Blessed Documentation

Release 1.9.4

Jeff Quast

October 03, 2015

Contents

CHAPTER 1

Read The Readme First

This is the API documentation for the Blessed terminal library.

Because Blessed uses quite a bit of dynamism, you should read the readme first for a general guide and overview.

However, if you're looking for the documentation of the internal classes, their methods, and related functions that make up the internals, you're in the right place.

CHAPTER 2

API Documentation

Internal modules are as follows.

2.1 terminal module (primary)

This primary module provides the Terminal class.

class `blessed.terminal.`**`Terminal`**(*kind=None*, *stream=None*, *force_styling=False*)

A wrapper for curses and related terminfo(5) terminal capabilities.

Instance attributes:

`stream` The stream the terminal outputs to. It's convenient to pass the stream around with the terminal; it's almost always needed when the terminal is and saves sticking lots of extra args on client functions in practice.

`kind`

Name of this terminal type as string.

`does_styling`

Whether this instance will emit terminal sequences (bool).

`is_a_tty`

Whether the `stream` associated with this instance is a terminal (bool).

`height`

T.height -> int

The height of the terminal in characters.

`width`

T.width -> int

The width of the terminal in characters.

`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:

```
term = Terminal()
with term.location(2, 5):
    print 'Hello, world!'
```

```
            for x in xrange(10):
                print 'I can do it %i times!' % x
```

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.

fullscreen()
Return a context manager that enters fullscreen mode while inside it and restores normal mode on leaving.

Fullscreen mode is characterized by instructing the terminal emulator to store and save the current screen state (all screen output), switch to "alternate screen". Upon exiting, the previous screen state is returned.

This call may not be tested; only one screen state may be saved at a time.

hidden_cursor()
Return a context manager that hides the cursor upon entering, and makes it visible again upon exiting.

color
Returns capability that sets the foreground color.

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.

Parameters **num** – The number, 0-15, of the color

on_color
Returns capability that sets the background color.

normal
Returns sequence that resets video attribute.

number_of_colors
Return the number of colors the terminal supports.

Common values are 0, 8, 16, 88, and 256. Most commonly this may be used to test color capabilities at all:

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

ljust(*text*[, *width*][, *fillchar*]) → unicode
Return string `text`, left-justified by printable length `width`. Padding is done using the specified fill character (default is a space). Default `width` is the attached terminal's width. `text` may contain terminal sequences.

rjust(*text*[, *width*][, *fillchar*]) → unicode
Return string `text`, right-justified by printable length `width`. Padding is done using the specified fill character (default is a space). Default `width` is the attached terminal's width. `text` may contain terminal sequences.

center(*text*[, *width*][, *fillchar*]) → unicode
Return string `text`, centered by printable length `width`. Padding is done using the specified fill character (default is a space). Default `width` is the attached terminal's width. `text` may contain terminal sequences.

length(*text*) → int
Return the printable length of string `text`, which may contain terminal sequences. Strings containing sequences such as 'clear', which repositions the cursor, does not give accurate results, and their printable length is evaluated *0*..

strip(*text*) → unicode
Return string `text` with terminal sequences removed, and leading and trailing whitespace removed.

rstrip(*text*) → unicode
Return string `text` with terminal sequences and trailing whitespace removed.

lstrip(*text*) → unicode
Return string `text` with terminal sequences and leading whitespace removed.

strip_seqs(*text*) → unicode
Return string `text` stripped only of its sequences.

wrap(*text*[, *width=None*, ***kwargs* ..]) → list[unicode]
Wrap paragraphs containing escape sequences `text` to the full `width` of Terminal instance *T*, unless `width` is specified. Wrapped by the virtual printable length, irregardless of the video attribute sequences it may contain, allowing text containing colors, bold, underline, etc. to be wrapped.

Returns a list of strings that may contain escape sequences. See `textwrap.TextWrapper` for all available additional kwargs to customize wrapping behavior such as `subsequent_indent`.

Note that the keyword argument `break_long_words` may not be set, it is not sequence-safe!

getch() → unicode
Read and decode next byte from keyboard stream. May return u" if decoding is not yet complete, or completed unicode character. Should always return bytes when self.kbhit() returns True.

Implementors of input streams other than os.read() on the stdin fd should derive and override this method.

kbhit([*timeout=None*]) → bool
Returns True if a keypress has been detected on keyboard.

When `timeout` is 0, this call is non-blocking, Otherwise blocking until keypress is detected (default). When `timeout` is a positive number, returns after `timeout` seconds have elapsed.

