Package 'cli'

July 17, 2021

Title Helpers for Developing Command Line Interfaces

Version 3.0.1

Description A suite of tools to build attractive command line interfaces ('CLIs'), from semantic elements: headings, lists, alerts, paragraphs, etc. Supports custom themes via a 'CSS'-like language. It also contains a number of lower level 'CLI' elements: rules, boxes, trees, and 'Unicode' symbols with 'ASCII' alternatives. It support ANSI colors and text styles as well.

License MIT + file LICENSE

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

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

RoxygenNote 7.1.1.9001

Depends R (>= 2.10)

Imports glue, utils

Suggests callr, covr, grDevices, htmlwidgets, knitr, methods, mockery, prettycode (>= 1.1.0), processx, ps (>= 1.3.4.9000), rlang, rmarkdown, rstudioapi, shiny, testthat, tibble, withr

Config/testthat/edition 3

Encoding UTF-8

NeedsCompilation yes

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

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

Repository CRAN

Date/Publication 2021-07-17 09:00:01 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. They provide a mode modern interface than the crayon package.

Usage

```
bg_black(...)

bg_blue(...)

bg_cyan(...)

bg_green(...)

bg_magenta(...)

bg_red(...)

bg_white(...)

bg_yellow(...)

bg_none(...)

col_black(...)

col_blue(...)

col_cyan(...)

col_green(...)

col_magenta(...)

col_red(...)

col_white(...)

col_yellow(...)

col_grey(...)

col_silver(...)
```

```
col_none(...)

style_dim(...)

style_blurred(...)

style_bold(...)

style_hidden(...)

style_inverse(...)

style_italic(...)

style_reset(...)

style_strikethrough(...)

style_underline(...)

style_no_bold(...)

style_no_blurred(...)

style_no_dim(...)

style_no_italic(...)

style_no_underline(...)

style_no_inverse(...)

style_no_hidden(...)

style_no_strikethrough(...)

style_no_color(...)

style_no_bg_color(...)
```

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. `col_none()` is the default color, this is useful in a substring of a colored string.

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. `bg_none()` the the default background color, this is useful in a substring of a background-colored string.

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.

Sometimes you want to revert back to the default text color, in the middle of colored text, or you want to have a normal font in the middle of italic text. You can use the style_no_* functions for this. Every style_*() function has a style_no_*() pair, which defends its argument from taking on the style. 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()`, `num_ansi_colors()`

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"))

# style_no_* functions, note that the color is not removed
style_italic(col_green(paste0(
  "italic before, ",
  style_no_italic("normal here, "),
  "italic after"
)))

# avoiding  color for substring
style_italic(col_red(paste(
  "red before",
  col_none("not red between"),
  "red after"
)))
```

`ansi_align` *Align an ANSI colored string*

Description

Align an ANSI colored string

Usage

```
ansi_align(
  text,
  width = console_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 `ansi_nchar()` and there to `nchar()`

Value

The aligned character vector.

See Also

Other ANSI string operations: ansi_columns(), ansi_nchar(), ansi_strsplit(), ansi_strtrim(), ansi_strwrap(), ansi_substring(), ansi_substr(), ansi_toupper(), ansi_trimws()

Examples

```
ansi_align(col_red("foobar"), 20, "left")
ansi_align(col_red("foobar"), 20, "center")
ansi_align(col_red("foobar"), 20, "right")
```

---

ansi_columns *Format a character vector in multiple columns*

Description

This function helps with multi-column output of ANSI styles strings. It works well together with boxx(), see the example below.

Usage

```
ansi_columns(
  text,
  width = console_width(),
  sep = " ",
  fill = c("rows", "cols"),
  max_cols = 4,
  align = c("left", "center", "right"),
  type = "width",
  ellipsis = symbol$ellipsis
)
```

Arguments

text	Character vector to format. Each element will formatted as a cell of a table.
width	Width of the screen.
sep	Separator between the columns. It may have ANSI styles.
fill	Whether to fill the columns row-wise or column-wise.
max_cols	Maximum number of columns to use. Will not use more, even if there is space for it.
align	Alignment within the columns.
type	Passed to ansi_nchar() and ansi_align(). Most probably you want the default, "width".
ellipsis	The string to append to truncated strings. Supply an empty string if you don't want a marker.

Details

If a string does not fit into the specified `width`, it will be truncated using `ansi_strtrim()`.

Value

ANSI string vector.

See Also

Other ANSI string operations: `ansi_align()`, `ansi_nchar()`, `ansi_strsplit()`, `ansi_strtrim()`, `ansi_strwrap()`, `ansi_substring()`, `ansi_substr()`, `ansi_toupper()`, `ansi_trimws()`

Examples

```
fmt <- ansi_columns(
  paste(col_red("foo"), 1:10),
  width = 50,
  fill = "rows",
  max_cols=10,
  align = "center",
  sep = "   "
)
fmt
ansi_nchar(fmt, type = "width")
boxx(fmt, padding = c(0,1,0,1), header = col_green("foobar"))
```

---

ansi_has_any *Check if a string has some ANSI styling*

Description

Check if a string has some ANSI styling

Usage

```
ansi_has_any(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.

See Also

Other low level ANSI functions: `ansi_hide_cursor()`, `ansi_regex()`, `ansi_strip()`

Examples

```
## The second one has style if ANSI colors are supported
ansi_has_any("foobar")
ansi_has_any(col_red("foobar"))
```

`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 = "auto")

ansi_show_cursor(stream = "auto")

ansi_with_hidden_cursor(expr, stream = "auto")
```

Arguments

`stream`	The stream to inspect or manipulate, 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()`.
`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.

See Also

Other low level ANSI functions: `ansi_has_any()`, `ansi_regex()`, `ansi_strip()`

ansi_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

```
ansi_nchar(x, type = c("chars", "bytes", "width"), ...)
```

Arguments

x	Character vector, potentially ANSO styled, or a vector to be coarced to character.
type	Whether to count characters, bytes, or calculate the display width of the string. Passed to base::nchar().
...	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: ansi_align(), ansi_columns(), ansi_strsplit(), ansi_strtrim(), ansi_strwrap(), ansi_substring(), ansi_substr(), ansi_toupper(), ansi_trimws()

Examples

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

cat(str, "\n")
nchar(str)
ansi_nchar(str)
nchar(ansi_strip(str))
```

ansi_regex *Perl comparible regular expression that matches ANSI escape sequences*

Description

Don't forget to use `perl = TRUE` when using this with `grepl()` and friends.

Usage

```
ansi_regex()
```

Value

String scalar, the regular expression.

See Also

Other low level ANSI functions: `ansi_has_any()`, `ansi_hide_cursor()`, `ansi_strip()`

ansi_strip *Remove ANSI escape sequences from a string*

Description

The input may be of class `ansi_string` class, this is also dropped from the result.

Usage

```
ansi_strip(string)
```

Arguments

`string`	The input string.

Value

The cleaned up string.

See Also

Other low level ANSI functions: `ansi_has_any()`, `ansi_hide_cursor()`, `ansi_regex()`

Examples

```
ansi_strip(col_red("foobar")) == "foobar"
```

`ansi_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

```
ansi_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: `ansi_align()`, `ansi_columns()`, `ansi_nchar()`, `ansi_strtrim()`, `ansi_strwrap()`, `ansi_substring()`, `ansi_substr()`, `ansi_toupper()`, `ansi_trimws()`

