Package 'fansi'

December 8, 2023

Title ANSI Control Sequence Aware String Functions

Description Counterparts to R string manipulation functions that account for the effects of ANSI text formatting control sequences.

Version 1.0.6

Depends R (>= 3.1.0)

License GPL-2 | GPL-3

URL https://github.com/brodieG/fansi

BugReports https://github.com/brodieG/fansi/issues

VignetteBuilder knitr

Suggests unitizer, knitr, rmarkdown

Imports grDevices, utils

RoxygenNote 7.2.3

Encoding UTF-8

Collate 'constants.R' 'fansi-package.R' 'internal.R' 'load.R' 'misc.R' 'nchar.R' 'strwrap.R' 'strtrim.R' 'strsplit.R' 'substr2.R' 'trimws.R' 'tohtml.R' 'unhandled.R' 'normalize.R' 'sgr.R'

NeedsCompilation yes

Author Brodie Gaslam [aut, cre],
Elliott Sales De Andrade [ctb],
R Core Team [cph] (UTF8 byte length calcs from src/util.c)

Maintainer Brodie Gaslam <brodie.gaslam@yahoo.com>

Repository CRAN

Date/Publication 2023-12-08 03:30:02 UTC

R topics documented:

`dflt_term_cap` *Default Arg Helper Funs*

Description

Terminal capabilities are assumed to include bright and 256 color SGR codes. 24 bit color support is detected based on the `COLORTERM` environment variable.

Usage

```
dflt_term_cap()

dflt_css()
```

Details

Default CSS may exceed or fail to cover the interline distance when two lines have background colors. To ensure lines are exactly touching use inline-block, although that has its own issues. Otherwise specify your own CSS.

Value

character to use as default value for `fansi` parameter.

See Also

term_cap_test.

---

fansi *Details About Manipulation of Strings Containing Control Sequences*

Description

Counterparts to R string manipulation functions that account for the effects of some ANSI X3.64 (a.k.a. ECMA-48, ISO-6429) control sequences.

Control Characters and Sequences

Control characters and sequences are non-printing inline characters or sequences initiated by them that can be used to modify terminal display and behavior, for example by changing text color or cursor position.

We will refer to X3.64/ECMA-48/ISO-6429 control characters and sequences as "*Control Sequences*" hereafter.

There are four types of *Control Sequences* that fansi can treat specially:

- "C0" control characters, such as tabs and carriage returns (we include delete in this set, even though technically it is not part of it).
- Sequences starting in "ESC[", also known as Control Sequence Introducer (CSI) sequences, of which the Select Graphic Rendition (SGR) sequences used to format terminal output are a subset.
- Sequences starting in "ESC]", also known as Operating System Commands (OSC), of which the subset beginning with "8" is used to encode URI based hyperlinks.
- Sequences starting in "ESC" and followed by something other than "[" or "]".

Control Sequences starting with ESC are assumed to be two characters long (including the ESC) unless they are of the CSI or OSC variety, in which case their length is computed as per the ECMA-48 specification, with the exception that OSC hyperlinks may be terminated with BEL ("\a") in addition to ST ("ESC\"). fansi handles most common *Control Sequences* in its parsing algorithms, but it is not a conforming implementation of ECMA-48. For example, there are non-CSI/OSC escape sequences that may be longer than two characters, but fansi will (incorrectly) treat them as if they were two characters long. There are many more unimplemented ECMA-48 specifications.

In theory it is possible to encode CSI sequences with a single byte introducing character in the 0x40-0x5F range instead of the traditional "ESC[". Since this is rare and it conflicts with UTF-8 encoding, fansi does not support it.

Within *Control Sequences*, fansi further distinguishes CSI SGR and OSC hyperlinks by recording format specification and URIs into string state, and applying the same to any output strings according to the semantics of the functions in use. CSI SGR and OSC hyperlinks are known together as *Special Sequences*. See the following sections for details.

Additionally, all *Control Sequences*, whether special or not, do not count as characters, graphemes, or display width. You can cause fansi to treat particular *Control Sequences* as regular characters with the ctl parameter.

CSI SGR Control Sequences

NOTE: not all displays support CSI SGR sequences; run `term_cap_test` to see whether your display supports them.

CSI SGR Control Sequences are the subset of CSI sequences that can be used to change text appearance (e.g. color). These sequences begin with "ESC[" and end in "m". `fansi` interprets these sequences and writes new ones to the output strings in such a way that the original formatting is preserved. In most cases this should be transparent to the user.

Occasionally there may be mismatches between how `fansi` and a display interpret the CSI SGR sequences, which may produce display artifacts. The most likely source of artifacts are *Control Sequences* that move the cursor or change the display, or that `fansi` otherwise fails to interpret, such as:

- Unknown SGR substrings.
- "C0" control characters like tabs and carriage returns.
- Other escape sequences.

Another possible source of problems is that different displays parse and interpret control sequences differently. The common CSI SGR sequences that you are likely to encounter in formatted text tend to be treated consistently, but less common ones are not. `fansi` tries to hew by the ECMA-48 specification **for CSI SGR control sequences**, but not all terminals do.

The most likely source of problems will be 24-bit CSI SGR sequences. For example, a 24-bit color sequence such as "ESC[38;2;31;42;4" is a single foreground color to a terminal that supports it, or separate foreground, background, faint, and underline specifications for one that does not. `fansi` will always interpret the sequences according to ECMA-48, but it will warn you if encountered sequences exceed those specified by the `term.cap` parameter or the "fansi.term.cap" global option.

`fansi` will will also warn if it encounters *Control Sequences* that it cannot interpret. You can turn off warnings via the `warn` parameter, which can be set globally via the "fansi.warn" option. You can work around "C0" tabs characters by turning them into spaces first with `tabs_as_spaces` or with the `tabs.as.spaces` parameter available in some of the `fansi` functions

`fansi` interprets CSI SGR sequences in cumulative "Graphic Rendition Combination Mode". This means new SGR sequences add to rather than replace previous ones, although in some cases the effect is the same as replacement (e.g. if you have a color active and pick another one).

OSC Hyperlinks

Operating System Commands are interpreted by terminal emulators typically to engage actions external to the display of text proper, such as setting a window title or changing the active color palette.

Some terminals have added support for associating URIs to text with OSCs in a similar way to anchors in HTML, so `fansi` interprets them and outputs or terminates them as needed. For example:

```
"\033]8;;xy.z\033\\LINK\033]8;;\033\\"
```

Might be interpreted as link to the URI "x.z". To make the encoding pattern clearer, we replace "\033]" with "<OSC>" and "\033\\" with "<ST>" below:

```
<OSC>8;;URI<ST>LINK TEXT<OSC>8;;<ST>
```

State Interactions

The cumulative nature of state as specified by SGR or OSC hyperlinks means that unterminated strings that are spliced will interact with each other. By extension, a substring does not inherently contain all the information required to recreate its state as it appeared in the source document. The default `fansi` configuration terminates extracted substrings and prepends original state to them so they present on a stand-alone basis as they did as part of the original string.

To allow state in substrings to affect subsequent strings set `terminate = FALSE`, but you will need to manually terminate them or deal with the consequences of not doing so (see "Terminal Quirks").

By default, `fansi` assumes that each element in an input character vector is independent, but this is incorrect if the input is a single document with each element a line in it. In that situation state from each line should bleed into subsequent ones. Setting `carry = TRUE` enables the "single document" interpretation.

To most closely approximate what `writeLines(x)` produces on your terminal, where `x` is a stateful string, use `writeLines(fansi_fun(x, carry=TRUE, terminate=FALSE))`. `fansi_fun` is a stand-in for any of the `fansi` string manipulation functions. Note that even with a seeming "null-op" such as `substr_ctl(x, 1, nchar_ctl(x), carry=TRUE, terminate=FALSE)` the output control sequences may not match the input ones, but the output *should* look the same if displayed to the terminal.

`fansi` strings will be affected by any active state in strings they are appended to. There are no parameters to control what happens in this case, but `fansi` provides functions that can help the user get the desired behavior. `state_at_end` computes the active state the end of a string, which can then be prepended onto the *input* of `fansi` functions so that they are aware of the active style at the beginning of the string. Alternatively, one could use `close_state(state_at_end(...))` and prepend that to the *output* of `fansi` functions so they are unaffected by preceding SGR. One could also just prepend "ESC[0m", but in some cases as described in `?normalize_state` that is sub-optimal.

If you intend to combine stateful `fansi` manipulated strings with your own, it may be best to set `normalize = TRUE` for improved compatibility (see `?normalize_state`.)

Terminal Quirks

Some terminals (e.g. OS X terminal, ITerm2) will pre-paint the entirety of a new line with the currently active background before writing the contents of the line. If there is a non-default active background color, any unwritten columns in the new line will keep the prior background color even if the new line changes the background color. To avoid this be sure to use `terminate = TRUE` or to manually terminate each line with e.g. "ESC[0m". The problem manifests as:

```
" " = default background
"#" = new background
">" = start new background
"!" = restore default background

+-----------+
| abc\n     |
|>###\n     |
|!abc\n#####| <- trailing "#" after newline are from pre-paint
| abc       |
+-----------+
```

The simplest way to avoid this problem is to split input strings by any newlines they contain, and use `terminate = TRUE` (the default). A more complex solution is to pad with spaces to the terminal window width before emitting the newline to ensure the pre-paint is overpainted with the current line's prevailing background color.

Encodings / UTF-8

`fansi` will convert any non-ASCII strings to UTF-8 before processing them, and `fansi` functions that return strings will return them encoded in UTF-8. In some cases this will be different to what base R does. For example, `substr` re-encodes substrings to their original encoding.

Interpretation of UTF-8 strings is intended to be consistent with base R. There are three ways things may not work out exactly as desired:

1. `fansi`, despite its best intentions, handles a UTF-8 sequence differently to the way R does.
2. R incorrectly handles a UTF-8 sequence.
3. Your display incorrectly handles a UTF-8 sequence.

These issues are most likely to occur with invalid UTF-8 sequences, combining character sequences, and emoji. For example, whether special characters such as emoji are considered one or two wide evolves as software implements newer versions the Unicode databases.

Internally, `fansi` computes the width of most UTF-8 character sequences outside of the ASCII range using the native `R_nchar` function. This will cause such characters to be processed slower than ASCII characters. Unlike R (at least as of version 4.1), `fansi` can account for graphemes.

Because `fansi` implements its own internal UTF-8 parsing it is possible that you will see results different from those that R produces even on strings without *Control Sequences*.

Overflow

The maximum length of input character vector elements allowed by `fansi` is the 32 bit INT_MAX, excluding the terminating NULL. As of R4.1 this is the limit for R character vector elements generally, but is enforced at the C level by `fansi` nonetheless.

It is possible that during processing strings that are shorter than INT_MAX would become longer than that. `fansi` checks for that overflow and will stop with an error if that happens. A work-around for this situation is to break up large strings into smaller ones. The limit is on each element of a character vector, not on the vector as a whole. `fansi` will also error on your system if `R_len_t`, the R type used to measure string lengths, is less than the processed length of the string.

R < 3.2.2 support

Nominally you can build and run this package in R versions between 3.1.0 and 3.2.1. Things should mostly work, but please be aware we do not run the test suite under versions of R less than 3.2.2. One key degraded capability is width computation of wide-display characters. Under R < 3.2.2 `fansi` will assume every character is 1 display width. Additionally, `fansi` may not always report malformed UTF-8 sequences as it usually does. One exception to this is `nchar_ctl` as that is just a thin wrapper around `base::nchar`.

fansi_lines *Colorize Character Vectors*

Description

Color each element in input with one of the "256 color" ANSI CSI SGR codes. This is intended for testing and demo purposes.

