipe Savvy .............................................. 13
2.7 Sequence Awareness ................................... 13
2.8 Keyboard Input .......................................... 14

### Examples
3.1 editor.py ............................................... 17
3.2 keymatrix.py ........................................... 17
3.3 on_resize.py ............................................ 17
3.4 progress_bar.py ....................................... 17
3.5 tprint.py ............................................... 18
3.6 worms.py ............................................... 18

### Further Reading
4.1 8 and 16 colors ......................................... 21
4.2 Where is brown, purple, or grey? .................... 21
4.3 Bold is bright ........................................... 22
4.4 Beware of customized color schemes ............... 23
4.5 Monochrome and reverse ............................... 24
4.6 Multibyte Encodings and Code pages .............. 24
4.7 Alt or meta sends Escape .............................. 25
4.8 Backspace sends delete ............................... 25
5.9 The misnomer of ANSI ................................................................. 25

6 API Documentation ................................................................. 27
6.1 terminal.py ................................................................. 27
6.2 formatters.py ............................................................... 33
6.3 keyboard.py ................................................................. 36
6.4 sequences.py ............................................................... 38

7 Contributing ................................................................. 43
7.1 Developing ................................................................. 43

8 Version History ................................................................. 45

9 Indexes ................................................................. 49

Python Module Index ................................................................. 51
Blessed is a thin, practical wrapper around terminal capabilities in Python.

Coding with Blessed looks like this...

```python
from blessed import Terminal

t = Terminal()

print(t.bold('Hi there!'))
print(t.bold_red_on_bright_green('It hurts my eyes!'))

with t.location(0, t.height - 1):
    print(t.center(t.blink('press any key to continue.')))

with t.cbreak():
    inp = t.inkey()
    print('You pressed ' + repr(inp))
```

### 1.1 Brief Overview

Blessed is a more simplified wrapper around curses, providing:

- Styles, color, and maybe a little positioning without necessarily clearing the whole screen first.
- Works great with standard Python string formatting.
- Provides up-to-the-moment 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: outputs to any file-like object such as StringIO, files, or pipes.
- Uses the terminfo(5) database so it works with any terminal type and supports any terminal capability: No more C-like calls to tigetstr and tparm.
- Keeps a minimum of internal state, so you can feel free to mix and match with calls to curses or whatever other terminal libraries you like.
- Provides plenty of context managers to safely express terminal modes, automatically restoring the terminal to a safe state on exit.
- Act intelligently when somebody redirects your output to a file, omitting all of the terminal sequences such as styling, colors, or positioning.
• Dead-simple keyboard handling: safely decoding unicode input in your system’s preferred locale and supports application/arrow keys.

• Allows the printable length of strings containing sequences to be determined.

Blessed does not provide...

• Windows command prompt support. A PDCurses build of python for windows provides only partial support at this time – there are plans to merge with the ansi module in concert with colorama to resolve this. Patches welcome!

1.2 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:

```python
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:

```python
from blessed import Terminal

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

Chapter 1. Introduction
1.3 Requirements

Blessed is compatible with Python 2.7, 3.4, and 3.5 on Debian Linux, Mac OSX, and FreeBSD.

1.4 Further Documentation

More documentation can be found at http://blessed.readthedocs.org/en/latest/

1.5 Bugs, Contributing, Support

Bugs or suggestions? Visit the issue tracker and file an issue. We welcome your bug reports and feature suggestions!

Would you like to contribute? That’s awesome! We’ve written a guide to help you.

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.

1.6 License

Blessed is under the MIT License. See the LICENSE file.

1.7 Forked

Blessed is a fork of blessings. Changes since 1.7 have all been proposed but unaccepted upstream.

Furthermore, a project in the node.js language of the same name is not related, or a fork of each other in any way.
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:

```python
>>> term = Terminal()
```

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

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

Its color support:

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

And use construct strings containing color and styling:

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

Furthermore, the special sequences inserted with application keys (arrow and function keys) are understood and decoded, as well as your locale-specific encoded multibyte input, such as utf-8 characters.

## 2.1 Styling and Formatting

Lots of handy formatting codes are available as attributes on a `Terminal` class instance. For example:

```python
from blessed import Terminal

term = Terminal()

print('I am ' + term.bold + 'bold' + term.normal + '!')
```

These capabilities (`bold`, `normal`) are translated to their sequences, which when displayed simply change the video attributes. And, when used as a callable, automatically wraps the given string with this sequence, and terminates it with `normal`.

The same can be written as:

```python
print('I am' + term.bold('bold') + '!
```

You may also use the `Terminal` instance as an argument for the `str.format`() method, so that capabilities can be displayed in-line for more complex strings:
2.1.1 Capabilities

The basic capabilities supported by most terminals are:

- **bold** Turn on ‘extra bright’ mode.
- **reverse** Switch fore and background attributes.
- **blink** Turn on blinking.
- **normal** Reset attributes to default.

The less commonly supported capabilities:

- **dim** Enable half-bright mode.
- **underline** Enable underline mode.
- **no_underline** Exit underline mode.
- **italic** Enable italicized text.
- **no_italic** Exit italics.
- **shadow** Enable shadow text mode (rare).
- **no_shadow** Exit shadow text mode.
- **standout** Enable standout mode (often, an alias for **reverse**).
- **no_standout** Exit standout mode.
- **subscript** Enable subscript mode.
- **no_subscript** Exit subscript mode.
- **superscript** Enable superscript mode.
- **no_superscript** Exit superscript mode.
- **flash** Visual bell, flashes the screen.

Note that, 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.

Many of these are aliases, their true capability names (such as ‘smul’ for ‘begin underline mode’) may still be used. Any capability in the terminfo(5) manual, under column **Cap-name**, may be used as an attribute of a **Terminal** instance. If it is not a supported capability, or a non-tty is used as an output stream, an empty string is returned.

2.1.2 Colors

Color terminals are capable of at least 8 basic colors.

- **black**
- **red**
- **green**
- **yellow**
- **blue**
• magenta
• cyan
• white

The same colors, prefixed with `bright_` (synonymous with `bold_`), such as `bright_blue`, provides 16 colors in total.