If input is not a terminal, False is always returned.

cbreak()
Return a context manager that enters 'cbreak' mode: disabling line buffering of keyboard input, making characters typed by the user immediately available to the program. Also referred to as 'rare' mode, this is the opposite of 'cooked' mode, the default for most shells.

In 'cbreak' mode, echo of input is also disabled: the application must explicitly print any input received, if they so wish.

More information can be found in the manual page for curses.h, http://www.openbsd.org/cgi-bin/man.cgi?query=cbreak

The python manual for curses, http://docs.python.org/2/library/curses.html

Note also that setcbreak sets VMIN = 1 and VTIME = 0, http://www.unixwiz.net/techtips/termios-vmin-vtime.html

raw()
Return a context manager that enters *raw* mode. Raw mode is similar to *cbreak* mode, in that characters typed are immediately available to `inkey()` with one exception: the interrupt, quit, suspend, and flow control characters are all passed through as their raw character values instead of generating a signal.

keypad()
Context manager that enables keypad input (*keyboard_transmit* mode).

This enables the effect of calling the curses function keypad(3x): display terminfo(5) capability *keypad_xmit* (smkx) upon entering, and terminfo(5) capability *keypad_local* (rmkx) upon exiting.

On an IBM-PC keypad of ttype *xterm*, with numlock off, the lower-left diagonal key transmits sequence `\x1b[F`, `KEY_END`.

However, upon entering keypad mode, `\x1b[OF` is transmitted, translating to `KEY_LL` (lower-left key), allowing diagonal direction keys to be determined.

inkey(*timeout=None, [esc_delay, [_intr_continue]]*) -> *Keypress()*

Receive next keystroke from keyboard (stdin), blocking until a keypress is received or `timeout` elapsed, if specified.

When used without the context manager `cbreak`, stdin remains line-buffered, and this function will block until return is pressed, even though only one unicode character is returned at a time..

The value returned is an instance of `Keystroke`, with properties `is_sequence`, and, when True, non-None values for attributes `code` and `name`. The value of `code` may be compared against attributes of this terminal beginning with *KEY*, such as `KEY_ESCAPE`.

To distinguish between `KEY_ESCAPE`, and sequences beginning with escape, the `esc_delay` specifies the amount of time after receiving the escape character (chr(27)) to seek for the completion of other application keys before returning `KEY_ESCAPE`.

Normally, when this function is interrupted by a signal, such as the installment of SIGWINCH, this function will ignore this interruption and continue to poll for input up to the `timeout` specified. If you'd rather this function return `u''` early, specify a value of `False` for `_intr_continue`.

class `blessed.terminal.`**WINSZ**(*ws_row*, *ws_col*, *ws_xpixel*, *ws_ypixel*)

ws_col
Alias for field number 1

ws_row
Alias for field number 0

ws_xpixel
Alias for field number 2

ws_ypixel
Alias for field number 3

2.2 formatters module

This sub-module provides formatting functions.

`blessed.formatters.`**COLORS = {'black', 'bright_black', 'on_bright_cyan', 'bright_yellow', 'on_red', 'bright_cyan', 'on_b**
Valid colors and their background (on), bright, and bright-bg derivatives.

`blessed.formatters.`**COMPOUNDABLES = {'on_bright_blue', 'bright_black', 'on_bright_cyan', 'bright_yellow', 'on_red',**
All valid compoundable names.

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

For example:

```
>> term = Terminal()
>> color = ParameterizingString(term.color, term.normal, 'color')
>> color(9)('color #9')
u'[91mcolor #9(B[m'
```

class blessed.formatters.**ParameterizingProxyString**
A Unicode string which can be called to proxy missing termcap entries.

For example:

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

blessed.formatters.**get_proxy_string**(*term*, *attr*)
Proxy and return callable StringClass for proxied attributes.

We know that some kinds of terminal kinds support sequences that the terminfo database often doesn't report – such as the 'move_x' attribute for terminal type 'screen', or 'hide_cursor' for 'ansi'.

Returns instance of ParameterizingProxyString or NullCallableString.

class blessed.formatters.**FormattingString**
A Unicode string which can be called using `text`, returning a new string, `attr` + `text` + `normal`:

```
>> style = FormattingString(term.bright_blue, term.normal)
>> style('Big Blue')
u'[94mBig Blue(B[m'
```

class blessed.formatters.**NullCallableString**
A dummy callable Unicode to stand in for `FormattingString` and `ParameterizingString` for terminals that cannot perform styling.

blessed.formatters.**split_compound**(*compound*)
Split a possibly compound format string into segments.