Usage

```
fansi_lines(txt, step = 1)
```

Arguments

txt	character vector or object that can be coerced to character vector
step	integer(1L) how quickly to step through the color palette

Value

character vector with each element colored

Examples

```
NEWS <- readLines(file.path(R.home('doc'), 'NEWS'))
writeLines(fansi_lines(NEWS[1:20]))
writeLines(fansi_lines(NEWS[1:20], step=8))
```

has_ctl *Check for Presence of Control Sequences*

Description

has_ctl checks for any *Control Sequence*. You can check for different types of sequences with the ctl parameter. Warnings are only emitted for malformed CSI or OSC sequences.

Usage

```
has_ctl(x, ctl = "all", warn = getOption("fansi.warn", TRUE), which)
```

Arguments

x	a character vector or object that can be coerced to such.
ctl	character, which *Control Sequences* should be treated specially. Special treatment is context dependent, and may include detecting them and/or computing their display/character width as zero. For the SGR subset of the ANSI CSI sequences, and OSC hyperlinks, fansi will also parse, interpret, and reapply the sequences as needed. You can modify whether a *Control Sequence* is treated specially with the ctl parameter. • "nl": newlines. • "c0": all other "C0" control characters (i.e. 0x01-0x1f, 0x7F), except for newlines and the actual ESC (0x1B) character. • "sgr": ANSI CSI SGR sequences. • "csi": all non-SGR ANSI CSI sequences. • "url": OSC hyperlinks • "osc": all non-OSC-hyperlink OSC sequences. • "esc": all other escape sequences. • "all": all of the above, except when used in combination with any of the above, in which case it means "all but".
warn	TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions fansi makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see ?fansi). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use validUTF8 for that).
which	character, deprecated in favor of ctl.

Value

logical of same length as x; NA values in x result in NA values in return

See Also

?fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, unhandled_ctl for detecting bad control sequences.

Examples

```
has_ctl("hello world")
has_ctl("hello\nworld")
has_ctl("hello\nworld", "sgr")
has_ctl("hello\033[31mworld\033[m", "sgr")
```

html_code_block *Format Character Vector for Display as Code in HTML*

Description

This simulates what rmarkdown / knitr do to the output of an R markdown chunk, at least as of rmarkdown 1.10. It is useful when we override the knitr output hooks so that we can have a result that still looks as if it was run by knitr.

Usage

```
html_code_block(x, class = "fansi-output")
```

Arguments

x	character vector
class	character vectors of classes to apply to the PRE HTML tags. It is the users responsibility to ensure the classes are valid CSS class names.

Value

character(1L) x, with <PRE> and <CODE> HTML tags applied and collapsed into one line with newlines as the line separator.

Examples

```
html_code_block(c("hello world"))
html_code_block(c("hello world"), class="pretty")
```

html_esc *Escape Characters With Special HTML Meaning*

Description

Arbitrary text may contain characters with special meaning in HTML, which may cause HTML display to be corrupted if they are included unescaped in a web page. This function escapes those special characters so they do not interfere with the HTML markup generated by e.g. to_html.

Usage

```
html_esc(x, what = getOption("fansi.html.esc", "<>&'\""))
```

Arguments

x	character vector
what	character(1) containing any combination of ASCII characters "<", ">", "&", "'", or "\"". These characters are special in HTML contexts and will be substituted by their HTML entity code. By default, all special characters are escaped, but in many cases "<>&" or even "<>" might be sufficient. @return x, but with the what characters replaced by their HTML entity codes.

Note

Non-ASCII strings are converted to and returned in UTF-8 encoding.

See Also

Other HTML functions: in_html(), make_styles(), to_html()

Examples

```
html_esc("day > night")
html_esc("<SPAN>hello world</SPAN>")
```

---

in_html *Frame HTML in a Web Page And Display*

Description

Helper function that assembles user provided HTML and CSS into a temporary text file, and by default displays it in the browser. Intended for use in examples.

Usage

```
in_html(x, css = character(), pre = TRUE, display = TRUE, clean = display)
```

Arguments

x	character vector of html encoded strings.
css	character vector of css styles.
pre	TRUE (default) or FALSE, whether to wrap x in PRE tags.
display	TRUE or FALSE, whether to display the resulting page in a browser window. If TRUE, will sleep for one second before returning, and will delete the temporary file used to store the HTML.
clean	TRUE or FALSE, if TRUE and display == TRUE, will delete the temporary file used for the web page, otherwise will leave it.

Value

character(1L) the file location of the page, invisibly, but keep in mind it will have been deleted if `clean=TRUE`.

See Also

`make_styles()`.

Other HTML functions: `html_esc()`, `make_styles()`, `to_html()`

Examples

```
txt <- "\033[31;42mHello \033[7mWorld\033[m"
writeLines(txt)
html <- to_html(txt)
## Not run:
in_html(html) # spawns a browser window

## End(Not run)
writeLines(readLines(in_html(html, display=FALSE)))
css <- "SPAN {text-decoration: underline;}"
writeLines(readLines(in_html(html, css=css, display=FALSE)))
## Not run:
in_html(html, css)

## End(Not run)
```

---

make_styles *Generate CSS Mapping Classes to Colors*

Description

Given a set of class names, produce the CSS that maps them to the default 8-bit colors. This is a helper function to generate style sheets for use in examples with either default or remixed `fansi` colors. In practice users will create their own style sheets mapping their classes to their preferred styles.

Usage

```
make_styles(classes, rgb.mix = diag(3))
```

Arguments

`classes`	a character vector of either 16, 32, or 512 class names. The character vectors are described in `to_html`.
`rgb.mix`	3 x 3 numeric matrix to remix color channels. Given a N x 3 matrix of numeric RGB colors `rgb`, the colors used in the style sheet will be `rgb %*% rgb.mix`. Out of range values are clipped to the nearest bound of the range.

Value

A character vector that can be used as the contents of a style sheet.

See Also

Other HTML functions: `html_esc()`, `in_html()`, `to_html()`

Examples

```
## Generate some class strings; order matters
classes <- do.call(paste, c(expand.grid(c("fg", "bg"), 0:7), sep="-"))
writeLines(classes[1:4])

## Some Default CSS
css0 <- "span {font-size: 60pt; padding: 10px; display: inline-block}"

## Associated class strings to styles
css1 <- make_styles(classes)
writeLines(css1[1:4])

## Generate SGR-derived HTML, mapping to classes
string <- "\033[43mYellow\033[m\n\033[45mMagenta\033[m\n\033[46mCyan\033[m"
html <- to_html(string, classes=classes)
writeLines(html)

## Combine in a page with styles and display in browser
## Not run:
in_html(html, css=c(css0, css1))

## End(Not run)

## Change CSS by remixing colors, and apply to exact same HTML
mix <- matrix(
  c(
    0, 1, 0,  # red output is green input
    0, 0, 1,  # green output is blue input
    1, 0, 0   # blue output is red input
  ),
  nrow=3, byrow=TRUE
)
css2 <- make_styles(classes, rgb.mix=mix)
## Display in browser: same HTML but colors changed by CSS
## Not run:
in_html(html, css=c(css0, css2))

## End(Not run)
```

nchar_ctl *Control Sequence Aware Version of nchar*

Description

nchar_ctl counts all non *Control Sequence* characters. nzchar_ctl returns TRUE for each input vector element that has non *Control Sequence* sequence characters. By default newlines and other C0 control characters are not counted.

Usage

```
nchar_ctl(
  x,
  type = "chars",
  allowNA = FALSE,
  keepNA = NA,
  ctl = "all",
  warn = getOption("fansi.warn", TRUE),
  strip
)

nzchar_ctl(
  x,
  keepNA = FALSE,
  ctl = "all",
  warn = getOption("fansi.warn", TRUE)
)
```

Arguments

x	a character vector or object that can be coerced to such.
type	character(1L) partial matching c("chars", "width", "graphemes"), although types other than "chars" only work correctly with R >= 3.2.2. See ?nchar.
allowNA	logical: should NA be returned for invalid multibyte strings or "bytes"-encoded strings (rather than throwing an error)?
keepNA	logical: should NA be returned when x is NA? If false, nchar() returns 2, as that is the number of printing characters used when strings are written to output, and nzchar() is TRUE. The default for nchar(), NA, means to use keepNA = TRUE unless type is "width".
ctl	character, which *Control Sequences* should be treated specially. Special treatment is context dependent, and may include detecting them and/or computing their display/character width as zero. For the SGR subset of the ANSI CSI sequences, and OSC hyperlinks, fansi will also parse, interpret, and reapply the sequences as needed. You can modify whether a *Control Sequence* is treated specially with the ctl parameter.

- "nl": newlines.
- "c0": all other "C0" control characters (i.e. 0x01-0x1f, 0x7F), except for newlines and the actual ESC (0x1B) character.
- "sgr": ANSI CSI SGR sequences.
- "csi": all non-SGR ANSI CSI sequences.
- "url": OSC hyperlinks
- "osc": all non-OSC-hyperlink OSC sequences.
- "esc": all other escape sequences.
- "all": all of the above, except when used in combination with any of the above, in which case it means "all but".

`warn` TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions `fansi` makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see `?fansi`). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use `validUTF8` for that).

`strip` character, deprecated in favor of `ctl`.

Details

`nchar_ctl` and `nzchar_ctl` are implemented in statically compiled code, so in particular `nzchar_ctl` will be much faster than the otherwise equivalent `nzchar(strip_ctl(...))`.

These functions will warn if either malformed or escape or UTF-8 sequences are encountered as they may be incorrectly interpreted.

Value

Like `base::nchar`, with *Control Sequences* excluded.

Control and Special Sequences

Control Sequences are non-printing characters or sequences of characters. *Special Sequences* are a subset of the *Control Sequences*, and include CSI SGR sequences which can be used to change rendered appearance of text, and OSC hyperlinks. See `fansi` for details.

Output Stability

Several factors could affect the exact output produced by `fansi` functions across versions of `fansi`, `R`, and/or across systems. **In general it is best not to rely on exact `fansi` output, e.g. by embedding it in tests**.

Width and grapheme calculations depend on locale, Unicode database version, and grapheme processing logic (which is still in development), among other things. For the most part `fansi` (currently) uses the internals of `base::nchar(type='width')`, but there are exceptions and this may change in the future.

How a particular display format is encoded in *Control Sequences* is not guaranteed to be stable across `fansi` versions. Additionally, which *Special Sequences* are re-encoded vs transcribed untouched may change. In general we will strive to keep the rendered appearance stable.

To maximize the odds of getting stable output set `normalize_state` to `TRUE` and `type` to `"chars"` in functions that allow it, and set `term.cap` to a specific set of capabilities.

Graphemes

`fansi` approximates grapheme widths and counts by using heuristics for grapheme breaks that work for most common graphemes, including emoji combining sequences. The heuristic is known to work incorrectly with invalid combining sequences, prepending marks, and sequence interruptors. `fansi` does not provide a full implementation of grapheme break detection to avoid carrying a copy of the Unicode grapheme breaks table, and also because the hope is that R will add the feature eventually itself.

The `utf8` package provides a conforming grapheme parsing implementation.

Note

The `keepNA` parameter is ignored for R < 3.2.2.

See Also

`?fansi` for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, `unhandled_ctl` for detecting bad control sequences.

Examples

```
nchar_ctl("\033[31m123\a\r")
## with some wide characters
cn.string <-  sprintf("\033[31m%s\a\r", "\u4E00\u4E01\u4E03")
nchar_ctl(cn.string)
nchar_ctl(cn.string, type='width')

## Remember newlines are not counted by default
nchar_ctl("\t\n\r")

## The 'c0' value for the `ctl` argument does not include
## newlines.
nchar_ctl("\t\n\r", ctl="c0")
nchar_ctl("\t\n\r", ctl=c("c0", "nl"))

## The _sgr flavor only treats SGR sequences as zero width
nchar_sgr("\033[31m123")
nchar_sgr("\t\n\n123")

## All of the following are Control Sequences or C0 controls
nzchar_ctl("\n\033[42;31m\033[123P\a")
```

`normalize_state` *Normalize CSI and OSC Sequences*

Description

Re-encodes SGR and OSC encoded URL sequences into a unique decomposed form. Strings containing semantically identical SGR and OSC sequences that are encoded differently should compare equal after normalization.

