Package 'crayon'

September 29, 2022

Title Colored Terminal Output

Version 1.5.2

Description The crayon package is now superseded. Please use the 'cli' package for new projects.
Colored terminal output on terminals that support 'ANSI' color and highlight codes. It also works in 'Emacs' 'ESS'. 'ANSI' color support is automatically detected. Colors and highlighting can be combined and nested. New styles can also be created easily.
This package was inspired by the 'chalk' 'JavaScript' project.

License MIT + file LICENSE

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

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

Collate 'aaa-rstudio-detect.R' 'aaaa-rematch2.R' 'aab-num-ansi-colors.R' 'aac-num-ansi-colors.R' 'ansi-256.r' 'ansi-palette.R' 'combine.r' 'string.r' 'utils.r' 'crayon-package.r' 'disposable.r' 'enc-utils.R' 'has_ansi.r' 'has_color.r' 'link.R' 'styles.r' 'machinery.r' 'parts.r' 'print.r' 'style-var.r' 'show.r' 'string_operations.r'

Imports grDevices, methods, utils

Suggests mockery, rstudioapi, testthat, withr

RoxygenNote 7.1.2

Encoding UTF-8

NeedsCompilation no

Author Gábor Csárdi [aut, cre], Brodie Gaslam [ctb]

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

Repository CRAN

Date/Publication 2022-09-29 16:20:24 UTC

R topics documented:

`chr` *Convert to character*

Description

This function just calls `as.character()`, but it is easier to type and read.

Usage

```
chr(x, ...)
```

Arguments

`x`	Object to be coerced.
`...`	Further arguments to pass to `as.character()`.

Value

Character value.

col_align *Align an ANSI colored string*

Description

Align an ANSI colored string

Usage

```
col_align(
  text,
  width = getOption("width"),
  align = c("left", "center", "right"),
  type = "width"
)
```

Arguments

text	The character vector to align.
width	Width of the field to align in.
align	Whether to align "left", "center" or "right".
type	Passed on to col_nchar() and there to nchar()

Value

The aligned character vector.

See Also

Other ANSI string operations: col_nchar(), col_strsplit(), col_substring(), col_substr()

Examples

```
col_align(red("foobar"), 20, "left")
col_align(red("foobar"), 20, "center")
col_align(red("foobar"), 20, "right")
```

col_nchar *Count number of characters in an ANSI colored string*

Description

This is a color-aware counterpart of `base::nchar()`, which does not do well, since it also counts the ANSI control characters.

Usage

```
col_nchar(x, ...)
```

Arguments

x	Character vector, potentially ANSO styled, or a vector to be coarced to character.
...	Additional arguments, passed on to `base::nchar()` after removing ANSI escape sequences.

Value

Numeric vector, the length of the strings in the character vector.

See Also

Other ANSI string operations: `col_align()`, `col_strsplit()`, `col_substring()`, `col_substr()`

Examples

```
str <- paste(
  red("red"),
  "default",
  green("green")
)

cat(str, "\n")
nchar(str)
col_nchar(str)
nchar(strip_style(str))
```

`col_strsplit` *Split an ANSI colored string*

Description

This is the color-aware counterpart of `base::strsplit()`. It works almost exactly like the original, but keeps the colors in the substrings.

Usage

```
col_strsplit(x, split, ...)
```

Arguments

`x`	Character vector, potentially ANSI styled, or a vector to coarced to character.
`split`	Character vector of length 1 (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 zero characters, `x` is split into single characters.
`...`	Extra arguments are passed to `base::strsplit()`.

Value

A list of the same length as `x`, the i-th element of which contains the vector of splits of `x[i]`. ANSI styles are retained.

See Also

Other ANSI string operations: `col_align()`, `col_nchar()`, `col_substring()`, `col_substr()`

Examples

```
str <- red("I am red---") %+%
  green("and I am green-") %+%
  underline("I underlined")

cat(str, "\n")

# split at dashes, keep color
cat(col_strsplit(str, "[-]+")[[1]], sep = "\n")
strsplit(strip_style(str), "[-]+")

# split to characters, keep color
cat(col_strsplit(str, "")[[1]], "\n", sep = " ")
strsplit(strip_style(str), "")
```

col_substr *Substring(s) of an ANSI colored string*

Description

This is a color-aware counterpart of base::substr(). It works exactly like the original, but keeps the colors in the substrings. The ANSI escape sequences are ignored when calculating the positions within the string.