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

blessed.formatters.**resolve_capability**(*term*, *attr*)
Return a Unicode string for the terminal capability `attr`, or an empty string if not found, or if terminal is without styling capabilities.

blessed.formatters.**resolve_color**(*T*, *color) -> FormattingString()*
Resolve a `color` name to callable capability, `FormattingString` unless `term.number_of_colors` is 0, then `NullCallableString`.

Valid `color` capabilities names are any of the simple color names, such as `red`, or compounded, such as `on_bright_green`.

blessed.formatters.**resolve_attribute**(*term*, *attr*)
Resolve a sugary or plain capability name, color, or compound formatting name into a *callable* unicode string capability, `ParameterizingString` or `FormattingString`.

2.3 keyboard module

This sub-module provides 'keyboard awareness'.

class blessed.keyboard.**Keystroke**
A unicode-derived class for describing keyboard input returned by the inkey() method of Terminal, which may, at times, be a multibyte sequence, providing properties is_sequence as True when the string is a known sequence, and code, which returns an integer value that may be compared against the terminal class attributes such as KEY_LEFT.

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

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

code
Integer keycode value of multibyte sequence (int).

blessed.keyboard.**get_keyboard_codes**() → dict
Returns dictionary of (code, name) pairs for curses keyboard constant values and their mnemonic name. Such as key 260, with the value of its identity, KEY_LEFT. These are derived from the attributes by the same of the curses module, with the following exceptions:

- KEY_DELETE in place of KEY_DC
- KEY_INSERT in place of KEY_IC
- KEY_PGUP in place of KEY_PPAGE
- KEY_PGDOWN in place of KEY_NPAGE
- KEY_ESCAPE in place of KEY_EXIT
- KEY_SUP in place of KEY_SR
- KEY_SDOWN in place of KEY_SF

blessed.keyboard.**get_keyboard_sequences**(*T) -> (OrderedDict*)
Initialize and return a keyboard map and sequence lookup table, (sequence, constant) from blessed Terminal instance term, where sequence is a multibyte input sequence, such as u'[D', and constant is a constant, such as term.KEY_LEFT. The return value is an OrderedDict instance, with their keys sorted longest-first.

2.4 sequences module

This sub-module provides 'sequence awareness' for blessed.

blessed.sequences.**init_sequence_patterns**(*term*)
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 may be re-attached to the terminal by attributes of the same key-name:

_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 temrinal 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.cuf1* 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 `blessed.sequences.`**`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 `blessed.sequences.`**`Sequence`**

This unicode-derived class understands the effect of escape sequences of printable length, allowing a properly implemented .rjust(), .ljust(), .center(), and .len()

`ljust`(*width*, *fillchar*) → unicode

Returns string derived from unicode string `S`, left-adjusted by trailing whitespace padding `fillchar`.

`rjust`(*width*, *fillchar=u' '*) → unicode

Returns string derived from unicode string `S`, right-adjusted by leading whitespace padding `fillchar`.

`center`(*width*, *fillchar=u' '*) → unicode

Returns string derived from unicode string `S`, centered and surrounded with whitespace padding `fillchar`.

`length`() → int

Returns printable length of unicode string `S` that may contain terminal sequences.

Although accounted for, strings containing sequences such as `term.clear` will not give accurate returns, it is not considered lengthy (a length of 0). Combining characters, are also not considered lengthy.

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

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

For example:

```
>>> from blessed import Terminal
>>> from blessed.sequences import Sequence
>>> term = Terminal()
>>> Sequence(term.clear + term.red(u'')).length()
5
```

strip([*chars*]) → unicode
Return a copy of the string S with terminal sequences removed, and leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

lstrip([*chars*]) → unicode
Return a copy of the string S with terminal sequences and leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

rstrip([*chars*]) → unicode
Return a copy of the string S with terminal sequences and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

strip_seqs() → unicode
Return a string without sequences for a string that contains sequences for the Terminal with which they were created.

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

All other sequences are simply removed. An example,

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

padd() → unicode
Make non-destructive space or backspace into destructive ones.

Where sequence `move_right(n)` is detected, it is replaced with `n * u' '`. Where sequence `move_left(n)` or `\b` is detected, those last-most characters are destroyed.

b

B

C

D

F

G

H

I

K

L

N

O

P

R

S

T

W