Usage

```
normalize_state(
  x,
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  carry = getOption("fansi.carry", FALSE)
)
```

Arguments

`x`	a character vector or object that can be coerced to such.
`warn`	TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions `fansi` makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see `?fansi`). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use `validUTF8` for that).
`term.cap`	character a vector of the capabilities of the terminal, can be any combination of "bright" (SGR codes 90-97, 100-107), "256" (SGR codes starting with "38;5" or "48;5"), "truecolor" (SGR codes starting with "38;2" or "48;2"), and "all". "all" behaves as it does for the `ctl` parameter: "all" combined with any other value means all terminal capabilities except that one. `fansi` will warn if it encounters SGR codes that exceed the terminal capabilities specified (see `term_cap_test` for details). In versions prior to 1.0, `fansi` would also skip exceeding SGRs entirely instead of interpreting them. You may add the string "old" to any otherwise valid `term.cap` spec to restore the pre 1.0 behavior. "old" will not interact with "all" the way other valid values for this parameter do.
`carry`	TRUE, FALSE (default), or a scalar string, controls whether to interpret the character vector as a "single document" (TRUE or string) or as independent elements (FALSE). In "single document" mode, active state at the end of an input element is considered active at the beginning of the next vector element, simulating what happens with a document with active state at the end of a line. If FALSE each vector element is interpreted as if there were no active state when it begins. If character, then the active state at the end of the `carry` string is carried into the first element of `x` (see "Replacement Functions" for differences there).

The carried state is injected in the interstice between an imaginary zeroeth character and the first character of a vector element. See the "Position Semantics" section of substr_ctl and the "State Interactions" section of ?fansi for details. Except for strwrap_ctl where NA is treated as the string "NA", carry will cause NAs in inputs to propagate through the remaining vector elements.

Details

Each compound SGR sequence is broken up into individual tokens, superfluous tokens are removed, and the SGR reset sequence "ESC[0m" (or "ESC[m") is replaced by the closing codes for whatever SGR styles are active at the point in the string in which it appears.

Unrecognized SGR codes will be dropped from the output with a warning. The specific order of SGR codes associated with any given SGR sequence is not guaranteed to remain the same across different versions of fansi, but should remain unchanged except for the addition of previously uninterpreted codes to the list of interpretable codes. There is no special significance to the order the SGR codes are emitted in other than it should be consistent for any given SGR state. URLs adjacent to SGR codes are always emitted after the SGR codes irrespective of what side they were on originally.

OSC encoded URL sequences are always terminated by "ESC]\", and those between abutting URLs are omitted. Identical abutting URLs are merged. In order for URLs to be considered identical both the URL and the "id" parameter must be specified and be the same. OSC URL parameters other than "id" are dropped with a warning.

The underlying assumption is that each element in the vector is unaffected by SGR or OSC URLs in any other element or elsewhere. This may lead to surprising outcomes if these assumptions are untrue (see examples). You may adjust this assumption with the carry parameter.

Normalization was implemented primarily for better compatibility with crayon which emits SGR codes individually and assumes that each opening code is paired up with its specific closing code, but it can also be used to reduce the probability that strings processed with future versions of fansi will produce different results than the current version.

Value

x, with all SGRs normalized.

See Also

?fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, unhandled_ctl for detecting bad control sequences.

Examples

```
normalize_state("hello\033[42;33m world")
normalize_state("hello\033[42;33m world\033[m")
normalize_state("\033[4mhello\033[42;33m world\033[m")

## Superflous codes removed
normalize_state("\033[31;32mhello\033[m")      # only last color prevails
normalize_state("\033[31\033[32mhello\033[m")  # only last color prevails
normalize_state("\033[31mhe\033[49mllo\033[m") # unused closing
```

```
## Equivalent normalized sequences compare identical
identical(
  normalize_state("\033[31;32mhello\033[m"),
  normalize_state("\033[31mhe\033[49mllo\033[m")
)
## External SGR will defeat normalization, unless we `carry` it
red <- "\033[41m"
writeLines(
  c(
    paste(red, "he\033[0mllo", "\033[0m"),
    paste(red, normalize_state("he\033[0mllo"), "\033[0m"),
    paste(red, normalize_state("he\033[0mllo", carry=red), "\033[0m")
) )
```

`set_knit_hooks` *Set an Output Hook Convert Control Sequences to HTML in Rmarkdown*

Description

This is a convenience function designed for use within an `rmarkdown` document. It overrides the `knitr` output hooks by using `knitr::knit_hooks$set`. It replaces the hooks with ones that convert *Control Sequences* into HTML. In addition to replacing the hook functions, this will output a <STYLE> HTML block to stdout. These two actions are side effects as a result of which R chunks in the `rmarkdown` document that contain CSI SGR are shown in their HTML equivalent form.

Usage

```
set_knit_hooks(
  hooks,
  which = "output",
 proc.fun = function(x, class) html_code_block(to_html(html_esc(x)), class = class),
  class = sprintf("fansi fansi-%s", which),
  style = getOption("fansi.css", dflt_css()),
  split.nl = FALSE,
  .test = FALSE
)
```

Arguments

`hooks`	list, this should the be `knitr::knit_hooks` object; we require you pass this to avoid a run-time dependency on `knitr`.
`which`	character vector with the names of the hooks that should be replaced, defaults to 'output', but can also contain values 'message', 'warning', and 'error'.
`proc.fun`	function that will be applied to output that contains CSI SGR sequences. Should accept parameters `x` and `class`, where `x` is the output, and `class` is the CSS class that should be applied to the <PRE><CODE> blocks the output will be placed in.

class	character the CSS class to give the output chunks. Each type of output chunk specified in which will be matched position-wise to the classes specified here. This vector should be the same length as which.
style	character a vector of CSS styles; these will be output inside HTML >STYLE< tags as a side effect. The default value is designed to ensure that there is no visible gap in background color with lines with height 1.5 (as is the default setting in rmarkdown documents v1.1).
split.nl	TRUE or FALSE (default), set to TRUE to split input strings by any newlines they may contain to avoid any newlines inside SPAN tags created by to_html(). Some markdown->html renders can be configured to convert embedded newlines into line breaks, which may lead to a doubling of line breaks. With the default proc.fun the split strings are recombined by html_code_block(), but if you provide your own proc.fun you'll need to account for the possibility that the character vector it receives will have a different number of elements than the chunk output. This argument only has an effect if chunk output contains CSI SGR sequences.
.test	TRUE or FALSE, for internal testing use only.

Details

The replacement hook function tests for the presence of CSI SGR sequences in chunk output with has_ctl, and if it is detected then processes it with the user provided proc.fun. Chunks that do not contain CSI SGR are passed off to the previously set hook function. The default proc.fun will run the output through html_esc, to_html, and finally html_code_block.

If you require more control than this function provides you can set the knitr hooks manually with knitr::knit_hooks$set. If you are seeing your output gaining extra line breaks, look at the split.nl option.

Value

named list with the prior output hooks for each of which.

Note

Since we do not formally import the knitr functions we do not guarantee that this function will always work properly with knitr / rmarkdown.

See Also

has_ctl, to_html, html_esc, html_code_block, knitr output hooks, embedding CSS in Rmd, and the vignette vignette(package='fansi', 'sgr-in-rmd').

Examples

```
## Not run:
## The following should be done within an `rmarkdown` document chunk with
## chunk option `results` set to 'asis' and the chunk option `comment` set
## to ''.
```

```
```{r comment="", results='asis', echo=FALSE}
## Change the "output" hook to handle ANSI CSI SGR

old.hooks <- set_knit_hooks(knitr::knit_hooks)

## Do the same with the warning, error, and message, and add styles for
## them (alternatively we could have done output as part of this call too)

styles <- c(
  getOption('fansi.style', dflt_css()),  # default style
  "PRE.fansi CODE {background-color: transparent;}",
  "PRE.fansi-error {background-color: #DD5555;}",
  "PRE.fansi-warning {background-color: #DDDD55;}",
  "PRE.fansi-message {background-color: #EEEEEE;}"
)
old.hooks <- c(
  old.hooks,
  fansi::set_knit_hooks(
    knitr::knit_hooks,
    which=c('warning', 'error', 'message'),
    style=styles
) )
```
You may restore old hooks with the following chunk

Restore Hooks
```{r}
do.call(knitr::knit_hooks$set, old.hooks)
```

End(Not run)
```

---

sgr_256 *Show 8 Bit CSI SGR Colors*

Description

Generates text with each 8 bit SGR code (e.g. the "###" in "38;5;###") with the background colored by itself, and the foreground in a contrasting color and interesting color (we sacrifice some contrast for interest as this is intended for demo rather than reference purposes).

Usage

```
sgr_256()
```

Value

character vector with SGR codes with background color set as themselves.

See Also

make_styles().

Examples

```
writeLines(sgr_256())
```

---

state_at_end *Utilities for Managing CSI and OSC State In Strings*

Description

state_at_end reads through strings computing the accumulated SGR and OSC hyperlinks, and outputs the active state at the end of them. close_state produces the sequence that closes any SGR active and OSC hyperlinks at the end of each input string. If normalize = FALSE (default), it will emit the reset code "ESC[0m" if any SGR is present. It is more interesting for closing SGRs if normalize = TRUE. Unlike state_at_end and other functions close_state has no concept of carry: it will only emit closing sequences for states explicitly active at the end of a string.

Usage

```
state_at_end(
  x,
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  normalize = getOption("fansi.normalize", FALSE),
  carry = getOption("fansi.carry", FALSE)
)

close_state(
  x,
  warn = getOption("fansi.warn", TRUE),
  normalize = getOption("fansi.normalize", FALSE)
)
```

Arguments

x	a character vector or object that can be coerced to such.
warn	TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions fansi makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see ?fansi). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use validUTF8 for that).

`term.cap` character a vector of the capabilities of the terminal, can be any combination of "bright" (SGR codes 90-97, 100-107), "256" (SGR codes starting with "38;5" or "48;5"), "truecolor" (SGR codes starting with "38;2" or "48;2"), and "all". "all" behaves as it does for the `ctl` parameter: "all" combined with any other value means all terminal capabilities except that one. `fansi` will warn if it encounters SGR codes that exceed the terminal capabilities specified (see `term_cap_test` for details). In versions prior to 1.0, `fansi` would also skip exceeding SGRs entirely instead of interpreting them. You may add the string "old" to any otherwise valid `term.cap` spec to restore the pre 1.0 behavior. "old" will not interact with "all" the way other valid values for this parameter do.

`normalize` TRUE or FALSE (default) whether SGR sequence should be normalized out such that there is one distinct sequence for each SGR code. normalized strings will occupy more space (e.g. "\033[31;42m" becomes "\033[31m\033[42m"), but will work better with code that assumes each SGR code will be in its own escape as `crayon` does.

`carry` TRUE, FALSE (default), or a scalar string, controls whether to interpret the character vector as a "single document" (TRUE or string) or as independent elements (FALSE). In "single document" mode, active state at the end of an input element is considered active at the beginning of the next vector element, simulating what happens with a document with active state at the end of a line. If FALSE each vector element is interpreted as if there were no active state when it begins. If character, then the active state at the end of the `carry` string is carried into the first element of `x` (see "Replacement Functions" for differences there). The carried state is injected in the interstice between an imaginary zeroeth character and the first character of a vector element. See the "Position Semantics" section of `substr_ctl` and the "State Interactions" section of `?fansi` for details. Except for `strwrap_ctl` where `NA` is treated as the string `"NA"`, `carry` will cause `NA`s in inputs to propagate through the remaining vector elements.

