Package 'cli'

March 19, 2019

Title Helpers for Developing Command Line Interfaces

Version 1.1.0

Description A suite of tools designed to build attractive command line interfaces ('CLIs'). Includes tools for drawing rules, boxes, trees, and 'Unicode' symbols with 'ASCII' alternatives.

License MIT + file LICENSE

LazyData true

URL https://github.com/r-lib/cli#readme

BugReports https://github.com/r-lib/cli/issues

RoxygenNote 6.1.1

Depends R (>= 2.10)

Imports assertthat, crayon (>= 1.3.4), methods, utils

Suggests covr, fansi, mockery, testthat, webshot, withr

Encoding UTF-8

NeedsCompilation no

Author Gábor Csárdi [aut, cre],
Hadley Wickham [ctb],
Kirill Müller [ctb]

Maintainer Gábor Csárdi <csardi.gabor@gmail.com>

Repository CRAN

Date/Publication 2019-03-19 10:43:26 UTC

R topics documented:

`ansi-styles` *ANSI colored text*

Description

cli has a number of functions to color and style text at the command line. These all use the crayon package under the hood, but provide a slightly simpler interface.

Usage

```
bg_black(...)

bg_blue(...)

bg_cyan(...)

bg_green(...)

bg_magenta(...)

bg_red(...)

bg_white(...)

bg_yellow(...)

col_black(...)

col_blue(...)

col_cyan(...)

col_green(...)
```

```
col_magenta(...)

col_red(...)

col_white(...)

col_yellow(...)

col_grey(...)

col_silver(...)

style_dim(...)

style_blurred(...)

style_bold(...)

style_hidden(...)

style_inverse(...)

style_italic(...)

style_reset(...)

style_strikethrough(...)

style_underline(...)
```

Arguments

...	Character strings, they will be pasted together with `paste0()`, before applying the style function.

Details

The `col_*` functions change the (foreground) color to the text. These are the eight original ANSI colors. Note that in some terminals, they might actually look differently, as terminals have their own settings for how to show them.

The `bg_*` functions change the background color of the text. These are the eight original ANSI background colors. These, too, can vary in appearence, depending on terminal settings.

The `style_*` functions apply other styling to the text. The currently supported styling funtions are:

- `style_reset()` to remove any style, including color,
- `style_bold()` for boldface / strong text, although some terminals show a bright, high intensity text instead,

- `style_dim()` (or `style_blurred()` reduced intensity text.
- `style_italic()` (not widely supported).
- `style_underline()`,
- `style_inverse()`,
- `style_hidden()`,
- ‘style_strikethrough() (not widely supported).

The style functions take any number of character vectors as arguments, and they concatenate them using `paste0()` before adding the style.

Styles can also be nested, and then inner style takes precedence, see examples below.

Value

An ANSI string (class `ansi_string`), that contains ANSI sequences, if the current platform supports them. You can simply use `cat()` to print them to the terminal.

See Also

Other ANSI styling: `combine_ansi_styles`, `make_ansi_style`

Examples

```
col_blue("Hello ", "world!")
cat(col_blue("Hello ", "world!"))

cat("... to highlight the", col_red("search term"),
    "in a block of text\n")

## Style stack properly
cat(col_green(
 "I am a green line ",
 col_blue(style_underline(style_bold("with a blue substring"))),
 " that becomes green again!"
))

error <- combine_ansi_styles("red", "bold")
warn <- combine_ansi_styles("magenta", "underline")
note <- col_cyan
cat(error("Error: subscript out of bounds!\n"))
cat(warn("Warning: shorter argument was recycled.\n"))
cat(note("Note: no such directory.\n"))
```

ansi_hide_cursor *Hide/show cursor in a terminal*

Description

This only works in terminal emulators. In other environments, it does nothing.

Usage

```
ansi_hide_cursor(stream = stderr())

ansi_show_cursor(stream = stderr())

ansi_with_hidden_cursor(expr, stream = stderr())
```

Arguments

stream	The stream of the terminal to output the ANSI sequence to.
expr	R expression to evaluate.

Details

`ansi_hide_cursor()` hides the cursor.

`ansi_show_cursor()` shows the cursor.

`ansi_with_hidden_cursor()` temporarily hides the cursor for evaluating an expression.

cat_line `cat()` *helpers*

Description

These helpers provide useful wrappers around `cat()`: most importantly they all set `sep = ""`, and `cat_line()` automatically adds a newline.