Prefixed with `on_`, the given color is used as the background color. Some terminals also provide an additional 8 high-intensity versions using `on_bright`, some example compound formats:

```python
from blessed import Terminal

term = Terminal()

print(term.on_bright_blue('Blue skies!'))

print(term.bright_red_on_bright_yellow('Pepperoni Pizza!'))
```

You may also specify the `color()` index by number, which should be within the bounds of value returned by `number_of_colors`:

```python
from blessed import Terminal

term = Terminal()

for idx in range(term.number_of_colors):
    print(term.color(idx)('Color {0}'.format(idx)))
```

You can check whether the terminal definition used supports colors, and how many, using the `number_of_colors` property, which returns any of 0, 8 or 256 for terminal types such as `vt220`, `ansi`, and `xterm-256color`, respectively.

### 2.1.3 Colorless Terminals

If the terminal defined by the Environment variable `TERM` does not support colors, these simply return empty strings. When used as a callable, the string passed as an argument is returned as-is. Most sequences emitted to a terminal that does not support them are usually harmless and have no effect.

Colorless terminals (such as the amber or green monochrome `vt220`) do not support colors but do support reverse video. For this reason, it may be desirable in some applications to simply select a foreground color, followed by reverse video to achieve the desired background color effect:

```python
from blessed import Terminal

term = Terminal()

print(term.green_reverse('some terminals standout more than others'))
```

Which appears as `black on green` on color terminals, but `black text on amber or green` on monochrome terminals. Whereas the more declarative formatter `black_on_green` would remain colorless.

**Note:** On most color terminals, `bright_black` is not invisible – it is actually a very dark shade of gray!

### 2.1.4 Compound Formatting

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

term = Terminal()

print(term.bold_underline_green_on_yellow('Woo'))

I’d be remiss if I didn’t credit couleur, where I probably got the idea for all this mashing.
This compound notation comes in handy if you want to allow users to customize formatting, just allow compound formatters, like bold_green, as a command line argument or configuration item:

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

parser = argparse.ArgumentParser(description='displays argument as specified style')
parser.add_argument('style', type=str, help='style formatter')
parser.add_argument('text', type=str, nargs='+')

term = Terminal()
args = parser.parse_args()
style = getattr(term, args.style)
print(style(' '.join(args.text)))
```

Saved as tprint.py, this could be used like:

$ ./tprint.py bright_blue_reverse Blue Skies

## 2.2 Moving The Cursor

When you want to move the cursor, you have a few choices:

- location(x=None, y=None) context manager.
- move(row, col) capability.
- move_y(row) capability.
- move_x(col) capability.

**Warning:** The location() method receives arguments in positional order (x, y), whereas the move() capability receives arguments in order (y, x). Please use keyword arguments as a later release may correct the argument order of location().

### 2.2.1 Moving Temporarily

A context manager, location() is provided to move the cursor to an (x, y) screen position and restore the previous position upon exit:

```python
from blessed import Terminal

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

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

Parameters to `location()` are the optional `x` and/or `y` keyword arguments:

```python
with term.location(y=10):
    print('We changed just the row.')
```

When omitted, it saves the cursor position and restore it upon exit:

```python
with term.location():
    print(term.move(1, 1) + 'Hi')
    print(term.move(9, 9) + 'Mom')
```

**Note:** calls to `location()` may not be nested.

### 2.2.2 Moving Permanently

If you just want to move and aren’t worried about returning, do something like this:

```python
from blessed import Terminal

term = Terminal()

print(term.move(10, 1) + 'Hi, mom!')
```

- **move** Position the cursor, parameter in form of `(y, x)`
- **move_x** Position the cursor at given horizontal column.
- **move_y** Position the cursor at given vertical column.

### 2.2.3 One-Notch Movement

Finally, there are some parameterless movement capabilities that move the cursor one character in various directions:

- `move_left`
- `move_right`
- `move_up`
- `move_down`

**Note:** `move_down` is often valued as `\n`, which additionally returns the carriage to column 0, depending on your terminal emulator, and may also destructively destroy any characters at the given position to the end of margin.

### 2.3 Height And Width

Use the `height` and `width` properties to determine the size of the window:
from blessed import Terminal

term = Terminal()
height, width = term.height, term.width
with term.location(x=term.width / 3, y=term.height / 3):
    print('1/3 ways in!')

These values are always current. To detect when the size of the window changes, you may author a callback for SIGWINCH signals:

```python
import signal
from blessed import Terminal

term = Terminal()

def on_resize(sig, action):
    print('height={t.height}, width={t.width}'.format(t=term))

signal.signal(signal.SIGWINCH, on_resize)

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

2.4 Clearing The Screen

Blessed provides syntactic sugar over some screen-clearing capabilities:

- **clear** Clear the whole screen.
- **clear_eol** Clear to the end of the line.
- **clear_bol** Clear backward to the beginning of the line.
- **clear_eos** Clear to the end of screen.

2.5 Full-Screen Mode

If you’ve ever noticed a program, such as an editor, restores the previous screen (such as your shell prompt) after exiting, you’re seeing the *enter_fullscreen* and *exit_fullscreen* attributes in effect.

- **enter_fullscreen** Switch to alternate screen, previous screen is stored by terminal driver.
- **exit_fullscreen** Switch back to standard screen, restoring the same terminal state.

There’s also a context manager you can use as a shortcut:

```python
from __future__ import division
from blessed import Terminal

term = Terminal()
with term.fullscreen():
    print(term.move_y(term.height // 2) + term.center('press any key').rstrip())
    term.inkey()
```
2.6 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. 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:

```python
from blessed import Terminal

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

2.7 Sequence Awareness

Blessed may measure the printable width of strings containing sequences, providing `center()`, `ljust()`, and `rjust()` methods, using the terminal screen’s width as the default `width` value:

```python
from __future__ import division
from blessed import Terminal

term = Terminal()
with term.location(y=term.height // 2):
    print(term.center(term.bold('bold and centered')))
```

Any string containing sequences may have its printable length measured using the `length()` method.

Additionally, a sequence-aware version of `textwrap.wrap()` is supplied as class as method `wrap()` that is also sequence-aware, so now you may word-wrap strings containing sequences. The following example displays a poem word-wrapped to 25 columns:

```python
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)))
```
2.8 Keyboard Input

The built-in python function `raw_input()` does not return a value until the return key is pressed, and is not suitable for detecting each individual keypress, much less arrow or function keys.

Furthermore, when calling `os.read()` on input stream, only bytes are received, which must be decoded to unicode using the locale-preferred encoding. Finally, multiple bytes may be emitted which must be paired with some verb like `KEY_LEFT`: blessed handles all of these special cases for you!