Value

character vector same length as `x`.

Control and Special Sequences

Control Sequences are non-printing characters or sequences of characters. *Special Sequences* are a subset of the *Control Sequences*, and include CSI SGR sequences which can be used to change rendered appearance of text, and OSC hyperlinks. See `fansi` for details.

Output Stability

Several factors could affect the exact output produced by `fansi` functions across versions of `fansi`, `R`, and/or across systems. **In general it is best not to rely on exact `fansi` output, e.g. by embedding it in tests**.

Width and grapheme calculations depend on locale, Unicode database version, and grapheme processing logic (which is still in development), among other things. For the most part `fansi` (currently) uses the internals of `base::nchar(type='width')`, but there are exceptions and this may change in the future.

How a particular display format is encoded in *Control Sequences* is not guaranteed to be stable across `fansi` versions. Additionally, which *Special Sequences* are re-encoded vs transcribed untouched may change. In general we will strive to keep the rendered appearance stable.

To maximize the odds of getting stable output set `normalize_state` to `TRUE` and `type` to `"chars"` in functions that allow it, and set `term.cap` to a specific set of capabilities.

See Also

`?fansi` for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, `unhandled_ctl` for detecting bad control sequences.

Examples

```
x <- c("\033[44mhello", "\033[33mworld")
state_at_end(x)
state_at_end(x, carry=TRUE)
(close <- close_state(state_at_end(x, carry=TRUE), normalize=TRUE))
writeLines(paste0(x, close, " no style"))
```

---

strip_ctl *Strip Control Sequences*

Description

Removes *Control Sequences* from strings. By default it will strip all known *Control Sequences*, including CSI/OSC sequences, two character sequences starting with ESC, and all C0 control characters, including newlines. You can fine tune this behavior with the `ctl` parameter.

Usage

```
strip_ctl(x, ctl = "all", warn = getOption("fansi.warn", TRUE), strip)
```

Arguments

`x`	a character vector or object that can be coerced to such.
`ctl`	character, any combination of the following values (see details):

- "nl": strip newlines.
- "c0": strip all other "C0" control characters (i.e. x01-x1f, x7F), except for newlines and the actual ESC character.
- "sgr": strip ANSI CSI SGR sequences.
- "csi": strip all non-SGR csi sequences.
- "esc": strip all other escape sequences.
- "all": all of the above, except when used in combination with any of the above, in which case it means "all but" (see details).

warn — TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions fansi makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see ?fansi). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use validUTF8 for that).

strip — character, deprecated in favor of ctl.

Details

The ctl value contains the names of **non-overlapping** subsets of the known *Control Sequences* (e.g. "csi" does not contain "sgr", and "c0" does not contain newlines). The one exception is "all" which means strip every known sequence. If you combine "all" with any other options then everything **but** those options will be stripped.

Value

character vector of same length as x with ANSI escape sequences stripped

Note

Non-ASCII strings are converted to and returned in UTF-8 encoding.

See Also

?fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, unhandled_ctl for detecting bad control sequences.

Examples

```
string <- "hello\033k\033[45p world\n\033[31mgoodbye\a moon"
strip_ctl(string)
strip_ctl(string, c("nl", "c0", "sgr", "csi", "esc")) # equivalently
strip_ctl(string, "sgr")
strip_ctl(string, c("c0", "esc"))

## everything but C0 controls, we need to specify "nl"
## in addition to "c0" since "nl" is not part of "c0"
## as far as the `strip` argument is concerned
strip_ctl(string, c("all", "nl", "c0"))
```

strsplit_ctl *Control Sequence Aware Version of strsplit*

Description

A drop-in replacement for base::strsplit.

Usage

```
strsplit_ctl(
  x,
  split,
  fixed = FALSE,
  perl = FALSE,
  useBytes = FALSE,
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  ctl = "all",
  normalize = getOption("fansi.normalize", FALSE),
  carry = getOption("fansi.carry", FALSE),
  terminate = getOption("fansi.terminate", TRUE)
)
```

Arguments

x	a character vector, or, unlike base::strsplit an object that can be coerced to character.
split	character vector (or object which can be coerced to such) containing regular expression(s) (unless fixed = TRUE) to use for splitting. If empty matches occur, in particular if split has length 0, x is split into single characters. If split has length greater than 1, it is re-cycled along x.
fixed	logical. If TRUE match split exactly, otherwise use regular expressions. Has priority over perl.
perl	logical. Should Perl-compatible regexps be used?
useBytes	logical. If TRUE the matching is done byte-by-byte rather than character-by-character, and inputs with marked encodings are not converted. This is forced (with a warning) if any input is found which is marked as "bytes" (see Encoding).
warn	TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions fansi makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see ?fansi). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use validUTF8 for that).

`term.cap` character a vector of the capabilities of the terminal, can be any combination of "bright" (SGR codes 90-97, 100-107), "256" (SGR codes starting with "38;5" or "48;5"), "truecolor" (SGR codes starting with "38;2" or "48;2"), and "all". "all" behaves as it does for the `ctl` parameter: "all" combined with any other value means all terminal capabilities except that one. `fansi` will warn if it encounters SGR codes that exceed the terminal capabilities specified (see `term_cap_test` for details). In versions prior to 1.0, `fansi` would also skip exceeding SGRs entirely instead of interpreting them. You may add the string "old" to any otherwise valid `term.cap` spec to restore the pre 1.0 behavior. "old" will not interact with "all" the way other valid values for this parameter do.

`ctl` character, which *Control Sequences* should be treated specially. Special treatment is context dependent, and may include detecting them and/or computing their display/character width as zero. For the SGR subset of the ANSI CSI sequences, and OSC hyperlinks, `fansi` will also parse, interpret, and reapply the sequences as needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter.

- "nl": newlines.
- "c0": all other "C0" control characters (i.e. 0x01-0x1f, 0x7F), except for newlines and the actual ESC (0x1B) character.
- "sgr": ANSI CSI SGR sequences.
- "csi": all non-SGR ANSI CSI sequences.
- "url": OSC hyperlinks
- "osc": all non-OSC-hyperlink OSC sequences.
- "esc": all other escape sequences.
- "all": all of the above, except when used in combination with any of the above, in which case it means "all but".

`normalize` TRUE or FALSE (default) whether SGR sequence should be normalized out such that there is one distinct sequence for each SGR code. normalized strings will occupy more space (e.g. "\033[31;42m" becomes "\033[31m\033[42m"), but will work better with code that assumes each SGR code will be in its own escape as `crayon` does.

`carry` TRUE, FALSE (default), or a scalar string, controls whether to interpret the character vector as a "single document" (TRUE or string) or as independent elements (FALSE). In "single document" mode, active state at the end of an input element is considered active at the beginning of the next vector element, simulating what happens with a document with active state at the end of a line. If FALSE each vector element is interpreted as if there were no active state when it begins. If character, then the active state at the end of the `carry` string is carried into the first element of `x` (see "Replacement Functions" for differences there). The carried state is injected in the interstice between an imaginary zeroeth character and the first character of a vector element. See the "Position Semantics" section of `substr_ctl` and the "State Interactions" section of `?fansi` for details. Except for `strwrap_ctl` where `NA` is treated as the string `"NA"`, `carry` will cause `NA`s in inputs to propagate through the remaining vector elements.

`terminate` TRUE (default) or FALSE whether substrings should have active state closed to avoid it bleeding into other strings they may be prepended onto. This does not

stop state from carrying if `carry = TRUE`. See the "State Interactions" section of `?fansi` for details.

Details

This function works by computing the position of the split points after removing *Control Sequences*, and uses those positions in conjunction with `substr_ctl` to extract the pieces. This concept is borrowed from `crayon::col_strsplit`. An important implication of this is that you cannot split by *Control Sequences* that are being treated as *Control Sequences*. You can however limit which control sequences are treated specially via the `ctl` parameters (see examples).

Value

Like `base::strsplit`, with *Control Sequences* excluded.

Control and Special Sequences

Control Sequences are non-printing characters or sequences of characters. *Special Sequences* are a subset of the *Control Sequences*, and include CSI SGR sequences which can be used to change rendered appearance of text, and OSC hyperlinks. See `fansi` for details.

Output Stability

Several factors could affect the exact output produced by `fansi` functions across versions of `fansi`, `R`, and/or across systems. **In general it is best not to rely on exact `fansi` output, e.g. by embedding it in tests**.

Width and grapheme calculations depend on locale, Unicode database version, and grapheme processing logic (which is still in development), among other things. For the most part `fansi` (currently) uses the internals of `base::nchar(type='width')`, but there are exceptions and this may change in the future.

How a particular display format is encoded in *Control Sequences* is not guaranteed to be stable across `fansi` versions. Additionally, which *Special Sequences* are re-encoded vs transcribed untouched may change. In general we will strive to keep the rendered appearance stable.

To maximize the odds of getting stable output set `normalize_state` to `TRUE` and `type` to `"chars"` in functions that allow it, and set `term.cap` to a specific set of capabilities.

Bidirectional Text

`fansi` is unaware of text directionality and operates as if all strings are left to right (LTR). Using `fansi` function with strings that contain mixed direction scripts (i.e. both LTR and RTL) may produce undesirable results.

Note

The split positions are computed after both `x` and `split` are converted to UTF-8.

Non-ASCII strings are converted to and returned in UTF-8 encoding. Width calculations will not work properly in R < 3.2.2.

See Also

?fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, normalize_state for more details on what the normalize parameter does, state_at_end to compute active state at the end of strings, close_state to compute the sequence required to close active state.

Examples

```
strsplit_ctl("\033[31mhello\033[42m world!", " ")

## Splitting by newlines does not work as they are _Control
## Sequences_, but we can use `ctl` to treat them as ordinary
strsplit_ctl("\033[31mhello\033[42m\nworld!", "\n")
strsplit_ctl("\033[31mhello\033[42m\nworld!", "\n", ctl=c("all", "nl"))
```

---

strtrim_ctl *Control Sequence Aware Version of strtrim*

Description

A drop in replacement for base::strtrim, with the difference that all C0 control characters such as newlines, carriage returns, etc., are always treated as zero width, whereas in base it may vary with platform / R version.