Usage

```
cat_line(..., col = NULL, background_col = NULL, file = stdout())

cat_bullet(..., col = NULL, background_col = NULL, bullet = "bullet",
  bullet_col = NULL, file = stdout())

cat_boxx(..., file = stdout())

cat_rule(..., file = stdout())

cat_print(x, file = "")
```

Arguments

`...`	For `cat_line()` and `cat_bullet()`, paste'd together with `collapse = "\n"`. For `cat_rule()` and `cat_boxx()` passed on to `rule()` and `boxx()` respectively.
`col, background_col, bullet_col`	Colours for text, background, and bullets respectively.
`file`	Output destination. Defaults to standard output.
`bullet`	Name of bullet character. Indexes into symbol
`x`	An object to print.

Examples

```
cat_line("This is ", "a ", "line of text.", col = "red")
cat_bullet(letters[1:5])
cat_bullet(letters[1:5], bullet = "tick", bullet_col = "green")
cat_rule()
```

`cli_sitrep` *cli situation report*

Description

Contains currenty:

- `cli_unicode_option`: whether the `cli.unicode` option is set and its value. See `is_utf8_output()`.
- `symbol_charset`: the selected character set for symbol, UTF-8, Windows, or ASCII.
- `console_utf8`: whether the console supports UTF-8. See `base::l10n_info()`.
- `latex_active`: whether we are inside knitr, creating a LaTeX document.
- `num_colors`: number of ANSI colors. See `crayon::num_colors()`.
- `console_with`: detected console width.

Usage

```
cli_sitrep()
```

Value

Named list with entries listed above. It has a `cli_sitrep` class, with a `print()` and `format()` method.

Examples

```
cli_sitrep()
```

`combine_ansi_styles` *Combine two or more ANSI styles*

Description

Combine two or more styles or style functions into a new style function that can be called on strings to style them.

Usage

```
combine_ansi_styles(...)
```

Arguments

`...`	The styles to combine. For character strings, the `make_ansi_style()` function is used to create a style first. They will be applied from right to left.

Details

It does not usually make sense to combine two foreground colors (or two background colors), because only the first one applied will be used.

It does make sense to combine different kind of styles, e.g. background color, foreground color, bold font.

Value

The combined style function.

See Also

Other ANSI styling: `ansi-styles`, `make_ansi_style`

Examples

```
## Use style names
alert <- combine_ansi_styles("bold", "red4")
cat(alert("Warning!"), "\n")

## Or style functions
alert <- combine_ansi_styles(style_bold, col_red, bg_cyan)
cat(alert("Warning!"), "\n")

## Combine a composite style
alert <- combine_ansi_styles(
  "bold",
  combine_ansi_styles("red", bg_cyan))
cat(alert("Warning!"), "\n")
```

console_width *Determine the width of the console*

Description

It uses the `RSTUDIO_CONSOLE_WIDTH` environment variable, if set. Otherwise it uses the `width` option. If this is not set either, then 80 is used.

Usage

```
console_width()
```

Value

Integer scalar, the console with, in number of characters.

demo_spinners *Show a demo of some (by default all) spinners*

Description

Each spinner is shown for about 2-3 seconds.

Usage

```
demo_spinners(which = NULL)
```

Arguments

`which`	Character vector, which spinners to demo.

See Also

Other spinners: `get_spinner`, `list_spinners`, `make_spinner`

Examples

```
## Not run:
  demo_spinners(sample(list_spinners(), 10))

## End(Not run)
```

get_spinner *Character vector to put a spinner on the screen*

Description

`cli` contains many different spinners, you choose one according to your taste.

Usage

```
get_spinner(which = NULL)
```

Arguments

`which`	The name of the chosen spinner. The default depends on whether the platform supports Unicode.

Value

A list with entries: `name`, `interval`: the suggested update interval in milliseconds and `frames`: the character vector of the spinner's frames.

See Also

Other spinners: `demo_spinners`, `list_spinners`, `make_spinner`

Examples

```
get_spinner()
get_spinner("shark")
```

is_ansi_tty *Detect if a stream support ANSI escape characters*

Description

We check that all of the following hold:

- The stream is a terminal.
- The platform is Unix.
- R is not running inside R.app (the macOS GUI).
- R is not running inside RStudio.
- R is not running inside Emacs.
- The terminal is not "dumb".
- `stream` is either the standard output or the standard error stream.

Usage

```
is_ansi_tty(stream = stderr())
```

Arguments

stream	The stream to check.

Value

TRUE or FALSE.

See Also

Other terminal capabilities: is_dynamic_tty

Examples

```
is_ansi_tty()
```

---

is_dynamic_tty *Detect whether a stream supports* \r *(Carriage return)*

Description

In a terminal, \r moves the cursor to the first position of the same line. It is also supported by most R IDEs. \r is typically used to achive a more dynamic, less cluttered user interface, e.g. to create progress bars.