Examples

```
str <- paste0(
  col_red("I am red---"),
  col_green("and I am green-"),
  style_underline("I underlined")
)

cat(str, "\n")

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

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

ansi_strtrim *Truncate an ANSI string*

Description

This function is similar to base::strtrim(), but works correcntly with ANSI styled strings. It also adds ... (or the corresponding Unicode character if Unicode characters are allowed) to the end of truncated strings.

Usage

```
ansi_strtrim(x, width = console_width(), ellipsis = symbol$ellipsis)
```

Arguments

x	Character vector of ANSI strings.
width	The width to truncate to.
ellipsis	The string to append to truncated strings. Supply an empty string if you don't want a marker.

See Also

Other ANSI string operations: ansi_align(), ansi_columns(), ansi_nchar(), ansi_strsplit(), ansi_strwrap(), ansi_substring(), ansi_substr(), ansi_toupper(), ansi_trimws()

Examples

```
text <- cli::col_red(cli:::lorem_ipsum())
ansi_strtrim(c(text, "foobar"), 40)
```

ansi_strwrap *Wrap an ANSI styled string to a certain width*

Description

This function is similar to base::strwrap(), but works on ANSI styled strings, and leaves the styling intact.

Usage

```
ansi_strwrap(
  x,
  width = console_width(),
  indent = 0,
  exdent = 0,
  simplify = TRUE
)
```

Arguments

x	ANSI string.
width	Width to wrap to.
indent	Indentation of the first line of each paragraph.
exdent	Indentation of the subsequent lines of each paragraph.
simplify	Whether to return all wrapped strings in a single charcter vector, or wrap each element of x independently and return a list.

Value

If simplify is FALSE, then a list of character vectors, each an ANSI string. Otherwise a single ANSI string vector.

See Also

Other ANSI string operations: ansi_align(), ansi_columns(), ansi_nchar(), ansi_strsplit(), ansi_strtrim(), ansi_substring(), ansi_substr(), ansi_toupper(), ansi_trimws()

Examples

```
text <- cli:::lorem_ipsum()
# Highlight some words, that start with 's'
rexp <- gregexpr("\\b([sS][a-zA-Z]+)\\b", text)
regmatches(text, rexp) <- lapply(regmatches(text, rexp), col_red)
cat(text)

wrp <- ansi_strwrap(text, width = 40)
cat(wrp, sep = "\n")
```

---

ansi_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

```
ansi_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: ansi_align(), ansi_columns(), ansi_nchar(), ansi_strsplit(), ansi_strtrim(), ansi_strwrap(), ansi_substring(), ansi_toupper(), ansi_trimws()

Examples

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

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

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

str2 <- paste(
  "another",
  col_red("multi-", style_underline("style")),
  "text"
)

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

---

ansi_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

```
ansi_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: ansi_align(), ansi_columns(), ansi_nchar(), ansi_strsplit(), ansi_strtrim(), ansi_strwrap(), ansi_substr(), ansi_toupper(), ansi_trimws()

Examples

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

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

substring(ansi_strip(str), 1, 5)
substring(ansi_strip(str), 1, 15)
substring(ansi_strip(str), 3, 7)

str2 <- paste(
  "another",
  col_red("multi-", style_underline("style")),
  "text"
)

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

---

ansi_toupper *ANSI character translation and case folding*

Description

There functions are similar to toupper(), tolower() and chartr(), but they keep the ANSI colors of the string.

Usage

```
ansi_toupper(x)

ansi_tolower(x)

ansi_chartr(old, new, x)
```

Arguments

x	Input string. May have ANSI colors and styles.
old	a character string specifying the characters to be translated. If a character vector of length 2 or more is supplied, the first element is used with a warning.
new	a character string specifying the translations. If a character vector of length 2 or more is supplied, the first element is used with a warning.

Value

Character vector of the same length as x, containing the translated strings. ANSI styles are retained.

See Also

Other ANSI string operations: ansi_align(), ansi_columns(), ansi_nchar(), ansi_strsplit(), ansi_strtrim(), ansi_strwrap(), ansi_substring(), ansi_substr(), ansi_trimws()

Other ANSI string operations: ansi_align(), ansi_columns(), ansi_nchar(), ansi_strsplit(), ansi_strtrim(), ansi_strwrap(), ansi_substring(), ansi_substr(), ansi_trimws()

Other ANSI string operations: ansi_align(), ansi_columns(), ansi_nchar(), ansi_strsplit(), ansi_strtrim(), ansi_strwrap(), ansi_substring(), ansi_substr(), ansi_trimws()

Examples

```
ansi_toupper(col_red("Uppercase"))

ansi_tolower(col_red("LowerCase"))

x <- paste0(col_green("MiXeD"), col_red(" cAsE 123"))
ansi_chartr("iXs", "why", x)
```

ansi_trimws *Remove leading and/or trailing whitespace from an ANSI string*

Description

This function is similar to `base::trimws()` but works on ANSI strings, and keeps color and other styling.

Usage

```
ansi_trimws(x, which = c("both", "left", "right"))
```

Arguments

x	ANSI string vector.
which	Whether to remove leading or trailing whitespace or both.

Value

ANSI string, with the whitespace removed.

See Also

Other ANSI string operations: `ansi_align()`, `ansi_columns()`, `ansi_nchar()`, `ansi_strsplit()`, `ansi_strtrim()`, `ansi_strwrap()`, `ansi_substring()`, `ansi_substr()`, `ansi_toupper()`

Examples

```
trimws(paste0("   ", col_red("I am red"), "   "))
ansi_trimws(paste0("   ", col_red("I am red"), "   "))
trimws(col_red("   I am red   "))
ansi_trimws(col_red("   I am red   "))
```

builtin_theme *The built-in CLI theme*

Description

This theme is always active, and it is at the bottom of the theme stack. See themes.

Usage

```
builtin_theme(dark = getOption("cli_theme_dark", "auto"))
```

Arguments

dark	Whether to use a dark theme. The cli_theme_dark option can be used to request a dark theme explicitly. If this is not set, or set to "auto", then cli tries to detect a dark theme, this works in recent RStudio versions and in iTerm on macOS.

Value

A named list, a CLI theme.

See Also

themes, simple_theme().

---

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 *Compose multiple cli functions*

Description

`cli()` will record all cli_* calls in `expr`, and emit them together in a single message. This is useful if you want to built a larger piece of output from multiple cli_* calls.

Usage

```
cli(expr)
```

Arguments

expr	Expression that contains cli_* calls. Their output is collected and sent as a single message.

Details

Use this function to build a more complex piece of CLI that would not make sense to show in pieces.

Value

Nothing.

Examples

