Package 'fansi'

May 25, 2021

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 0.5.0

Depends R (>= 3.1.0)

License GPL (>= 2)

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

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

VignetteBuilder knitr

Suggests unitizer, knitr, rmarkdown

Imports grDevices, utils

RoxygenNote 7.1.1

Encoding UTF-8

Collate 'constants.R' 'fansi-package.R' 'has.R' 'internal.R' 'load.R' 'misc.R' 'nchar.R' 'strip.R' 'strwrap.R' 'strtrim.R' 'strsplit.R' 'substr2.R' 'tohtml.R' 'unhandled.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 2021-05-25 04:40:10 UTC

R topics documented:

`fansi` *Details About Manipulation of Strings Containing Control Sequences*

Description

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

Control Characters and Sequences

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

We will refer to ANSI control characters and sequences as "*Control Sequences*" hereafter.

There are three 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 ANSI CSI sequences.
- Sequences starting in "ESC" and followed by something other than "[".

Control Sequences starting with ESC are assumed to be two characters long (including the ESC) unless they are of the CSI variety, in which case their length is computed as per the ECMA-48 specification. There are non-CSI escape sequences that may be longer than two characters, but `fansi` will (incorrectly) treat them as if they were two characters long.

In theory it is possible to encode *Control 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, we do not support it.

The special treatment of *Control Sequences* is to compute their display/character width as zero. For the SGR subset of the ANSI CSI sequences, `fansi` will also parse, interpret, and reapply the text styles they encode as needed. Whether a particular type of *Control Sequence* is treated specially can be specified via the `ctl` parameter to the `fansi` functions that have it.

ANSI CSI SGR Control Sequences

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

ANSI 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 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. To mitigate this particular problem you can tell `fansi` what your terminal capabilities are via the `term.cap` parameter or the "fansi.term.cap" global option, although `fansi` does try to detect them by default.

`fansi` will will warn if it encounters *Control Sequences* that it cannot interpret or that might conflict with terminal capabilities. You can turn off warnings via the `warn` parameter or via the "fansi.warn" global option.

`fansi` can work around "C0" tab control 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.

We chose to interpret ANSI CSI SGR sequences because this reduces how much string transcription we need to do during string manipulation. If we do not interpret the sequences then we need to record all of them from the beginning of the string and prepend all the accumulated tags up to beginning of a substring to the substring. In many case the bulk of those accumulated tags will be irrelevant as their effects will have been superseded by subsequent tags.

`fansi` assumes that ANSI CSI SGR sequences should be interpreted 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).

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 adopts newer versions of Unicode. Do not expect the `fansi` width calculations to always work correctly with strings containing emoji.

Internally, `fansi` computes the width of every UTF-8 character sequence outside of the ASCII range using the native `R_nchar` function. This will cause such characters to be processed slower than ASCII characters. Additionally, `fansi` character width computations can differ from R width computations despite the use of `R_nchar`. `fansi` always computes width for each character individually, which assumes that the sum of the widths of each character is equal to the width of a sequence. However, it is theoretically possible for a character sequence that forms a single grapheme to break that assumption. In informal testing we have found this to be rare because in the most common multi-character graphemes the trailing characters are computed as zero width.

As of R 3.4.0 `substr` appears to use UTF-8 character byte sizes as indicated by the leading byte, irrespective of whether the subsequent bytes lead to a valid sequence. Additionally, UTF-8 byte sequences as long as 5 or 6 bytes may be allowed, which is likely a holdover from older Unicode versions. `fansi` mimics this behavior. It is likely `substr` will start failing with invalid UTF-8 byte sequences with R 3.6.0 (as per SVN r74488). In general, you should assume that `fansi` may not replicate base R exactly when there are illegal UTF-8 sequences present.

Our long term objective is to implement proper UTF-8 character width computations, but for simplicity and also because R and our terminal do not do it properly either we are deferring the issue for now.

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`.

Overflow

The native code in this package assumes that all strings are NULL terminated and no longer than (32 bit) INT_MAX (excluding the NULL). This should be a safe assumption since the code is designed

to work with STRSXPs and CHRSXPs. Behavior is undefined and probably bad if you somehow manage to provide to `fansi` strings that do not adhere to these assumptions.

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.

`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` *Checks for Presence of Control Sequences*

Description

`has_ctl` checks for any *Control Sequence*, whereas `has_sgr` checks only for ANSI CSI SGR sequences. You can check for different types of sequences with the `ctl` parameter.