Usage

```
is_dynamic_tty(stream = stderr())
```

Arguments

stream	The stream to inspect, an R connection object. Note that it defaults to the standard *error* stream, since informative messages are typically printed there.

Details

If the output is directed to a file, then \r characters are typically unwanted. This function detects if \r can be used for the given stream or not.

The detection mechanism is as follows:

1. If the cli.dynamic option is set to TRUE, TRUE is returned.
2. If the cli.dynamic option is set to anything else, FALSE is returned.
3. If the R_CLI_DYNAMIC environment variable is not empty and set to the string "true", "TRUE" or "True", TRUE is returned.

4. If R_CLI_DYNAMIC is not empty and set to anything else, FALSE is returned.
5. If the stream is a terminal, then TRUE is returned.
6. If the stream is hte standard output or error within RStudio, the macOS R app, or RKWard IDE, TRUE is returned.
7. Otherwise FALSE is returned.

See Also

Other terminal capabilities: is_ansi_tty

Examples

```
is_dynamic_tty()
is_dynamic_tty(stdout())
```

---

is_utf8_output *Whether cli is emitting UTF-8 characters*

Description

UTF-8 cli characters can be turned on by setting the cli.unicode option to TRUE. They can be turned off by setting if to FALSE. If this option is not set, then base::l10n_info() is used to detect UTF-8 support.

Usage

```
is_utf8_output()
```

Value

Flag, whether cli uses UTF-8 characters.

---

list_border_styles *Draw a banner-like box in the console*

Description

Draw a banner-like box in the console

Usage

```
list_border_styles()

boxx(label, border_style = "single", padding = 1, margin = 0,
  float = c("left", "center", "right"), col = NULL,
  background_col = NULL, border_col = col, align = c("left",
  "center", "right"), width = console_width())
```

Arguments

label	Label to show, a character vector. Each element will be in a new line. You can color it using the col_*, bg_* and style_* functions, see ansi-styles and the examples below.
border_style	String that specifies the border style. list_border_styles lists all current styles.
padding	Padding within the box. Either an integer vector of four numbers (bottom, left, top, right), or a single number x, which is interpreted as c(x, 3*x, x, 3*x).
margin	Margin around the box. Either an integer vector of four numbers (bottom, left, top, right), or a single number x, which is interpreted as c(x, 3*x, x, 3*x).
float	Whether to display the box on the "left", "center", or the "right" of the screen.
col	Color of text, and default border color. Either a style function (see ansi-styles) or a color name that is passed to make_ansi_style().
background_col	Background color of the inside of the box. Either a style function (see ansi-styles), or a color name which will be used in make_ansi_style() to create a *background* style (i.e. bg = TRUE is used).
border_col	Color of the border. Either a style function (see ansi-styles) or a color name that is passed to make_ansi_style().
align	Alignment of the label within the box: "left", "center", or "right".
width	Width of the screen, defaults to getOption("width").

About fonts and terminal settings

The boxes might or might not look great in your terminal, depending on the box style you use and the font the terminal uses. We found that the Menlo font looks nice in most terminals an also in Emacs.

RStudio currently has a line height greater than one for console output, which makes the boxes ugly.

Examples

```
## Simple box
boxx("Hello there!")

## All border styles
list_border_styles()

## Change border style
boxx("Hello there!", border_style = "double")

## Multiple lines
boxx(c("Hello", "there!"), padding = 1)

## Padding
boxx("Hello there!", padding = 1)
boxx("Hello there!", padding = c(1, 5, 1, 5))
```

```
## Margin
boxx("Hello there!", margin = 1)
boxx("Hello there!", margin = c(1, 5, 1, 5))
boxx("Hello there!", padding = 1, margin = c(1, 5, 1, 5))

## Floating
boxx("Hello there!", padding = 1, float = "center")
boxx("Hello there!", padding = 1, float = "right")

## Text color
boxx(col_cyan("Hello there!"), padding = 1, float = "center")

## Backgorund color
boxx("Hello there!", padding = 1, background_col = "brown")
boxx("Hello there!", padding = 1, background_col = bg_red)

## Border color
boxx("Hello there!", padding = 1, border_col = "green")
boxx("Hello there!", padding = 1, border_col = col_red)

## Label alignment
boxx(c("Hi", "there", "you!"), padding = 1, align = "left")
boxx(c("Hi", "there", "you!"), padding = 1, align = "center")
boxx(c("Hi", "there", "you!"), padding = 1, align = "right")

## A very customized box
star <- symbol$star
label <- c(paste(star, "Hello", star), "  there!")
boxx(
  col_white(label),
  border_style="round",
  padding = 1,
  float = "center",
  border_col = "tomato3",
  background_col="darkolivegreen"
)
```

`list_spinners` *List all available spinners*

Description

List all available spinners

Usage

```
list_spinners()
```

Value

Character vector of all available spinner names.