Usage

```
col_substr(x, start, stop)
```

Arguments

x	Character vector, potentially ANSI styled, or a vector to coarced to character.
start	Starting index or indices, recycled to match the length of x.
stop	Ending index or indices, recycled to match the length of x.

Value

Character vector of the same length as x, containing the requested substrings. ANSI styles are retained.

See Also

Other ANSI string operations: col_align(), col_nchar(), col_strsplit(), col_substring()

Examples

```
str <- paste(
  red("red"),
  "default",
  green("green")
)

cat(str, "\n")
cat(col_substr(str, 1, 5), "\n")
cat(col_substr(str, 1, 15), "\n")
cat(col_substr(str, 3, 7), "\n")

substr(strip_style(str), 1, 5)
substr(strip_style(str), 1, 15)
substr(strip_style(str), 3, 7)

str2 <- "another " %+%
  red("multi-", sep = "", underline("style")) %+%
  " text"
```

```
cat(str2, "\n")
cat(col_substr(c(str, str2), c(3,5), c(7, 18)), sep = "\n")
substr(strip_style(c(str, str2)), c(3,5), c(7, 18))
```

`col_substring` *Substring(s) of an ANSI colored string*

Description

This is the color-aware counterpart of `base::substring()`. It works exactly like the original, but keeps the colors in the substrings. The ANSI escape sequences are ignored when calculating the positions within the string.

Usage

```
col_substring(text, first, last = 1000000L)
```

Arguments

`text`	Character vector, potentially ANSI styled, or a vector to coarced to character. It is recycled to the longest of `first` and `last`.
`first`	Starting index or indices, recycled to match the length of `x`.
`last`	Ending index or indices, recycled to match the length of `x`.

Value

Character vector of the same length as `x`, containing the requested substrings. ANSI styles are retained.

See Also

Other ANSI string operations: `col_align()`, `col_nchar()`, `col_strsplit()`, `col_substr()`

Examples

```
str <- paste(
  red("red"),
  "default",
  green("green")
)

cat(str, "\n")
cat(col_substring(str, 1, 5), "\n")
cat(col_substring(str, 1, 15), "\n")
cat(col_substring(str, 3, 7), "\n")

substring(strip_style(str), 1, 5)
```

```
substring(strip_style(str), 1, 15)
substring(strip_style(str), 3, 7)

str2 <- "another " %+%
  red("multi-", sep = "", underline("style")) %+%
  " text"

cat(str2, "\n")
cat(col_substring(str2, c(3,5), c(7, 18)), sep = "\n")
substring(strip_style(str2), c(3,5), c(7, 18))
```

`combine_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_styles(...)

## S3 method for class 'crayon'
crayon$style
```

Arguments

`...`	The styles to combine. They will be applied from right to left.
`crayon`	A style function.
`style`	A style name that is included in `names(styles())`.

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.

The `$` operator can also be used to combine styles. Note that the left hand side of `$` is a style function, and the right hand side is the name of a style in `styles()`.

Value

The combined style function.

Examples

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

## Or style functions
alert <- combine_styles(bold, red, bgCyan)
cat(alert("Warning!"), "\n")

## Combine a composite style
alert <- combine_styles(bold, combine_styles(red, bgCyan))
cat(alert("Warning!"), "\n")

## Shorter notation
alert <- bold $ red $ bgCyan
cat(alert("Warning!"), "\n")
```

---

concat *Concatenate character vectors*

Description

The length of the two arguments must match, or one of them must be of length one. If the length of one argument is one, then the output's length will match the length of the other argument. See examples below.

Usage

```
lhs %+% rhs
```

Arguments

lhs	Left hand side, character vector.
rhs	Right hand side, character vector.

Value

Concatenated vectors.

Examples

```
"foo" %+% "bar"

letters[1:10] %+% chr(1:10)

letters[1:10] %+% "-" %+% chr(1:10)

## This is empty (unlike for parse)
character() %+% "*"
```

`crayon` *Colored terminal output*

Description

With crayon it is easy to add color to terminal output, create styles for notes, warnings, errors; and combine styles.