```
cli({
  cli_h1("Title")
  cli_h2("Subtitle")
  cli_ul(c("this", "that", "end"))
})
```

cli-config *cli environment variables and options*

Description

cli environment variables and options

User facing configuration

These are env vars and options that usrs may set, to modify the behavior of cli.

User facing environment variables:

`NO_COLOR`:
Set to a nonempty value to turn off ANSI colors. See `num_ansi_colors()`.

`R_CLI_DYNAMIC`:
Set to `true`, `TRUE` or `True` to assume a dynamic terminal, that supports \r. Set to anything else to assume a non-dynamic terminal. See `is_dynamic_tty()`.

`R_CLI_NUM_COLORS`:
Set to a positive integer to assume a given number of colors. See `num_ansi_colors()`.

User facing options:

`cli.ansi`:
Set to `true`, `TRUE` or `True` to assume a terminal that supports ANSI control sequences. Set to anything else to assume a non-ANSI terminal. See `is_ansi_tty()`.

`cli.default_handler`:
General handler function for all cli conditions. See https://cli.r-lib.org/articles/semantic-cli.html#cli-messages-1

`cli.dynamic`:
Set to `TRUE` to assume a dynamic terminal, that supports \r. Set to anything else to assume a non-dynamic terminal. See `is_dynamic_tty()`.

`cli.hide_cursor`:
Whether the cli status bar should try to hide the cursor on termianals. Set the `FALSE` if the hidden cursor causes issues.

`cli.hyperlink`:
Set to `true`, `TRUE` or `True` to tell cli that the terminal supports ANSI hyperlinks. Set to anything else to assume no hyperlink support. See `style_hyperlink()`.

`cli.num_colors`:
Number of ANSI colors. See `num_ansi_colors()`.

`cli.message_class`:
Character vector of classes to add to cli's conditions.

`cli.progress_bar_style`:
Progress bar style. See `cli_progress_styles()`.

`cli.progress_bar_style_ascii`:
Progress bar style on ASCII consoles. See `cli_progress_styles()`.

`cli.progress_bar_style_unicode`:
Progress bar style on Unicode (UTF-8) consoles; See `cli_progress_styles()`.

`cli.progress_clear`:
Whether to clear terminated progress bar from the screen on dynamic terminals. See `cli_progress_bar()`.

`cli.progress_demo_live`:
Whether `cli_progress_demo()` should show a live demo, or just record the progress bar frames.

`cli.progress_format_download`:
Default format string for `download` progress bars.

`cli.progress_format_download_nototal`:
Default format string for `download` progress bars with unknown totals.

`cli.progress_format_iterator`:
Default format string for `iterator` progress bars.

`cli.progress_format_iterator_nototal`:
Default format string for `iterator` progress bars with unknown total number of progress units.

`cli.progress_format_tasks`:
Default format string for `tasks` progress bars.

`cli.progress_format_tasks_nototal`:
Default format string for `tasks` progress bars with unknown totals.

`cli.progress_handlers`:
Progress handlers to try. See `cli_progress_builtin_handlers()`.

`cli.progress_handlers_force`:
Progress handlers that will always be used, even if another handler was already selected. See `cli_progress_builtin_handlers()`.

`cli.progress_handlers_only`:
Progress handlers to force, ignoring handlers set in `cli.progress_handlers` and `cli.progress_handlers_force`. See `cli_progress_builtin_handlers()`.

`cli.progress_say_args`:
Command line arguments for the `say` progress handlers. See `cli_progress_builtin_handlers()`.

`cli.progress_say_command`:
External command to use in the `say` progress handler. See `cli_progress_builtin_handlers()`.

`cli.progress_say_frequency`:
Minimum delay between `say` calls in the `say` progress handler. `say` ignores very frequent updates, to keep the speech comprehensible. See `cli_progress_builtin_handlers()`.

`cli.progress_show_after`:
Delay before showing a progress bar, in seconds. Progress bars that finish before this delay are not shown at all.

`cli.spinner`:
Default spinner to use, see `get_spinner()`.

`cli.spinner_ascii`:
Default spinner to use on ASCII terminals, see `get_spinner()`.

`cli.spinner_unicode`:
Default spinner to use on Unicode terminals, see `get_spinner()`.

`cli.theme`:
Default cli theme, in addition to the built-in theme. This option in intended for the package developers. See themes and `start_app()`.

`cli_theme_dark`:
Whether cli should assume a dark theme for the builtin theme. See `builtin_theme()`.

`cli.unicode`:
Whether to assume a Unicode terminal. If not set, then it is auto-detected. See `is_utf8_output()`.

`cli.user_theme`:
cli user theme. This option is intended for end users. See themes.

`cli.width`:
Terminal width to assume. If not set, then it is auto-detected. See `console_width()`.

`rlib_interactive`:
Whether to assume an interactive R session. If not set, then it is auto-detected.

`width`:
Terminal width. This is used on some platforms, if `cli.width` is not set.

Internal configuration

These are env vars and options are for cli developers, users should not rely of them as they may change between cli releases.

Internal environment variables:

`ASCIICAST`:
Used to detect an asciicast subprocess in RStudio.

`ANSICON`:
Used to detect ANSICON when detecting the number of ANSI colors.

`CI`:
Used to detect if the code is running on a CI. If yes, we avoid ANSI hyperlinks.

`CLI_DEBUG_BAD_END`:
Whether to warn about `cli_end()` calls when there is no container to close.

`CLI_NO_BUILTIN_THEME`:
Set it to `true` to omit the builtin theme.

`CLI_SPEED_TIME`:
Can be used to speed up cli's timer. It is a factor, e.g. setting it to 2 makes cli's time go twice as fast.

`CLI_TICK_TIME`:
How often the cli timer should alert, in milliseconds.

`CMDER_ROOT`:
Used to detect cmder when detecting the number of ANSI colors.

`COLORTERM`:
Used when detecting ANSI color support.

`ConEmuANSI`:
Used to detect ConEmu when detecting the number of ANSI colors.

`EMACS`:
Used to detect Emacs.

`INSIDE_EMACS`:
Used to detect Emacs.

`NOT_CRAN`:
Set to `true` to run tests / examples / checks, that do not run on CRAN.

_R_CHECK_PACKAGE_NAME_:
Used to detect R CMD check.

`R_BROWSER`:
Used to detect the RStudio build pane.

`R_GUI_APP_VERSION`:
Used to detect R.app on macOS, to decide if the console has ANSI control sequences.

`R_PACKAGE_DIR`:
Used to detect if the code is running under R CMD INSTALL.

`R_PDFVIEWER`:
Used to detect the RStudio build pane.

`R_PROGRESS_NO_EXAMPLES`:
Set to `true` to avoid running examples, outside of R CMD check.

`RSTUDIO`:
Used to detect RStudio, in various functions.

`RSTUDIO_CONSOLE_COLOR`:
Used to detect the number of colors in RStudio. See `num_ansi_colors()`.

`RSTUDIO_CONSOLE_WIDTH`:
Used to auto-detect console width in RStudio.

`RSTUDIO_TERM`:
Used to detect the RStudio build pane.

`TEAMCITY_VERSION`:
Used to detect the TeamCity CI, to turn off ANSI hyperlinks.

`TERM`:
Used to detect if the console has ANSI control sequences, in a terminal.

`TERM_PROGRAM`:
Used to detect iTerm for the dark theme detection and the ANSI hyperlink support detection.

`TERM_PROGRAM_VERSION`:
Used to detect a suitable iTerm version for ANSI hyperlink support.

`TESTTHAT`:
Used to detect running in testthat tests.

`VTE_VERSION`:
Used to detect a suitable VTE version for ANSI hyperlinks.

Internal options:

`cli__pb`:
This option is set to the progress bar that is being updated, when interpolatinf the format string.

`cli.record`:
Internal option to mark the state that cli is recording messages.

`crayon.colors`:
Deprecated option for the number of ANSI colors, that is still supported by cli, when the new options are not set. See `num_ansi_colors()`.

`crayon.enabled`:
Deprecated option to turn ANSI colors on/off. This is still supported by cli when the new options are not set. See `num_ansi_colors()`.

`crayon.hyperlink`:
Whether to assume ANSI hyperlink support. See `ansi_has_hyperlink_support()`.

`knitr.in.progress`:
Used to detect knitr when detecting interactive sessions and ANSI color support.

`rstudio.notebook.executing`:
Used to detect knitr when detecting interactive sessions.

`cli_abort` *Signal an error, warning or message with a cli formatted message*

Description

These functions let you create error, warning or diagnostic messages with cli formatting, including inline styling, pluralization and glue substitutions.

Usage