Usage

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

has_sgr(x, warn = getOption("fansi.warn"))
```

Arguments

x	a character vector or object that can be coerced to character.
ctl	character, which *Control Sequences* should be treated specially. See the "_ctl vs. _sgr" section for details. • "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. • "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).
which	character, deprecated in favor of `ctl`.

Value

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

_ctl vs. _sgr

The *_ctl versions of the functions treat all *Control Sequences* specially by default. 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, `fansi` will also parse, interpret, and reapply the text styles they encode if needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter. You can exclude a type of *Control Sequence* from special treatment by combining "all" with that type of sequence (e.g. `ctl=c("all","nl")` for special treatment of all *Control Sequences* **but** newlines). The *_sgr versions only treat ANSI CSI SGR sequences specially, and are equivalent to the *_ctl versions with the `ctl` parameter set to "sgr".

See Also

fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results.

Examples

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

`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. sgr_to_html.

Usage

```
html_esc(x)
```

Arguments

x	character vector

Value

x, but with "<", ">", "&", "'", and "\"" characters replaced by their HTML entity codes, and Encoding set to UTF-8.

See Also

Other HTML functions: in_html(), make_styles(), sgr_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(), sgr_to_html()

Examples

```
txt <- "\033[31;42mHello \033[7mWorld\033[m"
writeLines(txt)
html <- sgr_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, or a scalar integer with value 8, 16, or 256. The character vectors are described in sgr_to_html. The scalar integers will cause this function to generate classes for the basic colors (8), basic + bright (16), or all 256 8-bit colors (256), with class names in "fansi-color-###" (or "fansi-bgcol-###" for background colors), which is what sgr_to_html generates when-user defined classes are not provided. TRUE is also a valid input and is equivalent to 256.
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(), sgr_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 <- sgr_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 *ANSI 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"),
  strip
)

nchar_sgr(
  x,
  type = "chars",
  allowNA = FALSE,
  keepNA = NA,
  warn = getOption("fansi.warn")
)

nzchar_ctl(x, keepNA = NA, ctl = "all", warn = getOption("fansi.warn"))

nzchar_sgr(x, keepNA = NA, warn = getOption("fansi.warn"))
```

Arguments

x	a character vector or object that can be coerced to character.
type	character(1L) partial matching `c("chars","width")`, although `type="width"` only works correctly with R >= 3.2.2. With "width", whether C0 and C1 are treated as zero width may depend on R version and locale in addition what the `ctl` parameter is set to. For example, for R4.1 in UTF-8 locales C0 and C1 will be zero width even if the value of `ctl` is such that they wouldn't be so in other circumstances.
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. See the "_ctl vs. _sgr" section for details. • "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. • "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).
strip	character, deprecated in favor of `ctl`.

Details

`nchar_ctl` is just a wrapper around `nchar(strip_ctl(...))`. `nzchar_ctl` is implemented in native code and is much faster than the otherwise equivalent `nzchar(strip_ctl(...))`.

These functions will warn if either malformed or non-CSI escape sequences are encountered, as these may be incorrectly interpreted.

_ctl vs. _sgr

The *_ctl versions of the functions treat all *Control Sequences* specially by default. 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, `fansi` will also parse, interpret, and reapply the text styles they encode if needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter. You can exclude a type of *Control Sequence* from special treatment by combining "all" with that type of sequence (e.g. `ctl=c("all","nl")` for special treatment of all *Control Sequences* **but** newlines). The *_sgr versions only treat ANSI CSI SGR sequences specially, and are equivalent to the *_ctl versions with the `ctl` parameter set to "sgr".

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, strip_ctl for removing *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")
```

---

set_knit_hooks *Set an Output Hook to Display ANSI CSI SGR 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 ANSI CSI SGR 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 ANSI CSI SGR are shown in their HTML equivalent form.

Usage

```
set_knit_hooks(
  hooks,
  which = "output",
  proc.fun = function(x, class) html_code_block(sgr_to_html(html_esc(x)), class =
```

```
    class),
  class = sprintf("fansi fansi-%s", which),
  style = getOption("fansi.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 ANSI 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 sgr_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 ANSI CSI SGR sequences.
.test	TRUE or FALSE, for internal testing use only.

Details

The replacement hook function tests for the presence of ANSI CSI SGR sequences in chunk output with has_sgr, and if it is detected then processes it with the user provided proc.fun. Chunks that do not contain ANSI CSI SGR are passed off to the previously set hook function. The default proc.fun will run the output through html_esc, sgr_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_sgr, sgr_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'),  # 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 ANSI 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())
```

sgr_to_html *Convert ANSI CSI SGR Escape Sequence to HTML Equivalents*

Description

Interprets CSI SGR sequences and produces a string with equivalent formats applied with SPAN elements and inline CSS styles. 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 "\""), particularly the first two, should be escaped with `html_esc`.