Usage

```
## Simple styles
red(...)
bold(...)
# ...

## See more styling below
```

Arguments

...	Strings to style.

Details

ANSI color support is automatically detected and used. Crayon was largely inspired by chalk https://github.com/chalk/chalk.

Crayon defines several styles, that can be combined. Each style in the list has a corresponding function with the same name.

Genaral styles

- reset
- bold
- blurred (usually called 'dim', renamed to avoid name clash)
- italic (not widely supported)
- underline
- inverse
- hidden
- strikethrough (not widely supported)

Text colors

- black
- red
- green

- yellow
- blue
- magenta
- cyan
- white
- silver (usually called 'gray', renamed to avoid name clash)

Background colors

- bgBlack
- bgRed
- bgGreen
- bgYellow
- bgBlue
- bgMagenta
- bgCyan
- bgWhite

Styling

The styling functions take any number of character vectors as arguments, and they concatenate and style them:

```
library(crayon)
cat(blue("Hello", "world!\n"))
```

Crayon defines the `%+%` string concatenation operator, to make it easy to assemble stings with different styles.

```
cat("... to highlight the " %+% red("search term") %+%
    " in a block of text\n")
```

Styles can be combined using the `$` operator:

```
cat(yellow$bgMagenta$bold('Hello world!\n'))
```

See also `combine_styles()`.

Styles can also be nested, and then inner style takes precedence:

```
cat(green(
  'I am a green line ' %+%
  blue$underline$bold('with a blue substring') %+%
  ' that becomes green again!\n'
))
```

It is easy to define your own themes:

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

See Also

`make_style()` for using the 256 ANSI colors.

Examples

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

cat("... to highlight the " %+% red("search term") %+%
    " in a block of text")

cat(yellow$bgMagenta$bold('Hello world!'))

cat(green(
 'I am a green line ' %+%
 blue$underline$bold('with a blue substring') %+%
 ' that becomes green again!'
))

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

---

drop_style *Remove a style*

Description

Remove a style

Usage

```
drop_style(style)
```

Arguments

style	The name of the style to remove. No error is given for non-existing names.

Value

Nothing.

See Also

Other styles: `make_style()`

Examples

```
make_style(new_style = "maroon", bg = TRUE)
cat(style("I am maroon", "new_style"), "\n")
drop_style("new_style")
"new_style" %in% names(styles())
```

---

has_color *Does the current R session support ANSI colors?*

Description

From crayon 2.0.0, this function is simply a wrapper on `num_ansi_colors()`.

Usage

```
has_color()
```

Value

`TRUE` if the current R session supports color.

Examples

```
has_color()
```

---

has_style *Check if a string has some ANSI styling*

Description

Check if a string has some ANSI styling

Usage

```
has_style(string)
```

Arguments

`string`	The string to check. It can also be a character vector.

Value

Logical vector, TRUE for the strings that have some ANSI styling.

Examples

```
## The second one has style if crayon is enabled
has_style("foobar")
has_style(red("foobar"))
```

---

hyperlink *Terminal Hyperlinks*

Description

Terminal Hyperlinks

Usage

```
hyperlink(text, url)

has_hyperlink()
```

Arguments

text	Text to show. text and url are recycled to match their length, via a paste0() call.
url	URL to link to.

Details

hyperlink()‘ creates an ANSI hyperlink.

has_hyperlink() checks if the current stdout() supports hyperlinks. terminal links.

See also https://gist.github.com/egmontkob/eb114294efbcd5adb1944c9f3cb5feda.

Value

Logical scalar, for has_hyperlink().

Examples

```
cat("This is an", hyperlink("R", "https://r-project.org"), "link.\n")
has_hyperlink()
```

`make_style` *Create an ANSI color style*

Description

Create a style, or a style function, or both. This function is intended for those who wish to use 256 ANSI colors, instead of the more widely supported eight colors.