Usage

```
strtrim_ctl(
  x,
  width,
  warn = getOption("fansi.warn", TRUE),
  ctl = "all",
  normalize = getOption("fansi.normalize", FALSE),
  carry = getOption("fansi.carry", FALSE),
  terminate = getOption("fansi.terminate", TRUE)
)

strtrim2_ctl(
  x,
  width,
  warn = getOption("fansi.warn", TRUE),
  tabs.as.spaces = getOption("fansi.tabs.as.spaces", FALSE),
  tab.stops = getOption("fansi.tab.stops", 8L),
  ctl = "all",
  normalize = getOption("fansi.normalize", FALSE),
  carry = getOption("fansi.carry", FALSE),
  terminate = getOption("fansi.terminate", TRUE)
)
```

Arguments

x	a character vector, or an object which can be coerced to a character vector by as.character.
width	Positive integer values: recycled to the length of x.
warn	TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions fansi makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see ?fansi). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use validUTF8 for that).
ctl	character, which *Control Sequences* should be treated specially. Special treatment is context dependent, and may include detecting them and/or computing their display/character width as zero. For the SGR subset of the ANSI CSI sequences, and OSC hyperlinks, fansi will also parse, interpret, and reapply the sequences as needed. You can modify whether a *Control Sequence* is treated specially with the ctl parameter. • "nl": newlines. • "c0": all other "C0" control characters (i.e. 0x01-0x1f, 0x7F), except for newlines and the actual ESC (0x1B) character. • "sgr": ANSI CSI SGR sequences. • "csi": all non-SGR ANSI CSI sequences. • "url": OSC hyperlinks • "osc": all non-OSC-hyperlink OSC sequences. • "esc": all other escape sequences. • "all": all of the above, except when used in combination with any of the above, in which case it means "all but".
normalize	TRUE or FALSE (default) whether SGR sequence should be normalized out such that there is one distinct sequence for each SGR code. normalized strings will occupy more space (e.g. "\033[31;42m" becomes "\033[31m\033[42m"), but will work better with code that assumes each SGR code will be in its own escape as crayon does.
carry	TRUE, FALSE (default), or a scalar string, controls whether to interpret the character vector as a "single document" (TRUE or string) or as independent elements (FALSE). In "single document" mode, active state at the end of an input element is considered active at the beginning of the next vector element, simulating what happens with a document with active state at the end of a line. If FALSE each vector element is interpreted as if there were no active state when it begins. If character, then the active state at the end of the carry string is carried into the first element of x (see "Replacement Functions" for differences there). The carried state is injected in the interstice between an imaginary zeroeth character and the first character of a vector element. See the "Position Semantics" section of substr_ctl and the "State Interactions" section of ?fansi for details. Except for strwrap_ctl where NA is treated as the string "NA", carry will cause NAs in inputs to propagate through the remaining vector elements.

terminate — TRUE (default) or FALSE whether substrings should have active state closed to avoid it bleeding into other strings they may be prepended onto. This does not stop state from carrying if carry = TRUE. See the "State Interactions" section of ?fansi for details.

tabs.as.spaces — FALSE (default) or TRUE, whether to convert tabs to spaces. This can only be set to TRUE if strip.spaces is FALSE.

tab.stops — integer(1:n) indicating position of tab stops to use when converting tabs to spaces. If there are more tabs in a line than defined tab stops the last tab stop is re-used. For the purposes of applying tab stops, each input line is considered a line and the character count begins from the beginning of the input line.

Details

strtrim2_ctl adds the option of converting tabs to spaces before trimming. This is the only difference between strtrim_ctl and strtrim2_ctl.

Value

Like base::strtrim, except that *Control Sequences* are treated as zero width.

Note

Non-ASCII strings are converted to and returned in UTF-8 encoding. Width calculations will not work properly in R < 3.2.2.

See Also

?fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, normalize_state for more details on what the normalize parameter does, state_at_end to compute active state at the end of strings, close_state to compute the sequence required to close active state.

Examples

```
strtrim_ctl("\033[42mHello world\033[m", 6)
```

---

strwrap_ctl *Control Sequence Aware Version of strwrap*

Description

Wraps strings to a specified width accounting for *Control Sequences*. strwrap_ctl is intended to emulate strwrap closely except with respect to the *Control Sequences* (see details for other minor differences), while strwrap2_ctl adds features and changes the processing of whitespace. strwrap_ctl is faster than strwrap.

Usage

```
strwrap_ctl(
  x,
  width = 0.9 * getOption("width"),
  indent = 0,
  exdent = 0,
  prefix = "",
  simplify = TRUE,
  initial = prefix,
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  ctl = "all",
  normalize = getOption("fansi.normalize", FALSE),
  carry = getOption("fansi.carry", FALSE),
  terminate = getOption("fansi.terminate", TRUE)
)

strwrap2_ctl(
  x,
  width = 0.9 * getOption("width"),
  indent = 0,
  exdent = 0,
  prefix = "",
  simplify = TRUE,
  initial = prefix,
  wrap.always = FALSE,
  pad.end = "",
  strip.spaces = !tabs.as.spaces,
  tabs.as.spaces = getOption("fansi.tabs.as.spaces", FALSE),
  tab.stops = getOption("fansi.tab.stops", 8L),
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  ctl = "all",
  normalize = getOption("fansi.normalize", FALSE),
  carry = getOption("fansi.carry", FALSE),
  terminate = getOption("fansi.terminate", TRUE)
)
```

Arguments

x	a character vector, or an object which can be converted to a character vector by as.character.
width	a positive integer giving the target column for wrapping lines in the output.
indent	a non-negative integer giving the indentation of the first line in a paragraph.
exdent	a non-negative integer specifying the indentation of subsequent lines in paragraphs.

`prefix, initial`
: a character string to be used as prefix for each line except the first, for which `initial` is used.

`simplify`
: a logical. If `TRUE`, the result is a single character vector of line text; otherwise, it is a list of the same length as `x` the elements of which are character vectors of line text obtained from the corresponding element of `x`. (Hence, the result in the former case is obtained by unlisting that of the latter.)

`warn`
: TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions `fansi` makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see `?fansi`). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use `validUTF8` for that).

`term.cap`
: character a vector of the capabilities of the terminal, can be any combination of "bright" (SGR codes 90-97, 100-107), "256" (SGR codes starting with "38;5" or "48;5"), "truecolor" (SGR codes starting with "38;2" or "48;2"), and "all". "all" behaves as it does for the `ctl` parameter: "all" combined with any other value means all terminal capabilities except that one. `fansi` will warn if it encounters SGR codes that exceed the terminal capabilities specified (see `term_cap_test` for details). In versions prior to 1.0, `fansi` would also skip exceeding SGRs entirely instead of interpreting them. You may add the string "old" to any otherwise valid `term.cap` spec to restore the pre 1.0 behavior. "old" will not interact with "all" the way other valid values for this parameter do.

`ctl`
: character, which *Control Sequences* should be treated specially. Special treatment is context dependent, and may include detecting them and/or computing their display/character width as zero. For the SGR subset of the ANSI CSI sequences, and OSC hyperlinks, `fansi` will also parse, interpret, and reapply the sequences as needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter.

  - "nl": newlines.
  - "c0": all other "C0" control characters (i.e. 0x01-0x1f, 0x7F), except for newlines and the actual ESC (0x1B) character.
  - "sgr": ANSI CSI SGR sequences.
  - "csi": all non-SGR ANSI CSI sequences.
  - "url": OSC hyperlinks
  - "osc": all non-OSC-hyperlink OSC sequences.
  - "esc": all other escape sequences.
  - "all": all of the above, except when used in combination with any of the above, in which case it means "all but".

`normalize`
: TRUE or FALSE (default) whether SGR sequence should be normalized out such that there is one distinct sequence for each SGR code. normalized strings will occupy more space (e.g. "\033[31;42m" becomes "\033[31m\033[42m"), but will work better with code that assumes each SGR code will be in its own escape as `crayon` does.

`carry` TRUE, FALSE (default), or a scalar string, controls whether to interpret the character vector as a "single document" (TRUE or string) or as independent elements (FALSE). In "single document" mode, active state at the end of an input element is considered active at the beginning of the next vector element, simulating what happens with a document with active state at the end of a line. If FALSE each vector element is interpreted as if there were no active state when it begins. If character, then the active state at the end of the `carry` string is carried into the first element of `x` (see "Replacement Functions" for differences there). The carried state is injected in the interstice between an imaginary zeroeth character and the first character of a vector element. See the "Position Semantics" section of `substr_ctl` and the "State Interactions" section of `?fansi` for details. Except for `strwrap_ctl` where `NA` is treated as the string `"NA"`, `carry` will cause `NA`s in inputs to propagate through the remaining vector elements.

`terminate` TRUE (default) or FALSE whether substrings should have active state closed to avoid it bleeding into other strings they may be prepended onto. This does not stop state from carrying if `carry = TRUE`. See the "State Interactions" section of `?fansi` for details.

`wrap.always` TRUE or FALSE (default), whether to hard wrap at requested width if no word breaks are detected within a line. If set to TRUE then `width` must be at least 2.

`pad.end` character(1L), a single character to use as padding at the end of each line until the line is `width` wide. This must be a printable ASCII character or an empty string (default). If you set it to an empty string the line remains unpadded.

`strip.spaces` TRUE (default) or FALSE, if TRUE, extraneous white spaces (spaces, newlines, tabs) are removed in the same way as base::strwrap does. When FALSE, whitespaces are preserved, except for newlines as those are implicit boundaries between output vector elements.

`tabs.as.spaces` FALSE (default) or TRUE, whether to convert tabs to spaces. This can only be set to TRUE if `strip.spaces` is FALSE.

`tab.stops` integer(1:n) indicating position of tab stops to use when converting tabs to spaces. If there are more tabs in a line than defined tab stops the last tab stop is re-used. For the purposes of applying tab stops, each input line is considered a line and the character count begins from the beginning of the input line.

Details

`strwrap2_ctl` can convert tabs to spaces, pad strings up to `width`, and hard-break words if single words are wider than `width`.

Unlike base::strwrap, both these functions will translate any non-ASCII strings to UTF-8 and return them in UTF-8. Additionally, invalid UTF-8 always causes errors, and `prefix` and `indent` must be scalar.

When replacing tabs with spaces the tabs are computed relative to the beginning of the input line, not the most recent wrap point. Additionally,`indent`, `exdent`, `initial`, and `prefix` will be ignored when computing tab positions.

Value

A character vector, or list of character vectors if `simplify` is false.

Control and Special Sequences

Control Sequences are non-printing characters or sequences of characters. *Special Sequences* are a subset of the *Control Sequences*, and include CSI SGR sequences which can be used to change rendered appearance of text, and OSC hyperlinks. See `fansi` for details.

Graphemes

`fansi` approximates grapheme widths and counts by using heuristics for grapheme breaks that work for most common graphemes, including emoji combining sequences. The heuristic is known to work incorrectly with invalid combining sequences, prepending marks, and sequence interruptors. `fansi` does not provide a full implementation of grapheme break detection to avoid carrying a copy of the Unicode grapheme breaks table, and also because the hope is that R will add the feature eventually itself.

The `utf8` package provides a conforming grapheme parsing implementation.

Output Stability

Several factors could affect the exact output produced by `fansi` functions across versions of `fansi`, `R`, and/or across systems. **In general it is best not to rely on exact `fansi` output, e.g. by embedding it in tests**.

Width and grapheme calculations depend on locale, Unicode database version, and grapheme processing logic (which is still in development), among other things. For the most part `fansi` (currently) uses the internals of `base::nchar(type='width')`, but there are exceptions and this may change in the future.

How a particular display format is encoded in *Control Sequences* is not guaranteed to be stable across `fansi` versions. Additionally, which *Special Sequences* are re-encoded vs transcribed untouched may change. In general we will strive to keep the rendered appearance stable.

To maximize the odds of getting stable output set `normalize_state` to `TRUE` and `type` to `"chars"` in functions that allow it, and set `term.cap` to a specific set of capabilities.

Bidirectional Text

`fansi` is unaware of text directionality and operates as if all strings are left to right (LTR). Using `fansi` function with strings that contain mixed direction scripts (i.e. both LTR and RTL) may produce undesirable results.

Note

Non-ASCII strings are converted to and returned in UTF-8 encoding. Width calculations will not work properly in R < 3.2.2.

For the `strwrap*` functions the `carry` parameter affects whether styles are carried across *input* vector elements. Styles always carry within a single wrapped vector element (e.g. if one of the input elements gets wrapped into three lines, the styles will carry through those three lines even if `carry=FALSE`, but not across input vector elements).

See Also

?fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, normalize_state for more details on what the normalize parameter does, state_at_end to compute active state at the end of strings, close_state to compute the sequence required to close active state.