```
cli_abort(message, ..., .envir = parent.frame())

cli_warn(message, ..., .envir = parent.frame())

cli_inform(message, ..., .envir = parent.frame())
```

Arguments

`message`	It is formatted via a call to `cli_bullets()`.
`...`	Passed to `rlang::abort()`, `rlang::warn()` or `rlang::inform()`.
`.envir`	Environment to evaluate the glue expressions in.

Examples

```
## Not run:
n <- "boo"
cli_abort(c(
        "{.var n} must be a numeric vector",
  "x" = "You've supplied a {.cls {class(n)}} vector."
))

len <- 26
idx <- 100
cli_abort(c(
        "Must index an existing element:",
  "i" = "There {?is/are} {len} element{?s}.",
  "x" = "You've tried to subset element {idx}."
))

## End(Not run)
```

`cli_alert` *CLI alerts*

Description

Alerts are typically short status messages.

Usage

```
cli_alert(text, id = NULL, class = NULL, wrap = FALSE, .envir = parent.frame())

cli_alert_success(
  text,
  id = NULL,
  class = NULL,
  wrap = FALSE,
  .envir = parent.frame()
)

cli_alert_danger(
  text,
  id = NULL,
  class = NULL,
  wrap = FALSE,
  .envir = parent.frame()
)

cli_alert_warning(
  text,
  id = NULL,
```

```
  class = NULL,
  wrap = FALSE,
  .envir = parent.frame()
)

cli_alert_info(
  text,
  id = NULL,
  class = NULL,
  wrap = FALSE,
  .envir = parent.frame()
)
```

Arguments

text	Text of the alert.
id	Id of the alert element. Can be used in themes.
class	Class of the alert element. Can be used in themes.
wrap	Whether to auto-wrap the text of the alert.
.envir	Environment to evaluate the glue expressions in.

Examples

```
cli_alert("Cannot lock package library.")
cli_alert_success("Package {.pkg cli} installed successfully.")
cli_alert_danger("Could not download {.pkg cli}.")
cli_alert_warning("Internet seems to be unreacheable.")
cli_alert_info("Downloaded 1.45MiB of data")
```

---

cli_blockquote *CLI block quote*

Description

A section that is quoted from another source. It is typically indented.

Usage

```
cli_blockquote(
  quote,
  citation = NULL,
  id = NULL,
  class = NULL,
  .envir = parent.frame()
)
```

Arguments

quote	Text of the quotation.
citation	Source of the quotation, typically a link or the name of a person.
id	Element id, a string. If NULL, then a new id is generated and returned.
class	Class name, sting. Can be used in themes.
.envir	Environment to evaluate the glue expressions in. It is also used to auto-close the container if .auto_close is TRUE.

Examples

```
cli_blockquote(cli:::lorem_ipsum(), citation = "Nobody, ever")
```

---

cli_bullets *List of items*

Description

It is often useful to print out a list of items, tasks a function or package performs, or a list of notes.

Usage

```
cli_bullets(text, id = NULL, class = NULL, .envir = parent.frame())
```

Arguments

text	Character vector of items. See details below on how names are interpreted.
id	Optional od of the div.memo element, can be used in themes.
class	Optional additional class(es) for the div.memo element.
.envir	Environment to evaluate the glue expressions in.

Details

Items may be formatted differently, e.g. they can have a prefix symbol. Formatting is specified by the names of text, and can be themed. cli creates a div element of class memo for the whole memo. Each item is another div element of class memo-item-<name>, where <name> is the name of the entry in text. Entries in text without a name create a div element of class memo-item-empty, and if the name is a single space character, the class is memo-item-space.

The builtin theme defines the following item types:

- No name: Item without a prefix.
- : Indented item.
- *: Item with a bullet.
- >: Item with an arrow or pointer.
- v: Item with a green "tick" symbol, like cli_alert_success().

- x: Item with a ref cross, like cli_alert_danger().
- !: Item with a yellow exclamation mark, like cli_alert_warning().
- i: Info item, like cli_alert_info().

You can define new item type by simply defining theming for the corresponding memo-item-<name> classes.

Examples

```
cli_bullets(c(
        "noindent",
  " " = "indent",
  "*" = "bullet",
  ">" = "arrow",
  "v" = "success",
  "x" = "danger",
  "!" = "warning",
  "i" = "info"
))
```

---

cli_code *A block of code*

Description

A helper function that creates a div with class code and then calls cli_verbatim() to output code lines. The builtin theme formats these containers specially. In particular, it adds syntax highlighting to valid R code.

Usage

```
cli_code(
  lines = NULL,
  ...,
  language = "R",
  .auto_close = TRUE,
  .envir = environment()
)
```

Arguments

lines	Chracter vector, each line will be a line of code, and newline charactes also create new lines. Note that *no* glue substitution is performed on the code.
...	More character vectors, they are appended to lines.
language	Programming language. This is also added as a class, in addition to code.

`.auto_close`	Passed to `cli_div()` when creating the container of the code. By default the code container is closed after emitting `lines` and `...` via `cli_verbatim()`. You can keep that container open with `.auto_close` and/or `.envir`, and then calling `cli_verbatim()` to add (more) code. Note that the code will be formatted and syntax highlighted separately for each `cli_verbatim()` call.
`.envir`	Passed to `cli_div()` when creating the container of the code.

Value

The id of the container that contains the code.

Examples

```
cli_code(format(cli::cli_blockquote))
```

---

cli_debug_doc *Debug cli internals*

Description

Return the current state of a cli app. It includes the currently open tags, their ids, classes and their computed styles.

Usage

```
cli_debug_doc(app = default_app() %||% start_app())
```

Arguments

`app`	The cli app to debug. Defaults to the current app. if there is no app, then it creates one by calling `start_app()`.

Details

The returned data frame has a print method, and if you want to create a plain data frame from it, index it with an empty bracket: `cli_debug_doc()[]`.

To see all currently active themes, use `app$themes`, e.g. for the default app: `default_app()$themes`.

Value

Data frame with columns: `tag`, `id`, `class` (space separated), theme (id of the theme the element added), `styles` (computed styles for the element).

See Also

`cli_sitrep()`. To debug containers, you can set the `CLI-DEBUG_BAD_END` environment variable to `true`, and then cli will warn when it cannot find the specified container to close (or any contained at all).

Examples

```
## Not run:
cli_debug_doc()

olid <- cli_ol()
cli_li()
cli_debug_doc()
cli_debug_doc()[]

cli_end(olid)
cli_debug_doc()

## End(Not run)
```

`cli_div` *Generic CLI container*

Description

See containers. A `cli_div` container is special, because it may add new themes, that are valid within the container.

Usage