2.8.1 cbreak

The context manager `cbreak()` can be used to enter key-at-a-time mode: Any keypress by the user is immediately consumed by read calls:

```python
from blessed import Terminal
import sys

term = Terminal()

with term.cbreak():
    # block until any single key is pressed.
    sys.stdin.read(1)
```

The mode entered using `cbreak()` is called `cbreak(3)` in curses:

The cbreak routine disables line buffering and erase/kill character-processing (interrupt and flow control characters are unaffected), making characters typed by the user immediately available to the program.

`raw()` is similar to cbreak, but not recommended.

2.8.2 inkey

The method `inkey()` combined with `cbreak` completes the circle of providing key-at-a-time keyboard input with multibyte encoding and awareness of application keys.

`inkey()` resolves many issues with terminal input by returning a unicode-derived `Keystroke` instance. Its return value may be printed, joined with, or compared like any other unicode strings, it also provides the special attributes `is_sequence`, `code`, and `name`:

```python
from blessed import Terminal

term = Terminal()

print("press 'q' to quit.")
with term.cbreak():
    val = u''
    while val not in (u'q', u'Q',):
        val = term.inkey(timeout=5)
        if not val:
            # timeout
            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('bye!')
```
Its output might appear as:

got sequence: ('\x1b[A', 'KEY_UP', 259).
got sequence: ('\x1b[1;2A', 'KEY_SUP', 337).
got sequence: ('\x1b[17-', 'KEY_F6', 270).
got sequence: ('\x1b\', 'KEY_ESCAPE', 361).
got sequence: ('\n', 'KEY_ENTER', 343).
got /.
It sure is quiet in here ...
got sequence: ('\x1b0P', 'KEY_F1', 265).
It sure is quiet in here ...
got q.
bye!

A **timeout** value of *None* (default) will block forever until a keypress is received. Any other value specifies the length of time to poll for input: if no input is received after the given time has elapsed, an empty string is returned. A **timeout** value of 0 is non-blocking.

### 2.8.3 keyboard codes

When the **is_sequence** property tests **True**, the value is a special application key of the keyboard. The **code** attribute may then be compared with attributes of **Terminal**, which are duplicated from those found in **curs_getch(3)**, or those constants in **curses** beginning with phrase **KEY**.

Some of these mnemonics are shorthand or predate modern PC terms and are difficult to recall. The following helpful aliases are provided instead:

<table>
<thead>
<tr>
<th>blessed</th>
<th>curses</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>KEY_DELETE</code></td>
<td><code>KEY_DC</code></td>
<td><code>chr(127)</code></td>
</tr>
<tr>
<td><code>KEY_TAB</code></td>
<td></td>
<td><code>chr(9)</code></td>
</tr>
<tr>
<td><code>KEY_INSERT</code></td>
<td><code>KEY_IC</code></td>
<td></td>
</tr>
<tr>
<td><code>KEY_PGUP</code></td>
<td><code>KEY_PPAGE</code></td>
<td></td>
</tr>
<tr>
<td><code>KEY_PGDOWN</code></td>
<td><code>KEY_NPAGE</code></td>
<td></td>
</tr>
<tr>
<td><code>KEY_ESCAPE</code></td>
<td><code>KEY_EXIT</code></td>
<td></td>
</tr>
<tr>
<td><code>KEY_SUP</code></td>
<td><code>KEY_SR</code></td>
<td>(shift + up)</td>
</tr>
<tr>
<td><code>KEY_SDOWN</code></td>
<td><code>KEY_SF</code></td>
<td>(shift + down)</td>
</tr>
<tr>
<td><code>KEY_DOWN_LEFT</code></td>
<td><code>KEY_C1</code></td>
<td>(keypad lower-left)</td>
</tr>
<tr>
<td><code>KEY_UP_RIGHT</code></td>
<td><code>KEY_A1</code></td>
<td>(keypad upper-left)</td>
</tr>
<tr>
<td><code>KEY_DOWN_RIGHT</code></td>
<td><code>KEY_C3</code></td>
<td>(keypad lower-left)</td>
</tr>
<tr>
<td><code>KEY_UP_RIGHT</code></td>
<td><code>KEY_A3</code></td>
<td>(keypad lower-right)</td>
</tr>
<tr>
<td><code>KEY_CENTER</code></td>
<td><code>KEY_B2</code></td>
<td>(keypad center)</td>
</tr>
<tr>
<td><code>KEY_BEGIN</code></td>
<td><code>KEY_BEG</code></td>
<td></td>
</tr>
</tbody>
</table>

The **name** property will prefer these aliases over the built-in **curses** names.

The following are **not** available in the **curses** module, but are provided for keypad support, especially where the **keypad()** context manager is used with numlock on:

- `KEY_KP_MULTIPLY`
- `KEY_KP_ADD`
- `KEY_KP_SEPARATOR`
- `KEY_KP_SUBTRACT`
- `KEY_KP_DECIMAL`
- `KEY_KP_DIVIDE`
• `KEY_KP_0` through `KEY_KP_9`
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.

### 3.1 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, which demonstrates how to provide a line-editor rudimentary line-editor as well.

### 3.2 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.

### 3.3 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.

### 3.4 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.
3.5 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.

3.6 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 designed after the class game of WORMS.BAS, distributed with early Microsoft Q-BASIC for PC-DOS, and later more popularly known as “snake” as it was named on early mobile platforms.
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:

- 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.
- termios(4) 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!

These are often written in the C language, and directly map the “Control Sequence Inducers” (CSI, literally \x1b[] for most modern terminal types) emitted by most terminal capabilities to an action in a series of case

- Many modern libraries are now based on libvte (or just ‘vte‘): Gnome Terminal, sakura, Terminator, Lilyterm, ROXTerm, evilvte, TermIt, Termite, Tilda, tinyterm, lxterminal.
- xterm, urxvt, SyncTerm, and EtherTerm.
- There are far too many to name, Chose one you like!

- The source code of the tty(4), pty(4), 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 is linked here (the only kernel file common among them), it is probably not the most interesting, but it can get you started:

  - FreeBSD
  - OpenBSD
  - Illumos (Solaris)
  - Minix
The TTY driver is a great introduction to Kernel and Systems programming, because familiar components may be discovered and experimented with. It is available on all operating systems (except windows), and because of its critical nature, examples of efficient file I/O, character buffers (often implemented as “ring buffers”) and even fine-grained kernel locking can be found.