Examples

```
hello.1 <- "hello \033[41mred\033[49m world"
hello.2 <- "hello\t\033[41mred\033[49m\tworld"

strwrap_ctl(hello.1, 12)
strwrap_ctl(hello.2, 12)

## In default mode strwrap2_ctl is the same as strwrap_ctl
strwrap2_ctl(hello.2, 12)

## But you can leave whitespace unchanged, `warn`
## set to false as otherwise tabs causes warning
strwrap2_ctl(hello.2, 12, strip.spaces=FALSE, warn=FALSE)

## And convert tabs to spaces
strwrap2_ctl(hello.2, 12, tabs.as.spaces=TRUE)

## If your display has 8 wide tab stops the following two
## outputs should look the same
writeLines(strwrap2_ctl(hello.2, 80, tabs.as.spaces=TRUE))
writeLines(hello.2)

## tab stops are NOT auto-detected, but you may provide
## your own
strwrap2_ctl(hello.2, 12, tabs.as.spaces=TRUE, tab.stops=c(6, 12))

## You can also force padding at the end to equal width
writeLines(strwrap2_ctl("hello how are you today", 10, pad.end="."))

## And a more involved example where we read the
## NEWS file, color it line by line, wrap it to
## 25 width and display some of it in 3 columns
## (works best on displays that support 256 color
## SGR sequences)

NEWS <- readLines(file.path(R.home('doc'), 'NEWS'))
NEWS.C <- fansi_lines(NEWS, step=2)  # color each line
W <- strwrap2_ctl(NEWS.C, 25, pad.end=" ", wrap.always=TRUE)
writeLines(c("", paste(W[1:20], W[100:120], W[200:220]), ""))
```

---

substr_ctl *Control Sequence Aware Version of substr*

Description

substr_ctl is a drop-in replacement for substr. Performance is slightly slower than substr, and more so for type = 'width'. Special *Control Sequences* will be included in the substrings to reflect their format when as it was when part of the source string. substr2_ctl adds the ability to extract substrings based on grapheme count or display width in addition to the normal character width, as well as several other options.

Usage

```
substr_ctl(
  x,
  start,
  stop,
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  ctl = "all",
  normalize = getOption("fansi.normalize", FALSE),
  carry = getOption("fansi.carry", FALSE),
  terminate = getOption("fansi.terminate", TRUE)
)

substr2_ctl(
  x,
  start,
  stop,
  type = "chars",
  round = "start",
  tabs.as.spaces = getOption("fansi.tabs.as.spaces", FALSE),
  tab.stops = getOption("fansi.tab.stops", 8L),
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  ctl = "all",
  normalize = getOption("fansi.normalize", FALSE),
  carry = getOption("fansi.carry", FALSE),
  terminate = getOption("fansi.terminate", TRUE)
)

substr_ctl(
  x,
  start,
  stop,
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  ctl = "all",
  normalize = getOption("fansi.normalize", FALSE),
  carry = getOption("fansi.carry", FALSE),
  terminate = getOption("fansi.terminate", TRUE)
) <- value
```

```
substr2_ctl(
  x,
  start,
  stop,
  type = "chars",
  round = "start",
  tabs.as.spaces = getOption("fansi.tabs.as.spaces", FALSE),
  tab.stops = getOption("fansi.tab.stops", 8L),
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  ctl = "all",
  normalize = getOption("fansi.normalize", FALSE),
  carry = getOption("fansi.carry", FALSE),
  terminate = getOption("fansi.terminate", TRUE)
) <- value
```

Arguments

x	a character vector or object that can be coerced to such.
start	integer. The first element to be extracted or replaced.
stop	integer. The first element to be extracted or replaced.
warn	TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions fansi makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see ?fansi). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use validUTF8 for that).
term.cap	character a vector of the capabilities of the terminal, can be any combination of "bright" (SGR codes 90-97, 100-107), "256" (SGR codes starting with "38;5" or "48;5"), "truecolor" (SGR codes starting with "38;2" or "48;2"), and "all". "all" behaves as it does for the ctl parameter: "all" combined with any other value means all terminal capabilities except that one. fansi will warn if it encounters SGR codes that exceed the terminal capabilities specified (see term_cap_test for details). In versions prior to 1.0, fansi would also skip exceeding SGRs entirely instead of interpreting them. You may add the string "old" to any otherwise valid term.cap spec to restore the pre 1.0 behavior. "old" will not interact with "all" the way other valid values for this parameter do.
ctl	character, which *Control Sequences* should be treated specially. Special treatment is context dependent, and may include detecting them and/or computing their display/character width as zero. For the SGR subset of the ANSI CSI sequences, and OSC hyperlinks, fansi will also parse, interpret, and reapply the sequences as needed. You can modify whether a *Control Sequence* is treated specially with the ctl parameter. • "nl": newlines.

- "c0": all other "C0" control characters (i.e. 0x01-0x1f, 0x7F), except for newlines and the actual ESC (0x1B) character.
- "sgr": ANSI CSI SGR sequences.
- "csi": all non-SGR ANSI CSI sequences.
- "url": OSC hyperlinks
- "osc": all non-OSC-hyperlink OSC sequences.
- "esc": all other escape sequences.
- "all": all of the above, except when used in combination with any of the above, in which case it means "all but".

`normalize` TRUE or FALSE (default) whether SGR sequence should be normalized out such that there is one distinct sequence for each SGR code. normalized strings will occupy more space (e.g. "\033[31;42m" becomes "\033[31m\033[42m"), but will work better with code that assumes each SGR code will be in its own escape as `crayon` does.

`carry` TRUE, FALSE (default), or a scalar string, controls whether to interpret the character vector as a "single document" (TRUE or string) or as independent elements (FALSE). In "single document" mode, active state at the end of an input element is considered active at the beginning of the next vector element, simulating what happens with a document with active state at the end of a line. If FALSE each vector element is interpreted as if there were no active state when it begins. If character, then the active state at the end of the `carry` string is carried into the first element of `x` (see "Replacement Functions" for differences there). The carried state is injected in the interstice between an imaginary zeroeth character and the first character of a vector element. See the "Position Semantics" section of `substr_ctl` and the "State Interactions" section of `?fansi` for details. Except for `strwrap_ctl` where `NA` is treated as the string `"NA"`, `carry` will cause `NA`s in inputs to propagate through the remaining vector elements.

`terminate` TRUE (default) or FALSE whether substrings should have active state closed to avoid it bleeding into other strings they may be prepended onto. This does not stop state from carrying if `carry = TRUE`. See the "State Interactions" section of `?fansi` for details.

`type` character(1L) partial matching `c("chars", "width", "graphemes")`, although types other than "chars" only work correctly with R >= 3.2.2. See `?nchar`.

`round` character(1L) partial matching `c("start", "stop", "both", "neither")`, controls how to resolve ambiguities when a `start` or `stop` value in "width" `type` mode falls within a wide display character. See details.

`tabs.as.spaces` FALSE (default) or TRUE, whether to convert tabs to spaces (and supress tab related warnings). This can only be set to TRUE if `strip.spaces` is FALSE.

`tab.stops` integer(1:n) indicating position of tab stops to use when converting tabs to spaces. If there are more tabs in a line than defined tab stops the last tab stop is re-used. For the purposes of applying tab stops, each input line is considered a line and the character count begins from the beginning of the input line.

`value` a character vector or object that can be coerced to such.

Value

A character vector of the same length and with the same attributes as x (after possible coercion and re-encoding to UTF-8).

Control and Special Sequences

Control Sequences are non-printing characters or sequences of characters. *Special Sequences* are a subset of the *Control Sequences*, and include CSI SGR sequences which can be used to change rendered appearance of text, and OSC hyperlinks. See `fansi` for details.

Position Semantics

When computing substrings, *Normal* (non-control) characters are considered to occupy positions in strings, whereas *Control Sequences* occupy the interstices between them. The string:

```
"hello-\033[31mworld\033[m!"
```

is interpreted as:

```
                  1 1 1
 1 2 3 4 5 6 7 8 9 0 1 2
 h e l l o -|w o r l d|!
            ^         ^
            \033[31m  \033[m
```

`start` and `stop` reference character positions so they never explicitly select for the interstitial *Control Sequences*. The latter are implicitly selected if they appear in interstices after the first character and before the last. Additionally, because *Special Sequences* (CSI SGR and OSC hyperlinks) affect all subsequent characters in a string, any active *Special Sequence*, whether opened just before a character or much before, will be reflected in the state `fansi` prepends to the beginning of each substring.

It is possible to select *Control Sequences* at the end of a string by specifying `stop` values past the end of the string, although for *Special Sequences* this only produces visible results if `terminate` is set to `FALSE`. Similarly, it is possible to select *Control Sequences* preceding the beginning of a string by specifying `start` values less than one, although as noted earlier this is unnecessary for *Special Sequences* as those are output by `fansi` before each substring.

Because exact substrings on anything other than character count cannot be guaranteed (e.g. as a result of multi-byte encodings, or double display-width characters) `substr2_ctl` must make assumptions on how to resolve provided `start`/`stop` values that are infeasible and does so via the `round` parameter.

If we use "start" as the `round` value, then any time the `start` value corresponds to the middle of a multi-byte or a wide character, then that character is included in the substring, while any similar partially included character via the `stop` is left out. The converse is true if we use "stop" as the `round` value. "neither" would cause all partial characters to be dropped irrespective whether they correspond to `start` or `stop`, and "both" could cause all of them to be included. See examples.

A number of *Normal* characters such as combining diacritic marks have reported width of zero. These are typically displayed overlaid on top of the preceding glyph, as in the case of `"e\u301"`

forming "e" with an acute accent. Unlike *Control Sequences*, which also have reported width of zero, `fansi` groups zero-width *Normal* characters with the last preceding non-zero width *Normal* character. This is incorrect for some rare zero-width *Normal* characters such as prepending marks (see "Output Stability" and "Graphemes").

Output Stability

Several factors could affect the exact output produced by `fansi` functions across versions of `fansi`, `R`, and/or across systems. **In general it is best not to rely on exact `fansi` output, e.g. by embedding it in tests**.

Width and grapheme calculations depend on locale, Unicode database version, and grapheme processing logic (which is still in development), among other things. For the most part `fansi` (currently) uses the internals of `base::nchar(type='width')`, but there are exceptions and this may change in the future.

How a particular display format is encoded in *Control Sequences* is not guaranteed to be stable across `fansi` versions. Additionally, which *Special Sequences* are re-encoded vs transcribed untouched may change. In general we will strive to keep the rendered appearance stable.

To maximize the odds of getting stable output set `normalize_state` to `TRUE` and `type` to `"chars"` in functions that allow it, and set `term.cap` to a specific set of capabilities.

Replacement Functions

Semantics for replacement functions have the additional requirement that the result appear as if it is the input modified in place between the positions designated by `start` and `stop`. `terminate` only affects the boundaries between the original substring and the spliced one, `normalize` only affects the same boundaries, and `tabs.as.spaces` only affects `value`, and `x` must be ASCII only or marked "UTF-8".

`terminate = FALSE` only makes sense in replacement mode if only one of `x` or `value` contains *Control Sequences*. `fansi` will not account for any interactions of state in `x` and `value`.

The `carry` parameter causes state to carry within the original string and the replacement values independently, as if they were columns of text cut from different pages and pasted together. String values for `carry` are disallowed in replacement mode as it is ambiguous which of `x` or `value` they would modify (see examples).

When in `type = 'width'` mode, it is only guaranteed that the result will be no wider than the original `x`. Narrower strings may result if a mixture of narrow and wide graphemes cannot be replaced exactly with the same `width` value, possibly because the provided `start` and `stop` values (or the implicit ones generated for `value`) do not align with grapheme boundaries.

Graphemes

`fansi` approximates grapheme widths and counts by using heuristics for grapheme breaks that work for most common graphemes, including emoji combining sequences. The heuristic is known to work incorrectly with invalid combining sequences, prepending marks, and sequence interruptors. `fansi` does not provide a full implementation of grapheme break detection to avoid carrying a copy of the Unicode grapheme breaks table, and also because the hope is that R will add the feature eventually itself.