```
cli_div(
  id = NULL,
  class = NULL,
  theme = NULL,
  .auto_close = TRUE,
  .envir = parent.frame()
)
```

Arguments

`id`	Element id, a string. If `NULL`, then a new id is generated and returned.
`class`	Class name, sting. Can be used in themes.
`theme`	A custom theme for the container. See themes.
`.auto_close`	Whether to close the container, when the calling function finishes (or `.envir` is removed, if specified).
`.envir`	Environment to evaluate the glue expressions in. It is also used to auto-close the container if `.auto_close` is `TRUE`.

Value

The id of the new container element, invisibly.

Examples

```
## div with custom theme
d <- cli_div(theme = list(h1 = list(color = "blue",
                                    "font-weight" = "bold")))
cli_h1("Custom title")
cli_end(d)

## Close automatically
div <- function() {
  cli_div(class = "tmp", theme = list(.tmp = list(color = "yellow")))
  cli_text("This is yellow")
}
div()
cli_text("This is not yellow any more")
```

`cli_dl` *Definition list*

Description

A definition list is a container, see containers.

Usage

```
cli_dl(
  items = NULL,
  id = NULL,
  class = NULL,
  .close = TRUE,
  .auto_close = TRUE,
  .envir = parent.frame()
)
```

Arguments

`items`	Named character vector, or `NULL`. If not `NULL`, they are used as list items.
`id`	Id of the list container. Can be used for closing it with `cli_end()` or in themes. If `NULL`, then an id is generated and returned invisibly.
`class`	Class of the list container. Can be used in themes.
`.close`	Whether to close the list container if the `items` were specified. If `FALSE` then new items can be added to the list.
`.auto_close`	Whether to close the container, when the calling function finishes (or `.envir` is removed, if specified).
`.envir`	Environment to evaluate the glue expressions in. It is also used to auto-close the container if `.auto_close` is `TRUE`.

Value

The id of the new container element, invisibly.

Examples

```
## Specifying the items at the beginning
cli_dl(c(foo = "one", bar = "two", baz = "three"))

## Adding items one by one
cli_dl()
cli_li(c(foo = "one"))
cli_li(c(bar = "two"))
cli_li(c(baz = "three"))
cli_end()
```

---

cli_end *Close a CLI container*

Description

Close a CLI container

Usage

```
cli_end(id = NULL)
```

Arguments

id	Id of the container to close. If missing, the current container is closed, if any.

Examples

```
## If id is omitted
cli_par()
cli_text("First paragraph")
cli_end()
cli_par()
cli_text("Second paragraph")
cli_end()
```

`cli_format` *Format a value for printing*

Description

This function can be used directly, or via the {.val ...} inline style. {.val {expr}} calls `cli_format()` automatically on the value of `expr`, before styling and collapsing it.

Usage

```
cli_format(x, style = NULL, ...)

## Default S3 method:
cli_format(x, style = NULL, ...)

## S3 method for class 'character'
cli_format(x, style = NULL, ...)

## S3 method for class 'numeric'
cli_format(x, style = NULL, ...)
```

Arguments

`x`	The object to format.
`style`	List of formatting options, see the individual methods for the style options they support.
`...`	Additional arguments for methods.

Details

It is possible to define new S3 methods for `cli_format` and then these will be used automatically for {.val ...} expressions.

See Also

`cli_vec()`

Examples

```
things <- c(rep("this", 3), "that")
cli_format(things)
cli_text("{.val {things}}")

nums <- 1:5 / 7
cli_format(nums, style = list(digits = 2))
cli_text("{.val {nums}}")
divid <- cli_div(theme = list(.val = list(digits = 3)))
cli_text("{.val {nums}}")
cli_end(divid)
```

`cli_format_method` *Create a format method for an object using cli tools*

Description

This method can be typically used in `format()` S3 methods. Then the `print()` method of the class can be easily defined in terms of such a `format()` method. See examples below.

Usage

```
cli_format_method(expr, theme = getOption("cli.theme"))
```

Arguments

`expr`	Expression that calls cli_* methods, `base::cat()` or `base::print()` to format an object's printout.
`theme`	Theme to use for the formatting.

Value

Character vector, one element for each line of the printout.

Examples

```
# Let's create format and print methods for a new S3 class that
# represents the an installed R package: `r_package`

# An `r_package` will contain the DESCRIPTION metadata of the package
# and also its installation path.
new_r_package <- function(pkg) {
  tryCatch(
    desc <- packageDescription(pkg),
    warning = function(e) stop("Cannot find R package `", pkg, "`")
  )
  file <- dirname(attr(desc, "file"))
  if (basename(file) != pkg) file <- dirname(file)
  structure(
    list(desc = unclass(desc), lib = dirname(file)),
    class = "r_package"
  )
}

format.r_package <- function(x, ...) {
  cli_format_method({
    cli_h1("{.pkg {x$desc$Package}} {cli::symbol$line} {x$desc$Title}")
    cli_text("{x$desc$Description}")
    cli_ul(c(
      "Version: {x$desc$Version}",
```

```
      if (!is.null(x$desc$Maintainer)) "Maintainer: {x$desc$Maintainer}",
      "License: {x$desc$License}"
    ))
    if (!is.na(x$desc$URL)) cli_text("See more at {.url {x$desc$URL}}")
  })
}

# Now the print method is easy:
print.r_package <- function(x, ...) {
  cat(format(x, ...), sep = "\n")
}

# Try it out
new_r_package("cli")

# The formatting of the output depends on the current theme:
opt <- options(cli.theme = simple_theme())
print(new_r_package("cli"))
options(opt)  # <- restore theme
```

`cli_h1` *CLI headings*

Description

CLI headings

Usage

```
cli_h1(text, id = NULL, class = NULL, .envir = parent.frame())

cli_h2(text, id = NULL, class = NULL, .envir = parent.frame())

cli_h3(text, id = NULL, class = NULL, .envir = parent.frame())
```

Arguments

`text`	Text of the heading. It can contain inline markup.
`id`	Id of the heading element, string. It can be used in themes.
`class`	Class of the heading element, string. It can be used in themes.
`.envir`	Environment to evaluate the glue expressions in.

Examples

```
cli_h1("Main title")
cli_h2("Subtitle")
cli_text("And some regular text....")
```

cli_li *CLI list item(s)*

Description

A list item is a container, see containers.

Usage

```
cli_li(
  items = NULL,
  id = NULL,
  class = NULL,
  .auto_close = TRUE,
  .envir = parent.frame()
)
```

Arguments

items	Character vector of items, or NULL.
id	Id of the new container. Can be used for closing it with cli_end() or in themes. If NULL, then an id is generated and returned invisibly.
class	Class of the item container. Can be used in themes.
.auto_close	Whether to close the container, when the calling function finishes (or .envir is removed, if specified).
.envir	Environment to evaluate the glue expressions in. It is also used to auto-close the container if .auto_close is TRUE.

Value

The id of the new container element, invisibly.

Examples

```
## Adding items one by one
cli_ul()
cli_li("one")
cli_li("two")
cli_li("three")
cli_end()

## Complex item, added gradually.
cli_ul()
cli_li()
cli_verbatim("Beginning of the {.emph first} item")
cli_text("Still the first item")
cli_end()
```