Usage

```
sgr_to_html(
  x,
  warn = getOption("fansi.warn"),
  term.cap = getOption("fansi.term.cap"),
  classes = FALSE
)
```

Arguments

x	a character vector or object that can be coerced to character.
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).
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"), and "truecolor" (SGR codes starting with "38;2" or "48;2"). Changing this parameter changes how fansi interprets escape sequences, so you should ensure that it matches your terminal capabilities. See term_cap_test for details.
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.

Details

Only "observable" styles 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.

Each element of the input vector is translated into a stand-alone valid HTML string. In particular, any open SPAN tags 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 pasteing 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.

Active SPAN tags are closed and new ones open anytime the "observable" state changes. `sgr_to_html` never produces nested SPAN tags, even if at times that might produce more compact output. This is because ANSI CSI SGR is a state based formatting system and is not constrained by the semantics of a nested one like HTML, so dealing with the complexity of nesting when it cannot reproduce all inputs anyway does not seem worthwhile.

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.

See Also

fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, set_knit_hooks for how to use ANSI CSI styled text with knitr and HTML output, sgr_256 to generate a demo string with all 256 8 bit colors.

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

Examples

```
sgr_to_html("hello\033[31;42;1mworld\033[m")
sgr_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(
    sgr_to_html(html_esc(x)),  # Good
    sgr_to_html(x)             # Bad!
) )

## 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.
sgr_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 <- sgr_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)
```

`strip_ctl` *Strip ANSI Control Sequences*

Description

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

Usage

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

strip_sgr(x, warn = getOption("fansi.warn"))
```

Arguments

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

`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).

`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 option then everything **but** that option will be stripped.

Value

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

_ctl vs. _sgr

The *_ctl versions of the functions treat all *Control Sequences* specially by default. 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, `fansi` will also parse, interpret, and reapply the text styles they encode if needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter. You can exclude a type of *Control Sequence* from special treatment by combining "all" with that type of sequence (e.g. `ctl=c("all","nl")` for special treatment of all *Control Sequences* **but** newlines). The *_sgr versions only treat ANSI CSI SGR sequences specially, and are equivalent to the *_ctl versions with the `ctl` parameter set to "sgr".

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.

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"))

## convenience function, same as `strip_ctl(ctl='sgr')`
strip_sgr(string)
```

`strsplit_ctl` *ANSI Control Sequence Aware Version of strsplit*

Description

A drop-in replacement for base::strsplit. It will be noticeably slower, but should otherwise behave the same way except for *Control Sequence* awareness.

Usage

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

strsplit_sgr(
  x,
  split,
  fixed = FALSE,
  perl = FALSE,
  useBytes = FALSE,
  warn = getOption("fansi.warn"),
  term.cap = getOption("fansi.term.cap")
)
```

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).
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"), and "truecolor" (SGR codes starting with "38;2" or "48;2"). Changing this parameter changes how fansi interprets escape sequences, so you should ensure that it matches your terminal capabilities. See term_cap_test for details.
ctl	character, which *Control Sequences* should be treated specially. See the "_ctl vs. _sgr" section for details. • "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. • "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

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

list, see base::strsplit.

_ctl vs. _sgr

The *_ctl versions of the functions treat all *Control Sequences* specially by default. 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, `fansi` will also parse, interpret, and reapply the text styles they encode if needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter. You can exclude a type of *Control Sequence* from special treatment by combining "all" with that type of sequence (e.g. `ctl=c("all","nl")` for special treatment of all *Control Sequences* **but** newlines). The *_sgr versions only treat ANSI CSI SGR sequences specially, and are equivalent to the *_ctl versions with the `ctl` parameter set to "sgr".

Note

Non-ASCII strings are converted to and returned in UTF-8 encoding. The split positions are computed after both `x` and `split` are converted to UTF-8.

See Also

fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results, base::strsplit for details on the splitting.

Examples

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

## Next two examples allow splitting by newlines, which
## normally doesn't work as newlines are _Control Sequences_
strsplit_sgr("\033[31mhello\033[42m\nworld!", "\n")
strsplit_ctl("\033[31mhello\033[42m\nworld!", "\n", ctl=c("all", "nl"))
```

---

strtrim_ctl *ANSI Control Sequence Aware Version of strtrim*

Description

One difference with base::strtrim is that all C0 control characters such as newlines, carriage returns, etc., are treated as zero width.