The `utf8` package provides a conforming grapheme parsing implementation.

Bidirectional Text

`fansi` is unaware of text directionality and operates as if all strings are left to right (LTR). Using `fansi` function with strings that contain mixed direction scripts (i.e. both LTR and RTL) may produce undesirable results.

Note

Non-ASCII strings are converted to and returned in UTF-8 encoding. Width calculations will not work properly in R < 3.2.2.

If `stop` < `start`, the return value is always an empty string.

See Also

`?fansi` for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, `normalize_state` for more details on what the `normalize` parameter does, `state_at_end` to compute active state at the end of strings, `close_state` to compute the sequence required to close active state.

Examples

```
substr_ctl("\033[42mhello\033[m world", 1, 9)
substr_ctl("\033[42mhello\033[m world", 3, 9)

## Positions 2 and 4 are in the middle of the full width W (\uFF37) for
## the `start` and `stop` positions respectively. Use `round`
## to control result:
x <- "\uFF37n\uFF37"
x
substr2_ctl(x, 2, 4, type='width', round='start')
substr2_ctl(x, 2, 4, type='width', round='stop')
substr2_ctl(x, 2, 4, type='width', round='neither')
substr2_ctl(x, 2, 4, type='width', round='both')

## We can specify which escapes are considered special:
substr_ctl("\033[31mhello\tworld", 1, 6, ctl='sgr', warn=FALSE)
substr_ctl("\033[31mhello\tworld", 1, 6, ctl=c('all', 'c0'), warn=FALSE)

## `carry` allows SGR to carry from one element to the next
substr_ctl(c("\033[33mhello", "world"), 1, 3)
substr_ctl(c("\033[33mhello", "world"), 1, 3, carry=TRUE)
substr_ctl(c("\033[33mhello", "world"), 1, 3, carry="\033[44m")

## We can omit the termination
bleed <- substr_ctl(c("\033[41mhello", "world"), 1, 3, terminate=FALSE)
writeLines(bleed)      # Style will bleed out of string
end <- "\033[0m\n"
writeLines(end)        # Stanch bleeding

## Trailing sequences omitted unless `stop` past end.
substr_ctl("ABC\033[42m", 1, 3, terminate=FALSE)
substr_ctl("ABC\033[42m", 1, 4, terminate=FALSE)
```

```
## Replacement functions
x0<- x1 <- x2 <- x3 <- c("\033[42mABC", "\033[34mDEF")
substr_ctl(x1, 2, 2) <- "_"
substr_ctl(x2, 2, 2) <- "\033[m_"
substr_ctl(x3, 2, 2) <- "\033[45m_"
writeLines(c(x0, end, x1, end, x2, end, x3, end))

## With `carry = TRUE` strings look like original
x0<- x1 <- x2 <- x3 <- c("\033[42mABC", "\033[34mDEF")
substr_ctl(x0, 2, 2, carry=TRUE) <- "_"
substr_ctl(x1, 2, 2, carry=TRUE) <- "\033[m_"
substr_ctl(x2, 2, 2, carry=TRUE) <- "\033[45m_"
writeLines(c(x0, end, x1, end, x2, end, x3, end))

## Work-around to specify carry strings in replacement mode
x <- c("ABC", "DEF")
val <- "#"
x2 <- c("\033[42m", x)
val2 <- c("\033[45m", rep_len(val, length(x)))
substr_ctl(x2, 2, 2, carry=TRUE) <- val2
(x <- x[-1])
```

`tabs_as_spaces` *Replace Tabs With Spaces*

Description

Finds horizontal tab characters (0x09) in a string and replaces them with the spaces that produce the same horizontal offset.

Usage

```
tabs_as_spaces(
  x,
  tab.stops = getOption("fansi.tab.stops", 8L),
  warn = getOption("fansi.warn", TRUE),
  ctl = "all"
)
```

Arguments

`x`	character vector or object coercible to character; any tabs therein will be replaced.
`tab.stops`	integer(1:n) indicating position of tab stops to use when converting tabs to spaces. If there are more tabs in a line than defined tab stops the last tab stop is re-used. For the purposes of applying tab stops, each input line is considered a line and the character count begins from the beginning of the input line.

warn
: TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions `fansi` makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see `?fansi`). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use `validUTF8` for that).

ctl
: character, which *Control Sequences* should be treated specially. Special treatment is context dependent, and may include detecting them and/or computing their display/character width as zero. For the SGR subset of the ANSI CSI sequences, and OSC hyperlinks, `fansi` will also parse, interpret, and reapply the sequences as needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter.

  - "nl": newlines.
  - "c0": all other "C0" control characters (i.e. 0x01-0x1f, 0x7F), except for newlines and the actual ESC (0x1B) character.
  - "sgr": ANSI CSI SGR sequences.
  - "csi": all non-SGR ANSI CSI sequences.
  - "url": OSC hyperlinks
  - "osc": all non-OSC-hyperlink OSC sequences.
  - "esc": all other escape sequences.
  - "all": all of the above, except when used in combination with any of the above, in which case it means "all but".

Details

Since we do not know of a reliable cross platform means of detecting tab stops you will need to provide them yourself if you are using anything outside of the standard tab stop every 8 characters that is the default.

Value

character, `x` with tabs replaced by spaces, with elements possibly converted to UTF-8.

Note

Non-ASCII strings are converted to and returned in UTF-8 encoding. The `ctl` parameter only affects which *Control Sequences* are considered zero width. Tabs will always be converted to spaces, irrespective of the `ctl` setting.

See Also

`?fansi` for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, `unhandled_ctl` for detecting bad control sequences.

Examples

```
string <- '1\t12\t123\t1234\t12345678'
tabs_as_spaces(string)
writeLines(
  c(
    '-------|-------|-------|-------|-------|',
    tabs_as_spaces(string)
) )
writeLines(
  c(
    '-|--|--|--|--|--|--|--|--|--|--|',
    tabs_as_spaces(string, tab.stops=c(2, 3))
) )
writeLines(
  c(
    '-|--|-------|-------|-------|',
    tabs_as_spaces(string, tab.stops=c(2, 3, 8))
) )
```

`term_cap_test` *Test Terminal Capabilities*

Description

Outputs ANSI CSI SGR formatted text to screen so that you may visually inspect what color capabilities your terminal supports.

Usage

```
term_cap_test()
```

Details

The three tested terminal capabilities are:

- "bright" for bright colors with SGR codes in 90-97 and 100-107
- "256" for colors defined by "38;5;x" and "48;5;x" where x is in 0-255
- "truecolor" for colors defined by "38;2;x;y;z" and "48;x;y;x" where x, y, and z are in 0-255

Each of the color capabilities your terminal supports should be displayed with a blue background and a red foreground. For reference the corresponding CSI SGR sequences are displayed as well.

You should compare the screen output from this function to `getOption('fansi.term.cap', dflt_term_cap)` to ensure that they are self consistent.

By default `fansi` assumes terminals support bright and 256 color modes, and also tests for truecolor support via the $COLORTERM system variable.

Functions with the `term.cap` parameter like `substr_ctl` will warn if they encounter 256 or true color SGR sequences and `term.cap` indicates they are unsupported as such a terminal may misinterpret those sequences. Bright codes and OSC hyperlinks in terminals that do not support them will likely be silently ignored, so `fansi` functions do not warn about those.

Value

character the test vector, invisibly

See Also

dflt_term_cap, has_ctl.

Examples

```
term_cap_test()
```

`to_html` *Convert Control Sequences to HTML Equivalents*

Description

Interprets CSI SGR sequences and OSC hyperlinks to produce strings with the state reproduced with SPAN elements, inline CSS styles, and A anchors. Optionally for colors, the SPAN elements may be assigned classes instead of inline styles, in which case it is the user's responsibility to provide a style sheet. Input that contains special HTML characters ("<", ">", "&", "'", and "\"") likely should be escaped with html_esc, and `to_html` will warn if it encounters the first two.

Usage

```
to_html(
  x,
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  classes = FALSE,
  carry = getOption("fansi.carry", TRUE)
)
```

Arguments

`x`	a character vector or object that can be coerced to such.
`warn`	TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions `fansi` makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see `?fansi`). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use validUTF8 for that).
`term.cap`	character a vector of the capabilities of the terminal, can be any combination of "bright" (SGR codes 90-97, 100-107), "256" (SGR codes starting with "38;5" or "48;5"), "truecolor" (SGR codes starting with "38;2" or "48;2"), and "all". "all" behaves as it does for the `ctl` parameter: "all" combined with any other value

means all terminal capabilities except that one. fansi will warn if it encounters SGR codes that exceed the terminal capabilities specified (see term_cap_test for details). In versions prior to 1.0, fansi would also skip exceeding SGRs entirely instead of interpreting them. You may add the string "old" to any otherwise valid term.cap spec to restore the pre 1.0 behavior. "old" will not interact with "all" the way other valid values for this parameter do.

classes

FALSE (default), TRUE, or character vector of either 16, 32, or 512 class names. Character strings may only contain ASCII characters corresponding to letters, numbers, the hyphen, or the underscore. It is the user's responsibility to provide values that are legal class names.

- FALSE: All colors rendered as inline CSS styles.
- TRUE: Each of the 256 basic colors is mapped to a class in form "fansi-color-###" (or "fansi-bgcol-###" for background colors) where "###" is a zero padded three digit number in 0:255. Basic colors specified with SGR codes 30-37 (or 40-47) map to 000:007, and bright ones specified with 90-97 (or 100-107) map to 008:015. 8 bit colors specified with SGR codes 38;5;### or 48;5;### map directly based on the value of "###". Implicitly, this maps the 8 bit colors in 0:7 to the basic colors, and those in 8:15 to the bright ones even though these are not exactly the same when using inline styles. "truecolor"s specified with 38;2;#;#;# or 48;2;#;#;# do not map to classes and are rendered as inline styles.
- character(16): The eight basic colors are mapped to the string values in the vector, all others are rendered as inline CSS styles. Basic colors are mapped irrespective of whether they are encoded as the basic colors or as 8-bit colors. Sixteen elements are needed because there must be eight classes for foreground colors, and eight classes for background colors. Classes should be ordered in ascending order of color number, with foreground and background classes alternating starting with foreground (see examples).
- character(32): Like character(16), except the basic and bright colors are mapped.
- character(512): Like character(16), except the basic, bright, and all other 8-bit colors are mapped.

carry

TRUE, FALSE (default), or a scalar string, controls whether to interpret the character vector as a "single document" (TRUE or string) or as independent elements (FALSE). In "single document" mode, active state at the end of an input element is considered active at the beginning of the next vector element, simulating what happens with a document with active state at the end of a line. If FALSE each vector element is interpreted as if there were no active state when it begins. If character, then the active state at the end of the carry string is carried into the first element of x (see "Replacement Functions" for differences there). The carried state is injected in the interstice between an imaginary zeroeth character and the first character of a vector element. See the "Position Semantics" section of substr_ctl and the "State Interactions" section of ?fansi for details. Except for strwrap_ctl where NA is treated as the string "NA", carry will cause NAs in inputs to propagate through the remaining vector elements.

Details

Only "observable" formats are translated. These include colors, background-colors, and basic styles (CSI SGR codes 1-6, 8, 9). Style 7, the "inverse" style, is implemented by explicitly switching foreground and background colors, if there are any. Styles 5-6 (blink) are rendered as "text-decoration" but likely will do nothing in the browser. Style 8 (conceal) sets the color to transparent.

Parameters in OSC sequences are not copied over as they might have different semantics in the OSC sequences than they would in HTML (e.g. the "id" parameter is intended to be non-unique in OSC).