See Also

Other spinners: demo_spinners, get_spinner, make_spinner

Examples

```
list_spinners()
get_spinner(list_spinners()[1])
```

`make_ansi_style` *Create a new ANSI style*

Description

Create a function that can be used to add ANSI styles to text. All arguments are passed to `crayon::make_style()`, but see the Details below.

Usage

```
make_ansi_style(..., bg = FALSE, grey = FALSE,
  colors = crayon::num_colors())
```

Arguments

`...`	The style to create. See details and examples below.
`bg`	Whether the color applies to the background.
`grey`	Whether to specifically create a grey color. This flag is included, because ANSI 256 has a finer color scale for greys, then the usual 0:5 scale for red, green and blue components. It is only used for RGB color specifications (either numerically or via a hexa string), and it is ignored on eigth color ANSI terminals.
`colors`	Number of colors, detected automatically by default.

Details

The styles (elements of `...`) can be any of the following:

- An R color name, see `grDevices::colors()`.
- A 6- or 8-digit hexa color string, e.g. `#ff0000` means red. Transparency (alpha channel) values are ignored.
- A one-column matrix with three rows for the red, green and blue channels, as returned by `grDevices::col2rgb()`.

`make_ansistyle()` detects the number of colors to use automatically (this can be overridden using the `colors` argument). If the number of colors is less than 256 (detected or given), then it falls back to the color in the ANSI eight color mode that is closest to the specified (RGB or R) color.

Value

A function that can be used to color (style) strings.

See Also

Other ANSI styling: ansi-styles, combine_ansi_styles

Examples

```
make_ansi_style("orange")
make_ansi_style("#123456")
make_ansi_style("orange", bg = TRUE)

orange <- make_ansi_style("orange")
orange("foobar")
cat(orange("foobar"))
```

---

make_spinner *Create a spinner*

Description

Create a spinner

Usage

```
make_spinner(which = NULL, stream = stderr(), template = "{spin}",
  static = c("dots", "print", "print_line", "silent"))
```

Arguments

which	The name of the chosen spinner. The default depends on whether the platform supports Unicode.
stream	The stream to use for the spinner. Typically this is standard error, or maybe the standard output stream.
template	A template string, that will contain the spinner. The spinner itself will be substituted for {spin}. See example below.
static	What to do if the terminal does not support dynamic displays: • "dots": show a dot for each $spin() call. • "print": just print the frames of the spinner, one after another. • "print_line": print the frames of the spinner, each on its own line. • "silent" do not print anything, just the template.

Value

A `cli_spinner` object, which is a list of functions. See its methods below.

`cli_spinner` methods:

- `$spin()`: output the next frame of the spinner.
- `$finish()`: terminate the spinner. Depending on terminal capabilities this removes the spinner from the screen. Spinners can be reused, you can start calling the `$spin()` method again.

All methods return the spinner object itself, invisibly.

The spinner is automatically throttled to its ideal update frequency.

Examples

```
## Default spinner
sp1 <- make_spinner()
fun_with_spinner <- function() {
  lapply(1:100, function(x) { sp1$spin(); Sys.sleep(0.05) })
  sp1$finish()
}
ansi_with_hidden_cursor(fun_with_spinner())

## Spinner with a template
sp2 <- make_spinner(template = "Computing {spin}")
fun_with_spinner2 <- function() {
  lapply(1:100, function(x) { sp2$spin(); Sys.sleep(0.05) })
  sp2$finish()
}
ansi_with_hidden_cursor(fun_with_spinner2())

## Custom spinner
sp3 <- make_spinner("simpleDotsScrolling", template = "Downloading {spin}")
fun_with_spinner3 <- function() {
  lapply(1:100, function(x) { sp3$spin(); Sys.sleep(0.05) })
  sp2$finish()
}
ansi_with_hidden_cursor(fun_with_spinner3())
```

See Also

Other spinners: `demo_spinners`, `get_spinner`, `list_spinners`

`rule` *Make a rule with one or two text labels*

Description

The rule can include either a centered text label, or labels on the left and right side.