- 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.

- 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.
Growing Pains

When making terminal applications, there are a surprisingly number of portability issues and edge cases. Although Blessed provides an abstraction for the full curses capability database, it is not sufficient to secure you from several considerations shared here.

5.1 8 and 16 colors

Where 8 and 16 colors are used, they should be assumed to be the CGA Color Palette. Though there is no terminal standard that proclaims that the CGA colors are used, their values are the best approximations across all common hardware terminals and terminal emulators.

A recent phenomenon of users is to customize their base 16 colors to provide (often, more “washed out”) color schemes. Furthermore, we are only recently getting LCD displays of colorspace that achieve close approximation to the original video terminals. Some find these values uncomfortably intense: in their original CRT form, their contrast and brightness was lowered by hardware dials, whereas today’s LCD’s typically display well only near full intensity.

Though we may not detect the colorspace of the remote terminal, we can:

- Trust that a close approximation of the CGA Color Palette for the base 16 colors will be displayed for most users.
- Trust that users who have made the choice to adjust their palette have made the choice to do so, and are able to re-adjust such palettes as necessary to accommodate different programs (such as through the use of “Themes”).

Note: It has become popular to use dynamic system-wide color palette adjustments in software such as f.lux, which adjust the system-wide “Color Profile” of the entire graphics display depending on the time of day. One might assume that term.blue("text") may be completely invisible to such users during the night!

5.2 Where is brown, purple, or grey?

There are only 8 color names on a 16-color terminal: The second set of eight colors are “high intensity” versions of the first in direct series.

The colors brown, purple, and grey are not named in the first series, though they are available:

- brown: yellow is brown, only high-intensity yellow (bright_yellow) is yellow!
- purple: magenta is purple. In earlier, 4-bit color spaces, there were only black, cyan, magenta, and white of low and high intensity, such as found on common home computers like the ZX Spectrum.
Additional “colors” were only possible through dithering. The color names cyan and magenta on later graphics adapters are carried over from its predecessors. Although the color cyan remained true in RGB value on 16-color to its predecessor, magenta shifted farther towards blue from red becoming purple (as true red was introduced as one of the new base 8 colors).

- **grey**: there are actually **three shades of grey** (or American spelling, ‘gray’), though the color attribute named ‘grey’ does not exist!
  
  In ascending order of intensity, the shades of grey are:
  
  - **bold_black**: in lieu of the uselessness of an “intense black”, this is color is instead mapped to “dark grey”.
  - **white**: white is actually mild compared to the true color ‘white’: this is more officially mapped to “common grey”, and is often the default foreground color.
  - **bright_white**: is pure white (ffffff).

5.2.1 white-on-black

The default foreground and background should be assumed as white-on-black.

For quite some time, the families of terminals produced by DEC, IBM, and Tektronix dominated the computing world with the default color scheme of green-on-black and less commonly amber-on-black monochrome displays: The inverse was a non-default configuration. The IBM 3270 clients exclusively used green-on-black in both hardware and software emulators, and is likely a driving factor of the default white-on-black appearance of the first IBM Personal Computer.

The less common black-on-white “ink paper” style of emulators is a valid concern for those designing terminal interfaces. The color scheme of black-on-white directly conflicts with the intention of **bold is bright**, where `term.bright_red('ATTENTION!')` may become difficult to read, as it appears as *pink on white!*

History of ink-paper inspired black-on-white

Early home computers with color video adapters, such as the Commodore 64 provided white-on-blue as their basic video terminal configuration. One can only assume such appearances were provided to demonstrate their color capabilities over competitors (such as the Apple []).

More common, X11’s xterm and the software HyperTerm bundle with MS Windows provided an “ink on paper” black-on-white appearance as their default configuration. Two popular emulators continue to supply black-on-white by default to this day: Xorg’s xterm and Apple’s Terminal.app.

**Note:** Windows no longer supplies a terminal emulator: the “command prompt” as we know it now uses the MSVCRT API routines to interact and does not make use of terminal sequences, even ignoring those sequences that MS-DOS family of systems previously interpreted through the ANSI.SYS driver, though it continues to default to white-on-black.

5.3 Bold is bright

Where Bold is used, it should be assumed to be *Bright*.

Due to the influence of early graphics adapters providing a set of 8 “low-intensity” and 8 “high intensity” versions of the first, the term “bold” for terminals sequences is synonymous with “high intensity” in almost all circumstances.
5.3.1 History of bold as “wide stroke”

In typography, the true translation of “bold” is that a font should be displayed with emphasis. In classical terms, this would be achieved by pen re-writing over the same letters. On a teletype or printer, this was similarly achieved by writing a character, backspacing, then repeating the same character in a form called overstriking.

To bold a character, C, one would emit the sequence `C^H` where `^H` is backspace (0x08). To underline C, one would emit `C^H_`.

**Video terminals do not support overstriking.** Though the mdoc format for manual pages continue to emit overstriking sequences for bold and underline, translators such as mandoc will instead emit an appropriate terminal sequence.

Many characters previously displayable by combining using overstriking of ASCII characters on teletypes, such as: ±, , or were delegated to a code page or lost entirely until the introduction of multibyte encodings.

Much like the “ink paper” introduction in windowing systems for terminal emulators, “wide stroke” bold was introduced only much later when combined with operating systems that provided font routines such as TrueType.

5.3.2 Enforcing white-on-black

In conclusion, white-on-black should be considered the default. If there is a need to enforce white-on-black for terminal clients suspected to be defaulted as black-on-white, one would want to trust that a combination of `term.home + term.white_on_black + term.clear` should repaint the entire emulator’s window with the desired effect.

However, this cannot be trusted to all terminal emulators to perform correctly! Depending on your audience, you may instead ensure that the entire screen (including whitespace) is painted using the `on_black` mnemonic.

5.4 Beware of customized color schemes

A recent phenomenon is for users to customize these first 16 colors of their preferred emulator to colors of their own liking. Though this has always been possible with `~/.XResources`, the introduction of PuTTy and iTerm2 to interactively adjust these colors have made this much more common.

This may cause your audience to see your intended interface in a wildly different form. Your intended presentation may appear mildly unreadable.

Users are certainly free to customize their colors however they like, but it should be known that displaying `term.black_on_red("DANGER!")` may appear as “grey on pastel red” to your audience, reducing the intended effect of intensity.