```
cli_li("Second item")
cli_end()
```

`cli_list_themes` *List the currently active themes*

Description

If there is no active app, then it calls `start_app()`.

Usage

```
cli_list_themes()
```

Value

A list of data frames with the active themes. Each data frame row is a style that applies to selected CLI tree nodes. Each data frame has columns:

- `selector`: The original CSS-like selector string. See themes.
- `parsed`: The parsed selector, as used by cli for matching to nodes.
- `style`: The original style.
- `cnt`: The id of the container the style is currently applied to, or `NA` if the style is not used.

See Also

themes

`cli_ol` *Ordered CLI list*

Description

An ordered list is a container, see containers.

Usage

```
cli_ol(
  items = NULL,
  id = NULL,
  class = NULL,
  .close = TRUE,
  .auto_close = TRUE,
  .envir = parent.frame()
)
```

Arguments

items	If not NULL, then a character vector. Each element of the vector will be one list item, and the list container will be closed by default (see the .close argument).
id	Id of the list container. Can be used for closing it with cli_end() or in themes. If NULL, then an id is generated and returned invisibly.
class	Class of the list container. Can be used in themes.
.close	Whether to close the list container if the items were specified. If FALSE then new items can be added to the list.
.auto_close	Whether to close the container, when the calling function finishes (or .envir is removed, if specified).
.envir	Environment to evaluate the glue expressions in. It is also used to auto-close the container if .auto_close is TRUE.

Value

The id of the new container element, invisibly.

Examples

```
## Specifying the items at the beginning
cli_ol(c("one", "two", "three"))

## Adding items one by one
cli_ol()
cli_li("one")
cli_li("two")
cli_li("three")
cli_end()

## Nested lists
cli_div(theme = list(ol = list("margin-left" = 2)))
cli_ul()
cli_li("one")
cli_ol(c("foo", "bar", "foobar"))
cli_li("two")
cli_end()
cli_end()
```

---

cli_output_connection *The connection option that cli would use*

Description

Note that this only refers to the current R process. If the output is produced in another process, then it is not relevant.

Usage

```
cli_output_connection()
```

Details

In interactive sessions the standard output is chosen, othrwise the standard error is used. This is to avoid painting output messages red in the R GUIs.

Value

Connection object.

---

cli_par *CLI paragraph*

Description

See containers.

Usage

```
cli_par(id = NULL, class = NULL, .auto_close = TRUE, .envir = parent.frame())
```

Arguments

id	Element id, a string. If NULL, then a new id is generated and returned.
class	Class name, sting. Can be used in themes.
.auto_close	Whether to close the container, when the calling function finishes (or .envir is removed, if specified).
.envir	Environment to evaluate the glue expressions in. It is also used to auto-close the container if .auto_close is TRUE.

Value

The id of the new container element, invisibly.

Examples

```
id <- cli_par()
cli_text("First paragraph")
cli_end(id)
id <- cli_par()
cli_text("Second paragraph")
cli_end(id)
```

`cli_process_start` *Indicate the start and termination of some computation in the status bar*

Description

Typically you call `cli_process_start()` to start the process, and then `cli_process_done()` when it is done. If an error happens before `cli_process_done()` is called, then cli automatically shows the message for unsuccessful termination.

Usage

```
cli_process_start(
  msg,
  msg_done = paste(msg, "... done"),
  msg_failed = paste(msg, "... failed"),
  on_exit = c("auto", "failed", "done"),
  msg_class = "alert-info",
  done_class = "alert-success",
  failed_class = "alert-danger",
  .auto_close = TRUE,
  .envir = parent.frame()
)

cli_process_done(
  id = NULL,
  msg_done = NULL,
  .envir = parent.frame(),
  done_class = "alert-success"
)

cli_process_failed(
  id = NULL,
  msg = NULL,
  msg_failed = NULL,
  .envir = parent.frame(),
  failed_class = "alert-danger"
)
```

Arguments

`msg`	The message to show to indicate the start of the process or computation. It will be collapsed into a single string, and the first line is kept and cut to `console_width()`.
`msg_done`	The message to use for successful termination.
`msg_failed`	The message to use for unsuccessful termination.
`on_exit`	Whether this process should fail or terminate successfully when the calling function (or the environment in `.envir`) exits.

msg_class	The style class to add to the message. Use an empty string to suppress styling.
done_class	The style class to add to the successful termination message. Use an empty string to suppress styling.a
failed_class	The style class to add to the unsuccessful termination message. Use an empty string to suppress styling.a
.auto_close	Whether to clear the status bar when the calling function finishes (or '.envir' is removed from the stack, if specified).
.envir	Environment to evaluate the glue expressions in. It is also used to auto-clear the status bar if .auto_close is TRUE.
id	Id of the status bar container to clear. If id is not the id of the current status bar (because it was overwritten by another status bar container), then the status bar is not cleared. If NULL (the default) then the status bar is always cleared.

Details

If you handle the errors of the process or computation, then you can do the opposite: call cli_process_start() with on_exit = "done", and in the error handler call cli_process_failed(). cli will automatically call cli_process_done() on successful termination, when the calling function finishes.

See examples below.

Value

Id of the status bar container.

See Also

Other status bar: cli_status_clear(), cli_status_update(), cli_status()

Examples

```
## Failure by default
fun <- function() {
  cli_process_start("Calculating")
  if (interactive()) Sys.sleep(1)
  if (runif(1) < 0.5) stop("Failed")
  cli_process_done()
}
tryCatch(fun(), error = function(err) err)

## Success by default
fun2 <- function() {
  cli_process_start("Calculating", on_exit = "done")
  tryCatch({
    if (interactive()) Sys.sleep(1)
    if (runif(1) < 0.5) stop("Failed")
  }, error = function(err) cli_process_failed())
}
fun2()
```

`cli_progress_along` *Add a progress bar to a mapping function or for loop*

Description

Note that this function is currently experimental!

Use `cli_progress_along()` in a mapping function or in a for loop, to add a progress bar. It uses `cli_progress_bar()` internally.

Usage

```
cli_progress_along(
  x,
  name = NULL,
  total = length(x),
  ...,
  .envir = parent.frame()
)
```

Arguments

`x`	Sequence to add the progress bar to.
`name`	Name of the progtess bar, a label, passed to `cli_progress_bar()`.
`total`	Passed to `cli_progress_bar()`.
`...`	Passed to `cli_progress_bar()`.
`.envir`	Passed to `cli_progress_bar()`.

Details

Usage:

```
lapply(cli_progress_along(X), function(i) ...)

for (i in cli_progress_along(seq)) {
  ...
}
```

Note that if you use `break` in the `for` loop, you probably want to terminate the progress bar explicitly when breaking out of the loop, or right after the loop:

```
for (i in cli_progress_along(seq)) {
  ...
  if (cond) cli_progress_done() && break
  ...
}
```

Value

An index vector from 1 to `length(x)` that triggers progress updates as you iterate over it.

See Also

`cli_progress_bar()` and the traditional progress bar API.

`cli_progress_bar` *cli progress bars*

Description

This is the reference manual of the three functions that create, update and terminate progress bars. For a tutorial see the cli progress bars.

`cli_progress_bar()` creates a progress bar.

`cli_progress_update()` updates the state of a progress bar, and potentially the display as well.

`cli_progress_done()` terminates a progress bar.

Usage