Usage

```
rule(left = "", center = "", right = "", line = 1, col = NULL,
  line_col = col, background_col = NULL, width = console_width())
```

Arguments

left	Label to show on the left. It interferes with the center label, only at most one of them can be present.
center	Label to show at the center. It interferes with the left and right labels.
right	Label to show on the right. It interferes with the center label, only at most one of them can be present.
line	The character or string that is used to draw the line. It can also 1 or 2, to request a single line (Unicode, if available), or a double line. Some strings are interpreted specially, see *Line styles* below.
col	Color of text, and default line color. Either an ANSI style function (see ansi-styles), or a color name that is passed to make_ansi_style().
line_col, background_col	Either a color name (used in make_ansi_style()), or a style function (see ansi-styles), to color the line and background.
width	Width of the rule. Defaults to the width option, see base::options().

Details

To color the labels, use the functions col_*, bg_* and style_* functions, see ansi-styles, and the examples below. To color the line, either these functions directly, or the line_col option.

Value

Character scalar, the rule.

Line styles

Some strings for the line argument are interpreted specially:

- "single": (same as 1), a single line,
- "double": (same as 2), a double line,
- "bar1", "bar2", "bar3", etc., "bar8" uses varying height bars.

Examples

```
## Simple rule
rule()

## Double rule
rule(line = 2)

## Bars
```

```
rule(line = "bar2")
rule(line = "bar5")

## Left label
rule(left = "Results")

## Centered label
rule(center = " * RESULTS * ")

## Colored labels
rule(center = col_red(" * RESULTS * "))

## Colored line
rule(center = col_red(" * RESULTS * "), line_col = "red")

## Custom line
rule(center = "TITLE", line = "~")

## More custom line
rule(center = "TITLE", line = col_blue("~-"))

## Even more custom line
rule(center = bg_red(" ", symbol$star, "TITLE",
  symbol$star, " "),
  line = "\u2582",
  line_col = "orange")
```

`symbol` *Various handy symbols to use in a command line UI*

Description

Various handy symbols to use in a command line UI

Usage

```
symbol
```

Format

A named list, see `names(symbol)` for all sign names.

Details

On Windows they have a fallback to less fancy symbols.

Examples

```
cat(symbol$tick, " SUCCESS\n", symbol$cross, " FAILURE\n", sep = "")

## All symbols
cat(paste(format(names(symbol), width = 20),
  unlist(symbol)), sep = "\n")
```

`tree` *Draw a tree*

Description

Draw a tree using box drawing characters. Unicode characters are used if available. (Set the `cli.unicode` option if auto-detection fails.)

Usage

```
tree(data, root = data[[1]][[1]], style = NULL,
  width = console_width())
```

Arguments

`data`	Data frame that contains the tree structure. The first column is an id, and the second column is a list column, that contains the ids of the child nodes. The optional third column may contain the text to print to annotate the node.
`root`	The name of the root node.
`style`	Optional box style list.
`width`	Maximum width of the output. Defaults to the `width` option, see `base::options()`.

Details

A node might appear multiple times in the tree, or might not appear at all.

Value

Character vector, the lines of the tree drawing.

Examples

```
data <- data.frame(
  stringsAsFactors = FALSE,
  package = c("processx", "backports", "assertthat", "Matrix",
    "magrittr", "rprojroot", "clisymbols", "prettyunits", "withr",
    "desc", "igraph", "R6", "crayon", "debugme", "digest", "irlba",
    "rcmdcheck", "callr", "pkgconfig", "lattice"),
  dependencies = I(list(
    c("assertthat", "crayon", "debugme", "R6"), character(0),
    character(0), "lattice", character(0), "backports", character(0),
```

```
    c("magrittr", "assertthat"), character(0),
    c("assertthat", "R6", "crayon", "rprojroot"),
    c("irlba", "magrittr", "Matrix", "pkgconfig"), character(0),
    character(0), "crayon", character(0), "Matrix",
    c("callr", "clisymbols", "crayon", "desc", "digest", "prettyunits",
      "R6", "rprojroot", "withr"),
    c("processx", "R6"), character(0), character(0)
  ))
)
tree(data)
tree(data, root = "rcmdcheck")

## Colored nodes
data$label <- paste(data$package,
  style_dim(paste0("(", c("2.0.0.1", "1.1.1", "0.2.0", "1.2-11",
    "1.5", "1.2", "1.2.0", "1.0.2", "2.0.0", "1.1.1.9000", "1.1.2",
    "2.2.2", "1.3.4", "1.0.2", "0.6.12", "2.2.1", "1.2.1.9002",
    "1.0.0.9000", "2.0.1", "0.20-35"), ")"))
  )
roots <- ! data$package %in% unlist(data$dependencies)
data$label[roots] <- col_cyan(style_italic(data$label[roots]))
tree(data)
tree(data, root = "rcmdcheck")
```

Index