Package 'fansi'

March 31, 2018

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 0.2.2

Depends R (>= 3.2.0)

License GPL (>= 2)

LazyData true

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

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

Suggests unitizer

RoxygenNote 6.0.1

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], R Core Team [cph] (UTF8 byte length calcs from src/util.c)

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

Repository CRAN

Date/Publication 2018-03-31 20:10:21 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` treats 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 "[".

All of these are considered zero display-width for purposes of string width calculations.

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-48specification. 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.

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, as of this writing R (and the OSX terminal) consider emojis to be one wide characters, when in reality they are two wide. Do not expect the `fansi` width calculations to to 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.

Miscellaneous

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.

`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 `which` parameter.

Usage

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

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

Arguments

`x`	a character vector or object that can be coerced to character.
`which`	character, what *Control Sequences* to check for; see `strip` parameter for strip_ctl for 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).

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.

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,
  strip = "all", warn = getOption("fansi.warn"))

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

Arguments

x	a character vector or object that can be coerced to character.
type	character string, one of "chars", or "width". For byte counts use base::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 where ever 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". Used to be (implicitly) hard coded to FALSE in R versions $\leq$ 3.2.0.
strip	character, any combination of the following values (see details): • "nl": strip newlines. • "c0": strip all other "C0" control characters (i.e. x01-x1f), 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).

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(...)). You cannot change which *Control Sequences* count in nzchar_ctl, but you can always resort to nzchar(strip_ctl(..., strip='...' if that is important.

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

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 `strip` argument does
## not include newlines.
nchar_ctl("\t\n\r", strip="c0")
nchar_ctl("\t\n\r", strip=c("c0", "nl"))

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

---

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

Description

Only the colors, background-colors, and basic styles (CSI SGR codes 1-9) are translated. Others are dropped silently.

Usage

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

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 "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.

Value

a character vector with all escape sequences removed and any basic ANSI CSI SGR escape sequences applied via SPAN html objects with inline css styles.

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

```
sgr_to_html("hello\033[31;42;1mworld\033[m")
```

---

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 strip parameter. strip_sgr only strips ANSI CSI SGR sequences.

Usage

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

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

Arguments

x	a character vector or object that can be coerced to character.
strip	character, any combination of the following values (see details): • "nl": strip newlines. • "c0": strip all other "C0" control characters (i.e. x01-x1f), 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).

Details

The `strip` 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

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(strip='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 CSI SGR sequence awareness.

Usage

```
strsplit_ctl(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 "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.

Value

list, see base::strsplit.

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

---

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"))

strtrim2_ctl(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).
`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`.

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. 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"))

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"))
```

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 "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.
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.
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.

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

```
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`.

Usage

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

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"))
```

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 "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.
`type`	character(1L) partial matching `c("chars", "width")`.
`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` adds 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. because 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.

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')
```

---

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"))
```

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).

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.

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. The three tested terminal capabilities are:

Usage

```
term_cap_test()
```

Details

- "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.

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)
```

Arguments

`x`	character vector

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).
  - 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