Usage

```
make_style(..., bg = FALSE, grey = FALSE, colors = 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 than the usual 0:5 scale for R, G and B components. It is only used for RGB color specifications (either numerically or via a hexa string) and is ignored on eigth color ANSI terminals.
`colors`	Number of colors, detected automatically by default.

Details

The crayon package comes with predefined styles (see `styles()` for a list) and functions for the basic eight-color ANSI standard (`red`, `blue`, etc., see crayon).

There are no predefined styles or style functions for the 256 color ANSI mode, however, because we simply did not want to create that many styles and functions. Instead, `make_style()` can be used to create a style (or a style function, or both).

There are two ways to use this function:

1. If its first argument is not named, then it returns a function that can be used to color strings.
2. If its first argument is named, then it also creates a style with the given name. This style can be used in `style()`. One can still use the return value of the function, to create a style function.

The style (the code... argument) can be anything of the following:

- An R color name, see `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 `col2rgb` (in the base grDevices package).

`make_style()` 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.

See the examples below.

Value

A function that can be used to color strings.

See Also

Other styles: `drop_style()`

Examples

```
## Create a style function without creating a style
pink <- make_style("pink")
bgMaroon <- make_style(rgb(0.93, 0.19, 0.65), bg = TRUE)
cat(bgMaroon(pink("I am pink if your terminal wants it, too.\n")))

## Create a new style for pink and maroon background
make_style(pink = "pink")
make_style(bgMaroon = rgb(0.93, 0.19, 0.65), bg = TRUE)
"pink" %in% names(styles())
"bgMaroon" %in% names(styles())
cat(style("I am pink, too!\n", "pink", bg = "bgMaroon"))
```

---

num_ansi_colors *Detect the number of ANSI colors to use*

Description

Certain Unix and Windows terminals, and also certain R GUIs, e.g. RStudio, support styling terminal output using special control sequences (ANSI sequences).

`num_ansi_colors()` detects if the current R session supports ANSI sequences, and if it does how many colors are supported.

Usage

```
num_ansi_colors(stream = "auto")

detect_tty_colors()
```

Arguments

`stream`	The stream that will be used for output, an R connection object. It can also be a string, one of `"auto"`, `"message"`, `"stdout"`, `"stderr"`. `"auto"` will select `stdout()` if the session is interactive and there are no sinks, otherwise it will select `stderr()`.

Details

The detection mechanism is quite involved and it is designed to work out of the box on most systems. If it does not work on your system, please report a bug. Setting options and environment variables to turn on ANSI support is error prone, because they are inherited in other environments, e.g. knitr, that might not have ANSI support.

If you want to *turn off* ANSI colors, set the `NO_COLOR` environment variable to a non-empty value.

The exact detection mechanism is as follows:

1. If the `cli.num_colors` options is set, that is returned.
2. If the `R_CLI_NUM_COLORS` environment variable is set to a non-empty value, then it is used.
3. If the `crayon.enabled` option is set to `FALSE`, 1L is returned. (This is for compatibility with code that uses the crayon package.)
4. If the `crayon.enabled` option is set to `TRUE` and the `crayon.colors` option is not set, then the value of the `cli.default_num_colors` option, or if it is unset, then 8L is returned.
5. If the `crayon.enabled` option is set to `TRUE` and the `crayon.colors` option is also set, then the latter is returned. (This is for compatibility with code that uses the crayon package.)
6. If the `NO_COLOR` environment variable is set, then 1L is returned.
7. If we are in knitr, then 1L is returned, to turn off colors in `.Rmd` chunks.
8. If `stream` is `"auto"` (the default) and there is an active sink (either for `"output"` or `"message"`), then we return 1L. (In theory we would only need to check the stream that will be be actually used, but there is no easy way to tell that.)
9. If `stream` is not `"auto"`, but it is `stderr()` and there is an active sink for it, then 1L is returned. (If a sink is active for "output", then R changes the `stdout()` stream, so this check is not needed.)
10. If R is running inside RGui on Windows, or R.app on macOS, then we return 1L.
11. If R is running inside RStudio, with color support, then the appropriate number of colors is returned, usually 256L.
12. If R is running on Windows, inside an Emacs version that is recent enough to support ANSI colors, then the value of the `cli.default_num_colors` option, or if unset 8L is returned. (On Windows, Emacs has `isatty(stdout()) == FALSE`, so we need to check for this here before dealing with terminals.)
13. If `stream` is not the standard output or standard error in a terminal, then 1L is returned.
14. Otherwise we use and cache the result of the terminal color detection (see below).

The terminal color detection algorithm:

1. If the `COLORTERM` environment variable is set to `truecolor` or `24bit`, then we return 16 million colors.
2. If the `COLORTERM` environment variable is set to anything else, then we return the value of the `cli.num_default_colors` option, 8L if unset.
3. If R is running on Unix, inside an Emacs version that is recent enough to support ANSI colors, then the value of the `cli.default_num_colors` option is returned, or 8L if unset.
4. If we are on Windows in an RStudio terminal, then apparently we only have eight colors, but the `cli.default_num_colors` option can be used to override this.

5. If we are in a recent enough Windows 10 terminal, then there is either true color (from build 14931) or 256 color (from build 10586) support. You can also use the `cli.default_num_colors` option to override these.
6. If we are on Windows, under ConEmu or cmder, or ANSICON is loaded, then the value of `cli.default_num_colors`, or 8L if unset, is returned.
7. Otherwise if we are on Windows, return 1L.
8. Otherwise we are on Unix and try to run `tput colors` to determine the number of colors. If this succeeds, we return its return value. If the `TERM` environment variable is `xterm` and `tput` returned 8L, we return 256L, because xterm compatible terminals tend to support 256 colors (https://github.com/r-lib/crayon/issues/17) You can override this with the `cli.default_num_colors` option.
9. If `TERM` is set to `dumb`, we return 1L.
10. If `TERM` starts with `screen`, `xterm`, or `vt100`, we return 8L.
11. If `TERM` contains `color`, `ansi`, `cygwin` or `linux`, we return 8L.
12. Otherwise we return 1L.

Value

Integer, the number of ANSI colors the current R session supports for `stream`.

Examples