5.4.1 256 colors can avoid customization

The first instinct of a user who aliases `ls(1)` to `ls -G` or `colorls`, when faced with the particularly low intensity of the default blue attribute is to adjust their terminal emulator’s color scheme of the base 16 colors.

This is not necessary: the environment variable `LSCOLORS` may be redefined to map an alternative color for blue, or to use `bright_blue` in its place.

Furthermore, all common terminal text editors such as emacs or vim may be configured with “colorschemes” to make use of the 256-color support found in most modern emulators. Many readable shades of blue are available, and many programs that emit such colors can be configured to emit a higher or lower intensity variant from the full 256 color space through program configuration.
5.5 Monochrome and reverse

Note that reverse takes the current foreground and background colors and reverses them. In contrast, the compound formatter `black_on_red` would fail to set the background or foreground color on a monochrome display, resulting in the same stylization as normal – it would not appear any different!

If your userbase consists of monochrome terminals, you may wish to provide “lightbars” and other such effects using the compound formatter `red_reverse`. In the literal sense of “set foreground color to red, then swap foreground and background”, this produces a similar effect on both color and monochrome displays.

For text, very few `{color}_on_{color}` formatters are visible with the base 16 colors, so you should generally wish for `black_on_{color}` anyway. By using `{color}_reverse` you may be portable with monochrome displays as well.

5.6 Multibyte Encodings and Code pages

A terminal that supports both multibyte encodings (UTF-8) and legacy 8-bit code pages (ISO 2022) may instruct the terminal to switch between both modes using the following sequences:

- `\x1b%G` activates UTF-8 with an unspecified implementation level from ISO 2022 in a way that allows to go back to ISO 2022 again.
- `\x1b%@` goes back from UTF-8 to ISO 2022 in case UTF-8 had been entered via `\x1b%G`.
- `\x1b%/G` switches to UTF-8 Level 1 with no return.
- `\x1b%/H` switches to UTF-8 Level 2 with no return.
- `\x1b%/I` switches to UTF-8 Level 3 with no return.

When a terminal is in ISO 2022 mode, you may use a sequence to request a terminal to change its code page. It begins by `\x1b(`, followed by an ASCII character representing a code page selection. For example `\x1b(U` on the legacy VGA Linux console switches to the IBM CP437 code page, allowing North American MS-DOS artwork to be displayed in its natural 8-bit byte encoding. A list of standard codes and the expected code page may be found on Thomas E. Dickey’s xterm control sequences section on sequences following the Control-Sequence-Inducer.

For more information, see What are the issues related to UTF-8 terminal emulators? by Markus Kuhn of the University of Cambridge.

5.6.1 Detecting multibyte

One can be assured that the connecting client is capable of representing UTF-8 and other multibyte character encodings by the Environment variable LANG. If this is not possible, there is an alternative method:

- Emit Report Cursor Position (CPR), `\x1b[6n` and store response.
- Emit a multibyte UTF-8 character, such as `(\x29\xb0).
- Emit Report Cursor Position (CPR), `\x1b[6n` and store response.
- Determine the difference of the (y, x) location of the response. If it is 1, then the client decoded the two UTF-8 bytes as a single character, and can be considered capable. If it is 2, the client is using a code page and is incapable of decoding a UTF-8 bytestream.

Note that both SSH and Telnet protocols provide means for forwarding the LANG environment variable. However, some transports such as a link by serial cable is incapable of forwarding Environment variables.
5.6.2 Detecting screen size

While we’re on the subject, there are times when height and width are not accurate – when a transport does not provide the means to propagate the COLUMNS and ROWS Environment values, or propagate the SIGWINCH signals, such as through a serial link.

The same means described above for multibyte encoding detection may be used to detect the remote client’s window size:

- Move cursor to row 999, 999.
- Emit Report Cursor Position (CPR), \x1b[6n and store response.
- The return value is the window dimensions of the client.

This is the method used by the program resize provided in the Xorg distribution, and its source may be viewed as file resize.c.

5.7 Alt or meta sends Escape

Programs using GNU readline such as bash continue to provide default mappings such as ALT+u to uppercase the word after cursor. This is achieved by the configuration option altSendsEscape or metaSendsEscape.

The default for most terminals, however, is that the meta key is bound by the operating system (such as META + F for find), and that ALT is 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, so as to avoid instructing your users to configure their terminal emulators to communicate such sequences.

If you wish to allow them optionally (such as through readline), the ability to detect alt or meta key combinations is achieved by prefacing the combining character with escape, so that ALT+z becomes Escape + z (or, in raw form \x1b\z). Blessings currently provides no further assistance in detecting these key combinations.

5.8 Backspace sends 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.

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.

5.9 The misnomer of ANSI

When people say ‘ANSI Sequence’, 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).
6.1 terminal.py

This module contains `Terminal`, the primary API entry point.

```python
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 within 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` 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:
```
For a parametrized capability such as `move` (or `cup`), pass the parameters as positional arguments:

```python
>>> term.move(line, column)
```

See the manual page `terminfo(5)` for a complete list of capabilities and their arguments.

```python
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](http://www.unixwiz.net/techtips/termios-vmin-vtime.html)

```python
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`

```python
color(num)
```

A callable string that sets the foreground color.

**Parameters**

- `num (int)` – The foreground color index. This should be within the bounds of `number_of_colors`.

**Return type** `ParameterizingString`

The capability is unparameterized until called and passed a number, 0-15, 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.

```python
does_styling
```

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

**Return type** `bool`
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:

```python
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.

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 is supports 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.

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

**Return type** int

hidden_cursor()  
Context manager that hides the cursor, setting visibility on exit.

```python
with term.hidden_cursor():
    main()
```

**Note:** `hidden_cursor()` calls cannot be nested: only one should be entered at a time.

inkey(timeout=None, esc_delay=0.35, **kwargs)  
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.

**Raises** RuntimeError When stream is not a terminal, having no keyboard attached, a timeout value of None would block indefinitely, prevented by by raising an exception.
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!

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

Return type bool

**kbhit**(timeout=None, **_kwargs)
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.

**keypad()**
Return a 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[HF, 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.

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

Return type str

**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.

**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

**location** *(x=None, y=None)*

Return a context manager for temporarily moving the cursor.

Move the cursor to a certain position on entry, let you print stuff there, then return the cursor to its original position:

```python
term = Terminal()
with term.location(2, 5):
    for x in xrange(10):
        print('I can do it %i times!' % x)
print('We\'re back to the 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.