```
cli_progress_bar(
  name = NULL,
  status = NULL,
  type = c("iterator", "tasks", "download", "custom"),
  total = NA,
  format = NULL,
  format_done = NULL,
  format_failed = NULL,
  clear = getOption("cli.progress_clear", TRUE),
  current = TRUE,
  auto_terminate = type != "download",
  extra = NULL,
  .auto_close = TRUE,
  .envir = parent.frame()
)

cli_progress_update(
  inc = NULL,
  set = NULL,
  total = NULL,
  status = NULL,
  extra = NULL,
  id = NULL,
  force = FALSE,
  .envir = parent.frame()
)
```

```
cli_progress_done(id = NULL, .envir = parent.frame(), result = "done")
```

Arguments

name	This is typically used as a label, and should be short, at most 20 characters.
status	New status string of the progress bar, if not NULL.
type	Type of the progress bar. It is used to select a default display if format is not specified. Currently supported types: • iterator: e.g. a for loop or a mapping function, • tasks: a (typically small) number of tasks, • download: download of one file, • custom: custom type, format must not be NULL for this type.
total	Total number of progress units, or NA if it is unknown. cli_progress_update() can update the total number of units. This is handy if you don't know the size of a download at the beginning, and also in some other casees. If format (plus format_done and format_done) will be updated if you change total from NA to a number, if you specify NULL for format. I.e. default format strings will be updated, custom ones won't be.
format	Format string. It has to be specified for custom progress bars, otherwise it is optional, and a default display is selected based on the progress bat type and whether the number of total units is known. Format strings may contain glue substitution, the support pluralization and cli styling.
format_done	Format string for successful termination. By default the same as format.
format_failed	Format string for unsuccessful termination. By default the same as format.
clear	Whether to remove the progress bar from the screen after it has temrinated.
current	Whether to use this progress bar as the current progress bar of the calling function. See more at 'The current progress bar' below.
auto_terminate	Whether to terminate the progress bar if the number of current units reaches the number of total units.
extra	Extra data to add to the progress bar. This can be used in custom format strings for example. It should be a named list. cli_progress_update() can update the extra data.
.auto_close	Whether to terminate the progress bar when the calling function (or the one with execution environment in .envir exits. (Auto termination does not work for progress bars created from the global environment, e.g. from a script.)
.envir	The environment to use for auto-termination and for glue substitution. It is also used to find and set the current progress bar.
inc	Increment in progress units. This is ignored if set is not NULL.
set	Set the current number of progress units to this value. Ignored if NULL.
id	Progress bar to update or terminate. If NULL, then the current progress bar of the calling function (or .envir if specified) is updated or terminated.
force	Whether to force a display update, even if no update is due.

`result` String to select successful or unsuccessful termination. It is only used if the progress bar is not cleared from the screen. It can be one of `"done"`, `"failed"`, `"clear"`, and `"auto"`.

Details

`cli_progress_update()` updates the state of the progress bar and potentially outputs the new progress bar to the display as well.

Value

`cli_progress_bar()` returns the id of the new progress bar. The id is a string constant.

The current progress bar:

If `current = TRUE` (the default), then the new progress bar will be the *current* progress bar of the calling frame. The previous current progress bar of the same frame, if there is any, is terminated.

`cli_progress_update()` returns the id of the progress bar, invisibly.

`cli_progress_done()` returns `TRUE`, invisibly, always.

See Also

`cli_progress_message()` and `cli_progress_step()` for simpler progress messages.

Examples

```
clean <- function() {
  cli_progress_bar("Cleaning data", total = 100)
  for (i in 1:100) {
    Sys.sleep(5/100)
    cli_progress_update()
  }
}
clean()
```

---

cli_progress_builtin_handlers

cli progress handlers

Description

The progress handler(s) to use can be selected with global options.

Usage

```
cli_progress_builtin_handlers()
```

Details

There are three options that specify which handlers will be selected, but most of the time you only need to use one of them. You can set these options to a character vector, the names of the built-in cli handlers you want to use:

- If `cli.progress_handlers_only` is set, then these handlers are used, without considering others and without checking if they are able to handle a progress bar. This option is mainly intended for testing purposes.
- The handlers named in `cli.progress_handlers` are checked if they are able to handle the progress bar, and from the ones that are, the first one is selected. This is usually the option that the end use would want to set.
- The handlers named in `cli.progress_handlers_force` are always appended to the ones selected via `cli.progress_handlers`. This option is useful to add an additional handler, e.g. a logger that writes to a file.

Value

`cli_progress_builtin_handlers()` returns the names of the currently supported progress handlers.

The built-in progress handlers

`cli`:
Use cli's internal status bar, the last line of the screen, to show the progress bar. This handler is always able to handle all progress bars.

`logger`:
Log progress updates to the screeen, with one line for each update, with time stamps. This handler is always able to handle all progress bars.

`progressr`:
Use the progressr package to create progress bars. This handler is always able to handle all progress bars. (The progressr package needs to be installed.)

`rstudio`:
Use RStudio's job panel to show the progress bars. This handler is available at the RStudio console, in recent versions of RStudio.

`say`:
Use the macOS command line `say` command to announce progress events in speech. Set the `cli.progress_say_frequency` option to set the minimum delay between `say` invocations, the default is three seconds. This handler is available on macOS, if the `say` command is on the path.

The external command and its arguments can be configured with options:

- `cli_progress_say_args`: command line arguments, e.g. you can use this to select a voice on macOS,
- `cli_progress_say_command`: external command to run,
- `cli_progress_say_frequency`: wait at least this many seconds between calling the external command.

`shiny`:
Use shiny's progress bars. This handler is available if a shiny app is running.

`cli_progress_demo` *cli progress bar demo*

Description

Useful for experimenting with format strings and for documentation. It creates a progress bar, iterates it until it terminates and saves the progress updates.

Usage

```
cli_progress_demo(
  name = NULL,
  status = NULL,
  type = c("iterator", "tasks", "download", "custom"),
  total = NA,
  .envir = parent.frame(),
  ...,
  at = if (is_interactive()) NULL else 50,
  show_after = 0,
  live = NULL,
  delay = 0,
  start = as.difftime(5, units = "secs")
)
```

Arguments

`name`	Passed to `cli_progress_bar()`.
`status`	Passed to `cli_progress_bar()`.
`type`	Passed to `cli_progress_bar()`.
`total`	Passed to `cli_progress_bar()`.
`.envir`	Passed to `cli_progress_bar()`.
`...`	Passed to `cli_progress_bar()`.
`at`	The number of progress units to show and capture the progress bar at. If `NULL`, then a sequence of states is generated to show the progress from beginning to end.
`show_after`	Delay to show the progress bar. Overrides the `cli.progress_show_after` option.
`live`	Whether to show the progress bat on the screen, or just return the recorded updates. Defaults to the value of the `cli.progress_demo_live` options. If unset, then it is `TRUE` in interactive sessions.
`delay`	Delay between progress bar updates.
`start`	Time to subtract from the start time, to simulate a progress bar that takes longer to run.

Value

List with class `cli_progress_demo`, which has a print and a format method for prretty printing. The `lines` entry contains the output lines, each correcponding to one update.

`cli_progress_message` *Simplified cli progress messages*

Description

This is a simplified progress bar, a single (dynamic) message, without progress units.

Usage