Usage

```
strtrim_ctl(x, width, warn = getOption("fansi.warn"), ctl = "all")

strtrim2_ctl(
  x,
  width,
  warn = getOption("fansi.warn"),
  tabs.as.spaces = getOption("fansi.tabs.as.spaces"),
  tab.stops = getOption("fansi.tab.stops"),
```

```
  ctl = "all"
)

strtrim_sgr(x, width, warn = getOption("fansi.warn"))

strtrim2_sgr(
  x,
  width,
  warn = getOption("fansi.warn"),
  tabs.as.spaces = getOption("fansi.tabs.as.spaces"),
  tab.stops = getOption("fansi.tab.stops")
)
```

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).
ctl	character, which *Control Sequences* should be treated specially. See the "_ctl vs. _sgr" section for details. • "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. • "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".
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.

_ctl vs. _sgr

The *_ctl versions of the functions treat all *Control Sequences* specially by default. 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, `fansi` will also parse, interpret, and reapply the text styles they encode if needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter. You can exclude a type of *Control Sequence* from special treatment by combining "all" with that type of sequence (e.g. `ctl=c("all","nl")` for special treatment of all *Control Sequences* **but** newlines). The *_sgr versions only treat ANSI CSI SGR sequences specially, and are equivalent to the *_ctl versions with the `ctl` parameter set to "sgr".

Note

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

See Also

fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results. strwrap_ctl is used internally by this function.

Examples

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

`strwrap_ctl` *ANSI Control Sequence Aware Version of strwrap*

Description

Wraps strings to a specified width accounting for zero display width *Control Sequences*. `strwrap_ctl` is intended to emulate `strwrap` exactly except with respect to the *Control Sequences*, while `strwrap2_ctl` adds features and changes the processing of whitespace.

Usage

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

```
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"),
  tab.stops = getOption("fansi.tab.stops"),
  warn = getOption("fansi.warn"),
  term.cap = getOption("fansi.term.cap"),
  ctl = "all"
)

strwrap_sgr(
  x,
  width = 0.9 * getOption("width"),
  indent = 0,
  exdent = 0,
  prefix = "",
  simplify = TRUE,
  initial = prefix,
  warn = getOption("fansi.warn"),
  term.cap = getOption("fansi.term.cap")
)

strwrap2_sgr(
  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"),
  tab.stops = getOption("fansi.tab.stops"),
  warn = getOption("fansi.warn"),
  term.cap = getOption("fansi.term.cap")
)
```

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	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.)
initial	a character string to be used as prefix for each line except the first, for which initial is used.
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).
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"), and "truecolor" (SGR codes starting with "38;2" or "48;2"). Changing this parameter changes how fansi interprets escape sequences, so you should ensure that it matches your terminal capabilities. See term_cap_test for details.
ctl	character, which *Control Sequences* should be treated specially. See the "_ctl vs. _sgr" section for details. • "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. • "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".
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 in boundaries between 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, malformed UTF-8 sequences are not converted to a text representation of bytes.

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.

_ctl vs. _sgr

The *_ctl versions of the functions treat all *Control Sequences* specially by default. 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, `fansi` will also parse, interpret, and reapply the text styles they encode if needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter. You can exclude a type of *Control Sequence* from special treatment by combining "all" with that type of sequence (e.g. `ctl=c("all","nl")` for special treatment of all *Control Sequences* **but** newlines). The *_sgr versions only treat ANSI CSI SGR sequences specially, and are equivalent to the *_ctl versions with the `ctl` parameter set to "sgr".

Note

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

See Also

fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results.

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` *ANSI Control Sequence Aware Version of substr*

Description

`substr_ctl` is a drop-in replacement for `substr`. Performance is slightly slower than `substr`. ANSI CSI SGR sequences will be included in the substrings to reflect the format of the substring when it was embedded in the source string. Additionally, other *Control Sequences* specified in `ctl` are treated as zero-width.

Usage

```
substr_ctl(
  x,
  start,
  stop,
  warn = getOption("fansi.warn"),
  term.cap = getOption("fansi.term.cap"),
```

```
  ctl = "all"
)

substr2_ctl(
  x,
  start,
  stop,
  type = "chars",
  round = "start",
  tabs.as.spaces = getOption("fansi.tabs.as.spaces"),
  tab.stops = getOption("fansi.tab.stops"),
  warn = getOption("fansi.warn"),
  term.cap = getOption("fansi.term.cap"),
  ctl = "all"
)

substr_sgr(
  x,
  start,
  stop,
  warn = getOption("fansi.warn"),
  term.cap = getOption("fansi.term.cap")
)

substr2_sgr(
  x,
  start,
  stop,
  type = "chars",
  round = "start",
  tabs.as.spaces = getOption("fansi.tabs.as.spaces"),
  tab.stops = getOption("fansi.tab.stops"),
  warn = getOption("fansi.warn"),
  term.cap = getOption("fansi.term.cap")
)
```

Arguments

x	a character vector or object that can be coerced to character.
start	integer. The first element to be replaced.
stop	integer. The last element to be 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).
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"), and "truecolor" (SGR codes starting with "38;2" or "48;2"). Changing

this parameter changes how `fansi` interprets escape sequences, so you should ensure that it matches your terminal capabilities. See term_cap_test for details.