**Note:** The store- and restore-cursor capabilities used internally provide no stack. This means that `location()` calls cannot be nested: only one should be entered at a time.

**lstrip** *(text, chars=None)*

Return `text` without terminal sequences or leading whitespace.

**Return type** str

```python
>>> term = blessed.Terminal()
>>> term.lstrip(u'\x1b[0;3m XXX ')
u'XXX '
```

**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.

**number_of_colors**

Read-only property: number of colors supported by terminal.

Common values are 0, 8, 16, 88, and 256.

Most commonly, this may be used to test whether the terminal supports colors. Though the underlying capability returns -1 when there is no color support, we return 0. This lets you test more Pythonically:

```python
if term.number_of_colors:
    ...
```

**on_color**

A callable capability that sets the background color.

**Parameters** num *(int)* – The background color index.

**Return type** ParameterizingString

6.1. terminal.py
**raw()**
A context manager for `tty.setraw()`.

Raw mode differs from `cbreak()` mode in that input and output processing of characters is disabled, in similar in that they both allow each keystroke to be read immediately after it is pressed.

For input, the interrupt, quit, suspend, and flow control characters are received as their raw control character values rather than generating a signal.

For output, the newline `chr(10)` is not sufficient enough to return the carriage, requiring `chr(13)` printed explicitly by your program:

```python
with term.raw():
    print("printing in raw mode", end="\r\n")
```

**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

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

**Return type** str

```python
>>> term = blessed.Terminal()
>>> term.rstrip(u'\x1b[0;3m XXX ')
u'XXX'
```

**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()`.

**strip(text, chars=none)**
Return `text` without sequences and leading or trailing whitespace.

**Return type** str

```python
>>> term = blessed.Terminal()
>>> term.strip(u'\x1b[0;3m XXX ')
u'XXX'
```

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

**Return type** str

```python
>>> term = blessed.Terminal()
>>> term.strip_seqs(u'\x1b[0;3mXXX')
u'XXX'
```

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

\text{\texttt{wrap}}(text, width=None, **kwargs)
Text-wrap a string, returning a list of wrapped lines.

Parameters

- text (str) – Unlike \texttt{textwrap.wrap()}, text may contain terminal sequences, such as colors, bold, or underline. By default, tabs in text are expanded by \texttt{string.expandtabs()}.

- width (int) – Unlike \texttt{textwrap.wrap()}, width will default to the width of the attached terminal.

Return type: list

See \texttt{textwrap.TextWrapper} for keyword arguments that can customize wrapping behaviour.

class WINSZ
Structure represents return value of \texttt{termios.TIOCGWINSZ}.

\texttt{ws_row}
rows, in characters

\texttt{ws_col}
columns, in characters

\texttt{ws_xpixel}
horizontal size, pixels

\texttt{ws_ypixel}
vertical size, pixels

\_CUR\_TERM = None

6.2 formatters.py

This sub-module provides sequence-formatting functions.

class FormattingString
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:

\begin{verbatim}
>>> style = FormattingString(term.bright_blue, term.normal)
>>> print(repr(style))
u'\x1b[94m'
>>> style('Big Blue')
u'\x1b[94mBig Blue\x1b(B\x1b[m'
\end{verbatim}

\_call\_\_\texttt{(*args)}
Return text joined by sequence and normal.

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.
__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.

class `ParameterizingProxyString`
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:

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

__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`

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

For example:

```python
>>> term = Terminal()
>>> color = ParameterizingString(term.color, term.normal, 'color')
>>> color(9)('color #9')
'\x1b[93mcolor #9\x1b[4m\x1b[0m'
```

__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.

Return type `FormattingString` or `NullCallableString`

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

Return type `set`
_make_compoundables(colors)
Return given set colors along with all “compoundable” attributes.

Parameters colors (set) – set of color names as string.
Return type set

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.

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”, respectively.

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

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.
Blessed Documentation, Release 1.11.0

Returns NullCallableString when number_of_colors is 0, otherwise FormattingString.

Return type NullCallableString or FormattingString

split_compound(compound)
Split compound formatting string into segments.

>>> split_compound('bold_underline_bright_blue_on_red')
['bold', 'underline', 'bright_blue', 'on_red']

Parameters compound (str) – a string that may contain compounds, separated by underline (_).

Return type list

COLORS = {'red', 'on_bright_blue', 'bright_green', 'bright_blue', 'yellow', 'bright_red', 'on_green', 'on_bright_yellow', ...
set() -> new empty set object set(iterable) -> new set object
Build an unordered collection of unique elements.

COMPOUNDABLES = {'bold', 'on_bright_blue', 'bright_blue', 'dim', 'shadow', 'on_green', 'on_bright_yellow', 'superscript', 'on_red', ...
set() -> new empty set object set(iterable) -> new set object
Build an unordered collection of unique elements.

6.3 keyboard.py

This sub-module provides ‘keyboard awareness’.

class Keystroke
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 be joined, split, or evaluated just as any other unicode string.

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

code
Integer keycode value of multibyte sequence (int).

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

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

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 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_PGU** in place of **KEY_PPAGE**
- **KEY_PGD** in place of **KEY_NPAGE**
- **KEY_ESC** 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

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

Return dict of sequences *term._cuf1*, and *term._cub1*, valued as KEY_RIGHT, KEY_LEFT (when appropriate). 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.

### _inject_curses_keynames()

Inject KEY_NAMES that we think would be useful into the curses module.

This function compliments the global constant *DEFAULT_SEQUENCE_MIXIN*. It is important to note that this function has the side-effect of *injecting* new attributes to the curses module, and is called from the global namespace at time of import.

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.
It is not necessary to directly “monkeypatch” the curses module to contain these constants, as they will also be accessible as attributes of the Terminal class instance, they are provided only for convenience when mixed in with other curses code.

```python
DEFAULT_SEQUENCE_MIXIN = (('\n', 343), ('\r', 343), ('\x08', 263), ('\t', 512), ('\x1b', 361), ('\x7f', 330), ('\x1b[A', 259), ('\x1b[B', ... 260), ('\x1b[O', 360), ('\x1b[OH', 262), ('\x1bOP', 265), ('\x1bOQ', 266), ('\x1bOR', 267), ('\x1bOS', 268))
```

If the argument is a tuple, the return value is the same object.

```python
CURSES_KEYCODE_OVERRIDE_MIXIN = (('KEY_DELETE', 330), ('KEY_INSERT', 331), ('KEY_PGUP', 339), ('KEY_PAGE_DOWN', 338), ('KEY_ESCAPE', 361), ('KEY_SUP', 337), ('KEY_SDOWN', 336), ('KEY_UP_LEFT', 348), ('KEY_UP_RIGHT', 349), ('KEY_CENTER', 350), ('KEY_BEGIN', 354))
```

If the argument is a tuple, the return value is the same object.