Each element of the input vector is translated into a stand-alone valid HTML string. In particular, any open tags generated by `fansi` are closed at the end of an element and re-opened on the subsequent element with the same style. This allows safe combination of HTML translated strings, for example by `paste`ing them together. The trade-off is that there may be redundant HTML produced. To reduce redundancy you can first collapse the input vector into one string, being mindful that very large strings may exceed maximum string size when converted to HTML.

`fansi`-opened tags are closed and new ones open anytime the "observable" state changes. `to_html` never produces nested tags, even if at times that might produce more compact output. While it would be possible to match a CSI/OSC encoded state with nested tags, it would increase the complexity of the code substantially for little gain.

Value

A character vector of the same length as `x` with all escape sequences removed and any basic ANSI CSI SGR escape sequences applied via SPAN HTML tags.

Note

Non-ASCII strings are converted to and returned in UTF-8 encoding.

`to_html` always terminates as not doing so produces invalid HTML. If you wish for the last active SPAN to bleed into subsequent text you may do so with e.g. `sub("</span>(?:</a>)?$", "", x)` or similar. Additionally, unlike other functions, the default is `carry = TRUE` for compatibility with semantics of prior versions of `fansi`.

See Also

Other HTML functions: `html_esc()`, `in_html()`, `make_styles()`

Examples

```
to_html("hello\033[31;42;1mworld\033[m")
to_html("hello\033[31;42;1mworld\033[m", classes=TRUE)

## Input contains HTML special chars
x <- "<hello \033[42m'there' \033[34m &amp;\033[m \"moon\"!"
writeLines(x)
## Not run:
in_html(
  c(
    to_html(html_esc(x)),  # Good
    to_html(x)             # Bad (warning)!
```

```
) )

## End(Not run)
## Generate some class names for basic colors
classes <- expand.grid(
  "myclass",
  c("fg", "bg"),
  c("black", "red", "green", "yellow", "blue", "magenta", "cyan", "white")
)
classes  # order is important!
classes <- do.call(paste, c(classes, sep="-"))
## We only provide 16 classes, so Only basic colors are
## mapped to classes; others styled inline.
to_html(
  "\033[94mhello\033[m \033[31;42;1mworld\033[m",
  classes=classes
)
## Create a whole web page with a style sheet for 256 colors and
## the colors shown in a table.
class.256 <- do.call(paste, c(expand.grid(c("fg", "bg"), 0:255), sep="-"))
sgr.256 <- sgr_256()     # A demo of all 256 colors
writeLines(sgr.256[1:8]) # SGR formatting

## Convert to HTML using classes instead of inline styles:
html.256 <- to_html(sgr.256, classes=class.256)
writeLines(html.256[1])  # No inline colors

## Generate different style sheets.  See `?make_styles` for details.
default <- make_styles(class.256)
mix <- matrix(c(.6,.2,.2, .2,.6,.2, .2,.2,.6), 3)
desaturated <- make_styles(class.256, mix)
writeLines(default[1:4])
writeLines(desaturated[1:4])

## Embed in HTML page and diplay; only CSS changing
## Not run:
in_html(html.256)                  # no CSS
in_html(html.256, css=default)     # default CSS
in_html(html.256, css=desaturated) # desaturated CSS

## End(Not run)
```

`trimws_ctl` *Control Sequence Aware Version of trimws*

Description

Removes any whitespace before the first and/or after the last non-*Control Sequence* character. Unlike with the `base::trimws`, only the default `whitespace` specification is supported.

Usage

```
trimws_ctl(
  x,
  which = c("both", "left", "right"),
  whitespace = "[ \t\r\n]",
  warn = getOption("fansi.warn", TRUE),
  term.cap = getOption("fansi.term.cap", dflt_term_cap()),
  ctl = "all",
  normalize = getOption("fansi.normalize", FALSE)
)
```

Arguments

x	a character vector
which	a character string specifying whether to remove both leading and trailing whitespace (default), or only leading (`"left"`) or trailing (`"right"`). Can be abbreviated.
whitespace	must be set to the default value, in the future it may become possible to change this parameter.
warn	TRUE (default) or FALSE, whether to warn when potentially problematic *Control Sequences* are encountered. These could cause the assumptions `fansi` makes about how strings are rendered on your display to be incorrect, for example by moving the cursor (see `?fansi`). At most one warning will be issued per element in each input vector. Will also warn about some badly encoded UTF-8 strings, but a lack of UTF-8 warnings is not a guarantee of correct encoding (use `validUTF8` for that).
term.cap	character a vector of the capabilities of the terminal, can be any combination of "bright" (SGR codes 90-97, 100-107), "256" (SGR codes starting with "38;5" or "48;5"), "truecolor" (SGR codes starting with "38;2" or "48;2"), and "all". "all" behaves as it does for the `ctl` parameter: "all" combined with any other value means all terminal capabilities except that one. `fansi` will warn if it encounters SGR codes that exceed the terminal capabilities specified (see `term_cap_test` for details). In versions prior to 1.0, `fansi` would also skip exceeding SGRs entirely instead of interpreting them. You may add the string "old" to any otherwise valid `term.cap` spec to restore the pre 1.0 behavior. "old" will not interact with "all" the way other valid values for this parameter do.
ctl	character, which *Control Sequences* should be treated specially. Special treatment is context dependent, and may include detecting them and/or computing their display/character width as zero. For the SGR subset of the ANSI CSI sequences, and OSC hyperlinks, `fansi` will also parse, interpret, and reapply the sequences as needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter. • "nl": newlines. • "c0": all other "C0" control characters (i.e. 0x01-0x1f, 0x7F), except for newlines and the actual ESC (0x1B) character. • "sgr": ANSI CSI SGR sequences.

- "csi": all non-SGR ANSI CSI sequences.
- "url": OSC hyperlinks
- "osc": all non-OSC-hyperlink OSC sequences.
- "esc": all other escape sequences.
- "all": all of the above, except when used in combination with any of the above, in which case it means "all but".

`normalize` TRUE or FALSE (default) whether SGR sequence should be normalized out such that there is one distinct sequence for each SGR code. normalized strings will occupy more space (e.g. "\033[31;42m" becomes "\033[31m\033[42m"), but will work better with code that assumes each SGR code will be in its own escape as `crayon` does.

Value

The input with white space removed as described.

Control and Special Sequences

Control Sequences are non-printing characters or sequences of characters. *Special Sequences* are a subset of the *Control Sequences*, and include CSI SGR sequences which can be used to change rendered appearance of text, and OSC hyperlinks. See `fansi` for details.

Output Stability

Several factors could affect the exact output produced by `fansi` functions across versions of `fansi`, `R`, and/or across systems. **In general it is best not to rely on exact `fansi` output, e.g. by embedding it in tests**.

Width and grapheme calculations depend on locale, Unicode database version, and grapheme processing logic (which is still in development), among other things. For the most part `fansi` (currently) uses the internals of `base::nchar(type='width')`, but there are exceptions and this may change in the future.

How a particular display format is encoded in *Control Sequences* is not guaranteed to be stable across `fansi` versions. Additionally, which *Special Sequences* are re-encoded vs transcribed untouched may change. In general we will strive to keep the rendered appearance stable.

To maximize the odds of getting stable output set `normalize_state` to `TRUE` and `type` to `"chars"` in functions that allow it, and set `term.cap` to a specific set of capabilities.

Examples

```
trimws_ctl(" \033[31m\thello world\t\033[39m  ")
```

unhandled_ctl *Identify Unhandled Control Sequences*

Description

Will return position and types of unhandled *Control Sequences* in a character vector. Unhandled sequences may cause fansi to interpret strings in a way different to your display. See fansi for details. Functions that interpret *Special Sequences* (CSI SGR or OSC hyperlinks) might omit bad *Special Sequences* or some of their components in output substrings, particularly if they are leading or trailing. Some functions are more tolerant of bad inputs than others. For example nchar_ctl will not report unsupported colors because it only cares about counts or widths. unhandled_ctl will report all potentially problematic sequences.

Usage

```
unhandled_ctl(x, term.cap = getOption("fansi.term.cap", dflt_term_cap()))
```

Arguments

x	character vector
term.cap	character a vector of the capabilities of the terminal, can be any combination of "bright" (SGR codes 90-97, 100-107), "256" (SGR codes starting with "38;5" or "48;5"), "truecolor" (SGR codes starting with "38;2" or "48;2"), and "all". "all" behaves as it does for the ctl parameter: "all" combined with any other value means all terminal capabilities except that one. fansi will warn if it encounters SGR codes that exceed the terminal capabilities specified (see term_cap_test for details). In versions prior to 1.0, fansi would also skip exceeding SGRs entirely instead of interpreting them. You may add the string "old" to any otherwise valid term.cap spec to restore the pre 1.0 behavior. "old" will not interact with "all" the way other valid values for this parameter do.

Details

To work around tabs present in input, you can use tabs_as_spaces or the tabs.as.spaces parameter on functions that have it, or the strip_ctl function to remove the troublesome sequences. Alternatively, you can use warn=FALSE to suppress the warnings.

This is a debugging function that is not optimized for speed and the precise output of which might change with fansi versions.

The return value is a data frame with five columns:

- index: integer the index in x with the unhandled sequence
- start: integer the start position of the sequence (in characters)
- stop: integer the end of the sequence (in characters), but note that if there are multiple ESC sequences abutting each other they will all be treated as one, even if some of those sequences are valid.
- error: the reason why the sequence was not handled:

  - unknown-substring: SGR substring with a value that does not correspond to a known SGR code or OSC hyperlink with unsupported parameters.
  - invalid-substr: SGR contains uncommon characters in ":<=>", intermediate bytes, other invalid characters, or there is an invalid subsequence (e.g. "ESC[38;2m" which should specify an RGB triplet but does not). OSCs contain invalid bytes, or OSC hyperlinks contain otherwise valid OSC bytes in 0x08-0x0d.
  - exceed-term-cap: contains color codes not supported by the terminal (see term_cap_test). Bright colors with color codes in the 90-97 and 100-107 range in terminals that do not support them are not considered errors, whereas 256 or truecolor codes in terminals that do not support them are. This is because the latter are often misinterpreted by terminals that do not support them, whereas the former are typically silently ignored.
  - CSI/OSC: a non-SGR CSI sequence, or non-hyperlink OSC sequence.
  - CSI/OSC-bad-substr: a CSI or OSC sequence containing invalid characters.
  - malformed-CSI/OSC: a malformed CSI or OSC sequence, typically one that never encounters its closing sequence before the end of a string.
  - non-CSI/OSC: a non-CSI or non-OSC escape sequence, i.e. one where the ESC is followed by something other than "[" or "]". Since we assume all non-CSI sequences are only 2 characters long include the ESC, this type of sequence is the most likely to cause problems as some are not actually two characters long.
  - malformed-ESC: a malformed two byte ESC sequence (i.e. one not ending in 0x40-0x7e).
  - C0: a "C0" control character (e.g. tab, bell, etc.).
  - malformed-UTF8: illegal UTF8 encoding.
  - non-ASCII: non-ASCII bytes in escape sequences.
- translated: whether the string was translated to UTF-8, might be helpful in odd cases were character offsets change depending on encoding. You should only worry about this if you cannot tie out the `start`/`stop` values to the escape sequence shown.
- esc: character the unhandled escape sequence

Value

Data frame with as many rows as there are unhandled escape sequences and columns containing useful information for debugging the problem. See details.

Note

Non-ASCII strings are converted to UTF-8 encoding.

See Also

`?fansi` for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, `unhandled_ctl` for detecting bad control sequences.

Examples

```
string <- c(
  "\033[41mhello world\033[m", "foo\033[22>m", "\033[999mbar",
  "baz \033[31#3m", "a\033[31k", "hello\033m world"
)
```

```
unhandled_ctl(string)
```

Index