```
num_ansi_colors()
```

`num_colors` *Number of colors the terminal supports*

Description

From crayon version 2.0.0, this function is a simple wrapper on `num_ansi_colors()`, with the additional twist that the `crayon.colors` option is still obseved, and takes precedence, for compatibility.

Usage

```
num_colors(forget = FALSE)
```

Arguments

`forget`	Ignored. Included for backwards compatibility.

Value

Number of ANSI colors.

Examples

```
num_colors()
```

---

show_ansi_colors *Show the ANSI color table on the screen*

Description

Show the ANSI color table on the screen

Usage

```
show_ansi_colors(colors = num_colors())
```

Arguments

colors	Number of colors to show, meaningful values are 8 and 256. It is automatically set to the number of supported colors, if not specified.

Value

The printed string, invisibly.

---

start.crayon *Switch on or off a style*

Description

Make a style active. The text printed to the screen from now on will use this style.

Usage

```
## S3 method for class 'crayon'
start(x, ...)

finish(x, ...)

## S3 method for class 'crayon'
finish(x, ...)
```

Arguments

x	Style.
...	Ignored.

Details

This function is very rarely needed, e.g. for colored user input. For other reasons, just call the style as a function on the string.

Examples

```
## The input is red (if color is supported)
get_name <- function() {
  cat("Enter your name:", start(red))
  input <- readline()
  cat(finish(red))
  input
}
name <- get_name()
name
```

`strip_style` *Remove ANSI escape sequences from a string*

Description

Remove ANSI escape sequences from a string

Usage

```
strip_style(string)
```

Arguments

`string`	The input string.

Value

The cleaned up string.

Examples

```
strip_style(red("foobar")) == "foobar"
```

style *Add style to a string*

Description

See `names(styles)`, or the crayon manual for available styles.

Usage

```
style(string, as = NULL, bg = NULL)
```

Arguments

`string`	Character vector to style.
`as`	Style function to apply, either the function object, or its name, or an object to pass to `make_style()`.
`bg`	Background style, a style function, or a name that is passed to `make_style()`.

Value

Styled character vector.

Examples

```
## These are equivalent
style("foobar", bold)
style("foobar", "bold")
bold("foobar")
```

styles *ANSI escape sequences of crayon styles*

Description

You can use this function to list all availables crayon styles, via `names(styles())`, or to explicitly apply an ANSI escape seauence to a string.

Usage

```
styles()
```

Value

A named list. Each list element is a list of two strings, named ‘open’ and ‘close’.

See Also

`crayon()` for the beginning of the crayon manual.

Examples

```
names(styles())
cat(styles()[["bold"]]$close)
```

Index