## 6.4 sequences.py

This module provides ‘sequence awareness’.

### init_sequence_patterns(term)

Build database of regular expressions of terminal sequences.

Given a Terminal instance, `term`, this function processes and parses several known terminal capabilities, and builds and returns a dictionary database of regular expressions, which is re-attached to the terminal by attributes of the same key-name.

**Parameters**


**Return type**

`dict`

**Returns**

Dictionary containing mappings of sequence “groups”, containing a compiled regular expression which it matches:

- `_re_will_move`
  
  Any sequence matching this pattern will cause the terminal cursor to move (such as `term.home`).

- `_re_wont_move`
  
  Any sequence matching this pattern will not cause the cursor to move (such as `term.bold`).

- `_re_cuf`
  
  Regular expression that matches `term.cuf(N)` (move N characters forward), or None if terminal is without cuf sequence.

- `_cuf1`
  
  `term.cuf1` sequence (cursor forward 1 character) as a static value.

- `_re_cub`
  
  Regular expression that matches `term.cub(N)` (move N characters backward), or None if terminal is without cub sequence.

- `_cub1`
  
  `term.cub1` sequence (cursor backward 1 character) as a static value.

These attributes make it possible to perform introspection on strings containing sequences generated by this terminal, to determine the printable length of a string.
class **Sequence**

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()`.

**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:

```python
>>> from blessed import Terminal
>>> from blessed.sequences import Sequence
>>> term = Terminal()
>>> Sequence(term.clear + term.red(u''), 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).

**ljust**(width, fillchar=' ')  
Return string containing sequences, left-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 right-of `text`.

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

**Return type** `str`

**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`

**padd**()  
Transform non-destructive space or backspace into destructive ones.
>>> from blessed import Terminal
>>> from blessed.sequences import Sequence
>>> term = Terminal()
>>> seq = term.cuf(10) + '->' + '\b\b'
>>> padded = Sequence(seq, Terminal()).pad()
>>> print(seq, padded)
(\x1b[10C--\x08\x08', u'  ')

Return type str

This method is used to determine the printable width of a string, and is the first pass of strip_seqs().

Where sequence term.cuf(n) is detected, it is replaced with n * u' ', and where sequence term.cubl(n) or \b is detected, those last-most characters are destroyed.

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

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

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

strip_seqs ()
Return string of all sequences removed.

Return type str

This method is used to determine the printable width of a string, and is the first pass of length().

Note: Non-destructive sequences that adjust horizontal distance (such as \b or term.cuf(5)) are replaced by destructive space or erasing.

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.

**_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.

**_wrap_chunks**


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.

**_sort_sequences**

Sort, filter, and return `regex_seqlist` in ascending order of length.

**Parameters** `regex_seqlist` *(list)* – list of strings.

**Return type** `list`

**Returns** given list filtered and sorted.
Any items that are Falsey (such as `None`, `''`) are removed from the return list. The longest expressions are returned first. Merge a list of input sequence patterns for use in a regular expression. Order by lengthyness (full sequence set precedent over subset), and exclude any empty (u'') sequences.

`_build_numeric_capability` *(term, cap, optional=False, base_num=99, nparams=1)*

Return regular expression for capabilities containing specified digits.

This differs from function `_build_any_numeric_capability()` in that, for the given `base_num` and `nparams`, the value of `<base_num>-1`, through `<base_num>+1` inclusive is replaced by regular expression pattern \d. Any other digits found are *not* replaced.

**Parameters**

- `cap` *(str)* – terminal capability name.
- `num` *(int)* – the numeric to use for parameterized capability.
- `nparams` *(int)* – the number of parameters to use for capability.

**Return type** `str`

**Returns** regular expression for the given capability.

`_build_any_numeric_capability` *(term, cap, num=99, nparams=1)*

Return regular expression for capabilities containing any numerics.

**Parameters**

- `cap` *(str)* – terminal capability name.
- `num` *(int)* – the numeric to use for parameterized capability.
- `nparams` *(int)* – the number of parameters to use for capability.

**Return type** `str`

**Returns** regular expression for the given capability.

Build regular expression from capabilities having *any* digit parameters: substitute any matching \d with literal \d and return.
We welcome contributions via GitHub pull requests:

- Fork a Repo
- Creating a pull request

7.1 Developing

Prepare a developer environment. Then, from the blessed code folder:

```
pip install --editable .
```

Any changes made are automatically made available to the python interpreter matching pip as the ‘blessed’ module path irregardless of the current working directory.

7.1.1 Running Tests

Install and run tox

```
pip install --upgrade tox
tox
```

Py.test is used as the test runner, supporting positional arguments, you may for example use looponfailing <https://pytest.org/latest/xdist.html#running-tests-in-looponfailing-mode> with python 3.5, stopping at the first failing test case, and looping (retrying) after a filesystem save is detected:

```
tox -epy35 -- -fx
```

7.1.2 Test Coverage

When you contribute a new feature, make sure it is covered by tests. Likewise, a bug fix should include a test demonstrating the bug. Blessed has nearly 100% line coverage, with roughly 1/2 of the codebase in the form of tests, which are further combined by a matrix of varying TERM types, providing plenty of existing test cases to augment or duplicate in your favor.
7.1.3 Style and Static Analysis

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 through the sa target, invoked using:

```
tox -esa
```

Similarly, positional arguments can be used, for example to verify URL links:

```
tox -esa -- -blinkcheck
```

All standards enforced by the underlying tools are adhered to by the blessed project, with the declarative exception of those found in landscape.yml, or inline using pylint: disable= directives.
CHAPTER 8

Version History

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 \texttt{rstrip()} and \texttt{lstrip()}, strips both sequences and trailing or leading whitespace, respectively.

• enhancement: include \texttt{wcwidth} library support for \texttt{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 \texttt{ecma-48} codes when using xterm or aixterm.

• bugfix: when \texttt{locale.getpreferredencoding()} returns empty string or an encoding that is not valid for \texttt{codecs.getincrementaldecoder}, fallback to ASCII and emit a warning.

• bugfix: ensure \texttt{FormattingString} and \texttt{ParameterizingString} may be pickled.

• bugfix: allow \texttt{~.inkey} and related to be called without a keyboard.

• change: \texttt{term.keyboard_fd} is set \texttt{None} if \texttt{stream or sys.stdout} is not a tty, making \texttt{term.inkey()}, \texttt{term.cbreak()}, \texttt{term.raw()}, \texttt{no-op}.

• bugfix: escape code \texttt{\x1bOH} (KEY_HOME) was incorrectly mapped as KEY_LEFT.

1.7

• Forked github project \texttt{erikrose/blessings} to \texttt{jquast/blessed}, this project was previously known as \texttt{blessings} version 1.6 and prior.

• introduced: context manager \texttt{cbreak()}, which is equivalent to entering terminal state by \texttt{tty.setcbreak()} and returning on exit, as well as the lesser recommended \texttt{raw()}, pairing from \texttt{tty.setraw()}.

• introduced: \texttt{inkey()}, which will return one or more characters received by the keyboard as a unicode sequence, with additional attributes \texttt{code} and \texttt{name}. This allows application keys (such as the up arrow, or home key) to be detected. Optional value \texttt{timeout} allows for timed poll.

• introduced: \texttt{center()}, \texttt{rjust()}, \texttt{ljust()}, allowing text containing sequences to be aligned to detected horizontal screen width, or by \texttt{width} specified.

• introduced: \texttt{wrap()} method. Allows text containing sequences to be word-wrapped without breaking mid-sequence, honoring their printable width.

• introduced: \texttt{strip()}, strips all sequences and whitespace.

• introduced: \texttt{strip_seqs()} strip only sequences.

• introduced: \texttt{rstrip()} and \texttt{lstrip()} strips both sequences and trailing or leading whitespace, respectively.

• bugfix: cannot call \texttt{curses.setupterm()} more than once per process (from \texttt{Terminal.__init__()}): Previously, blessed pretended to support several instances of different Terminal \texttt{kind}, but was actually using the \texttt{kind} specified by the first instantiation of \texttt{Terminal}. A warning is now issued. Although this is misbehavior is still allowed, a \texttt{warnings.WarningMessage} is now emitted to notify about subsequent terminal misbehavior.

• bugfix: resolved issue where \texttt{number_of_colors} fails when \texttt{does_styling} is \texttt{False}. Resolves issue where piping tests output would fail.

• bugfix: warn and set \texttt{does_styling} to \texttt{False} when the given \texttt{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 \texttt{color(n)} may be called on terminals without color capabilities.
• bugfix: for terminals without underline, such as vt220, `term.underline('text')` would emit `u'text' + term.normal`. Now it emits only `u'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 support).

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 PR #29).

• Parameterizing a capability no longer crashes when there is no tty (Vitja Makarov PR #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 (http://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 9

Indexes

- genindex
- modindex
b

blessed.formatters, 33
blessed.keyboard, 36
blessed.sequences, 38
blessed.terminal, 27
Symbols

_INDEX_TERM (in module blessed.terminal), 33
__call__() (FormattingString method), 33
__call__() (NullCallableString method), 33
__call__() (ParameterizingProxyString method), 34
__call__() (ParameterizingString method), 34
__getattr__() (Terminal method), 27
__new__() (Keystroke static method), 36
_alterative_left_right() (in module blessed.keyboard), 37
_build_any_numeric_capability() (in module blessed.sequences), 42
_build_numeric_capability() (in module blessed.sequences), 42
_handle_long_word() (SequenceTextWrapper method), 41
_inject_curses_keynames() (in module blessed.keyboard), 37
_make_colors() (in module blessed.formatters), 37
_make_compoundables() (in module blessed.formatters), 34
_sort_sequences() (in module blessed.sequences), 41
_wrap_chunks() (SequenceTextWrapper method), 41

D
DEFAULT_SEQUENCE_MIXIN (in module blessed.keyboard), 38
does_styling (Terminal attribute), 28

F
FormattingString (class in blessed.formatters), 33
fullscreen() (Terminal method), 28

G
get_keyboard_codes() (in module blessed.keyboard), 36
get_keyboard_sequences() (in module blessed.keyboard), 37
get_proxy_string() (in module blessed.formatters), 35
getch() (Terminal method), 29

H
height (Terminal attribute), 29
hidden_cursor() (Terminal method), 29

I
inkey() (Terminal method), 29
is_a_tty (Terminal attribute), 30
is_sequence (Keystroke attribute), 36

K
kbhit() (Terminal method), 30
keypad() (Terminal method), 30
Keystroke (class in blessed.keyboard), 36
kind (Terminal attribute), 30

L
length() (Sequence method), 39
length() (Terminal method), 30
ljust() (Sequence method), 39
ljust() (Terminal method), 30
location() (Terminal method), 31
lstrip() (Sequence method), 39
lstrip() (Terminal method), 31

N
name (Keystroke attribute), 36
normal (Terminal attribute), 31
NullCallableString (class in blessed.formatters), 33
number_of_colors (Terminal attribute), 31

O
on_color (Terminal attribute), 31

P
padd() (Sequence method), 39
ParameterizingProxyString (class in blessed.formatters), 34
ParameterizingString (class in blessed.formatters), 34

R
raw() (Terminal method), 31
resolve_attribute() (in module blessed.formatters), 35
resolve_capability() (in module blessed.formatters), 35
resolve_color() (in module blessed.formatters), 35
rjust() (Sequence method), 40
rjust() (Terminal method), 32
rstrip() (Sequence method), 40
rstrip() (Terminal method), 32

S
Sequence (class in blessed.sequences), 38
SequenceTextWrapper (class in blessed.sequences), 40
split_compound() (in module blessed.formatters), 36
stream (Terminal attribute), 32
strip() (Sequence method), 40
strip() (Terminal method), 32
strip_seqs() (Sequence method), 40
strip_seqs() (Terminal method), 32

T
Terminal (class in blessed.terminal), 27

W
width (Terminal attribute), 32
WINSZ (class in blessed.terminal), 33
wrap() (Terminal method), 33
ws_col (WINSZ attribute), 33
ws_row (WINSZ attribute), 33
ws_xpixel (WINSZ attribute), 33
ws_ypixel (WINSZ attribute), 33