`ctl` character, which *Control Sequences* should be treated specially. See the "_ctl vs. _sgr" section for details.

- "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.
- "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".

`type` character(1L) partial matching `c("chars","width")`, although `type="width"` only works correctly with R >= 3.2.2. With "width", whether C0 and C1 are treated as zero width may depend on R version and locale in addition what the `ctl` parameter is set to. For example, for R4.1 in UTF-8 locales C0 and C1 will be zero width even if the value of `ctl` is such that they wouldn't be so in other circumstances.

`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 multi-byte character or a wide display character. See 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

`substr2_ctl` and `substr2_sgr` add the ability to retrieve substrings based on display width, and byte width in addition to the normal character width. `substr2_ctl` also provides the option to convert tabs to spaces with tabs_as_spaces prior to taking substrings.

Because exact substrings on anything other than character width 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.

These functions map string lengths accounting for ANSI CSI SGR sequence semantics to the naive length calculations, and then use the mapping in conjunction with `base::substr()` to extract the string. This concept is borrowed directly from Gábor Csárdi's `crayon` package, although the implementation of the calculation is different.

_ctl vs. _sgr

The *_ctl versions of the functions treat all *Control Sequences* specially by default. 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, `fansi` will also parse, interpret, and reapply the text styles they encode if needed. You can modify whether a *Control Sequence* is treated specially with the `ctl` parameter. You can exclude a type of *Control Sequence* from special treatment by combining "all" with that type of sequence (e.g. `ctl=c("all","nl")` for special treatment of all *Control Sequences* **but** newlines). The *_sgr versions only treat ANSI CSI SGR sequences specially, and are equivalent to the *_ctl versions with the `ctl` parameter set to "sgr".

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.

Examples

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

## Width 2 and 3 are in the middle of an ideogram as
## start and stop positions respectively, so we control
## what we get with `round`

cn.string <- paste0("\033[42m", "\u4E00\u4E01\u4E03", "\033[m")

substr2_ctl(cn.string, 2, 3, type='width')
substr2_ctl(cn.string, 2, 3, type='width', round='both')
substr2_ctl(cn.string, 2, 3, type='width', round='start')
substr2_ctl(cn.string, 2, 3, type='width', round='stop')

## the _sgr variety only treat as special CSI SGR,
## compare the following:

substr_sgr("\033[31mhello\tworld", 1, 6)
substr_ctl("\033[31mhello\tworld", 1, 6)
substr_ctl("\033[31mhello\tworld", 1, 6, ctl=c('all', 'c0'))
```

`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"),
  warn = getOption("fansi.warn"),
  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).
ctl	character, which *Control Sequences* should be treated specially. See the "_ctl vs. _sgr" section for details. • "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. • "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.

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')` 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 in terminals that do not support them are more likely to be silently ignored, so `fansi` functions do not warn about those.

Value

character the test vector, invisibly

See Also

fansi for details on how *Control Sequences* are interpreted, particularly if you are getting unexpected results.

Examples

```
term_cap_test()
```

`unhandled_ctl` *Identify Unhandled ANSI 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.

Usage

```
unhandled_ctl(x, term.cap = getOption("fansi.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"), and "truecolor" (SGR codes starting with "38;2" or "48;2"). Changing this parameter changes how `fansi` interprets escape sequences, so you should ensure that it matches your terminal capabilities. See term_cap_test for details.

Details

This is a debugging function that is not optimized for speed.

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:
  - 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.
  - special: SGR substring contains uncommon characters in ":<=>".
  - unknown: SGR substring with a value that does not correspond to a known SGR code.
  - non-SGR: a non-SGR CSI sequence.
  - non-CSI: a non-CSI escape sequence, i.e. one where the ESC is followed by something other than "[". 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 many are not actually two characters long.
  - malformed-CSI: a malformed CSI sequence.
  - malformed-ESC: a malformed ESC sequence (i.e. one not ending in 0x40-0x7e).
  - C0: a "C0" control character (e.g. tab, bell, etc.).
- 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.

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