```
cli_progress_message(
  msg,
  current = TRUE,
  .auto_close = TRUE,
  .envir = parent.frame(),
  ...
)
```

Arguments

`msg`	Message to show. It may contain glue substitution and cli styling. It can be updated via `cli_progress_update()`, as usual.
`current`	Passed to `cli_progress_bar()`.
`.auto_close`	Passed to `cli_progress_bar()`.
`.envir`	Passed to `cli_progress_bar()`.
`...`	Passed to `cli_progress_bar()`.

Details

`cli_progress_message()` always shows the message, even if no update is due. When the progress message is terminated, it is removed from the screen by default.

Value

The id of the new progress bar.

See Also

`cli_progress_bar()` for the complete progress bar API. `cli_progress_step()` for a similar display that is styled by default.

cli_progress_num *Progress bar utility functions.*

Description

Progress bar utility functions.

Usage

```
cli_progress_num()

cli_progress_cleanup()
```

Details

`cli_progress_num()` returns the number of currently active progress bars. (These do not currently include the progress bars created in C/C++ code.)

`cli_progress_cleanup()` terminates all active progress bars. (It currently ignores progress bars created in the C/C++ code.)

Value

`cli_progress_num()` returns an integer scalar.

'cli_progress_cleanup() does not return anything.

cli_progress_output *Add text output to a progress bar*

Description

The text is calculated via `cli_text()`, so all cli features can be used here, including progress variables.

Usage

```
cli_progress_output(text, id = NULL, .envir = parent.frame())
```

Arguments

`text`	Text to output. It is formatted via `cli_text()`.
`id`	Progress bar id. The default is the current progress bar.
`.envir`	Environment to use for glue interpolation of `text`.

Details

The text is passed to the progress handler(s), that may or may not be able to print it.

Value

TRUE, always.

---

cli_progress_step *Simplified cli progress messages, with styling*

Description

This is a simplified progress bar, a single (dynamic) message, without progress units.

Usage

```
cli_progress_step(
  msg,
  msg_done = msg,
  msg_failed = msg,
  spinner = FALSE,
  class = if (!spinner) ".alert-info",
  current = TRUE,
  .auto_close = TRUE,
  .envir = parent.frame(),
  ...
)
```

Arguments

msg	Message to show. It may contain glue substitution and cli styling. It can be updated via cli_progress_update(), as usual. It is style as a cli info alert (see cli_alert_info()).
msg_done	Message to show on successful termination. By default this it is the same as msg and it is styled as a cli success alert (see cli_alert_success()).
msg_failed	Message to show on unsuccessful termination. By default it is the same as msg and it is styled as a cli danger alert (see cli_alert_danger()).
spinner	Whether to show a spinner at the beginning of the line. To make the spinner spin, you'll need to call cli_progress_update() regularly.
class	cli class to add to the message. By default there is no class for steps with a spinner.
current	Passed to cli_progress_bar().
.auto_close	Passed to cli_progress_bar().
.envir	Passed to cli_progress_bar().
...	Passed to cli_progress_bar().

Details

`cli_progress_step()` always shows the progress message, even if no update is due.

---

cli_progress_styles *List of built-in cli progress styles*

Description

The following options are used to select a style:

- `cli_progress_bar_style`
- `cli_progress_bar_style_ascii`
- `cli_progress_bar_style_unicode`

Usage

```
cli_progress_styles()
```

Details

On Unicode terminals (if `is_utf8_output()` is `TRUE`), the `cli_progress_bar_style_unicode` and `cli_progress_bar_style` options are used.

On ASCII terminals (if `is_utf8_output()` is `FALSE`), the `cli_pgoress_bar_style_ascii` and `cli_progress_bar_style` options are are used.

Value

A named list with sublists containing elements `complete`, `incomplete` and potentially `current`.

Examples

```
cli_progress_styles()
```

---

cli_rule *CLI horizontal rule*

Description

It can be used to separate parts of the output. The line style of the rule can be changed via the the `line-type` property. Possible values are:

Usage

```
cli_rule(
  left = "",
  center = "",
  right = "",
  id = NULL,
  .envir = parent.frame()
)
```

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.
id	Element id, a string. If NULL, then a new id is generated and returned.
.envir	Environment to evaluate the glue expressions in.

Details

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

Colors and background colors can similarly changed via a theme, see examples below.

Examples

```
cli_rule()
cli_text(packageDescription("cli")$Description)
cli_rule()

# Theming
d <- cli_div(theme = list(rule = list(
  color = "blue",
  "background-color" = "darkgrey",
  "line-type" = "double")))
cli_rule("Left", right = "Right")
cli_end(d)

# Interpolation
cli_rule(left = "One plus one is {1+1}")
cli_rule(left = "Package {.pkg mypackage}")
```

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 `num_ansi_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()
```

cli_status *Update the status bar*

Description

The status bar is the last line of the terminal. cli apps can use this to show status information, progress bars, etc. The status bar is kept intact by all semantic cli output.

Usage

```
cli_status(
  msg,
  msg_done = paste(msg, "... done"),
  msg_failed = paste(msg, "... failed"),
  .keep = FALSE,
  .auto_close = TRUE,
  .envir = parent.frame(),
  .auto_result = c("clear", "done", "failed", "auto")
)
```

Arguments

msg	The text to show, a character vector. It will be collapsed into a single string, and the first line is kept and cut to console_width(). The message is often associated with the start of a calculation.
msg_done	The message to use when the message is cleared, when the calculation finishes successfully. If .auto_close is TRUE and .auto_result is "done", then this is printed automatically when the calling function (or .envir) finishes.
msg_failed	The message to use when the message is cleared, when the calculation finishes unsuccessfully. If .auto_close is TRUE and .auto_result is "failed", then this is printed automatically when the calling function (or .envir) finishes.
.keep	What to do when this status bar is cleared. If TRUE then the content of this status bar is kept, as regular cli output (the screen is scrolled up if needed). If FALSE, then this status bar is deleted.
.auto_close	Whether to clear the status bar when the calling function finishes (or '.envir' is removed from the stack, if specified).
.envir	Environment to evaluate the glue expressions in. It is also used to auto-clear the status bar if .auto_close is TRUE.
.auto_result	What to do when auto-closing the status bar.

Details

Use cli_status_clear() to clear the status bar.

Often status messages are associated with processes. E.g. the app starts downloading a large file, so it sets the status bar accordingly. Once the download is done (or has failed), the app typically updates the status bar again. cli automates much of this, via the msg_done, msg_failed, and .auto_result arguments. See examples below.

Value

The id of the new status bar container element, invisibly.

See Also

cli_process_start() for a higher level interface to the status bar, that adds automatic styling.

Other status bar: cli_process_start(), cli_status_clear(), cli_status_update()

---

cli_status_clear *Clear the status bar*

Description

Clear the status bar

Usage

```
cli_status_clear(
  id = NULL,
  result = c("clear", "done", "failed"),
  msg_done = NULL,
  msg_failed = NULL,
  .envir = parent.frame()
)
```

Arguments

id	Id of the status bar container to clear. If `id` is not the id of the current status bar (because it was overwritten by another status bar container), then the status bar is not cleared. If `NULL` (the default) then the status bar is always cleared.
result	Whether to show a message for success or failure or just clear the status bar.
msg_done	If not `NULL`, then the message to use for successful process termination. This overrides the message given when the status bar was created.
msg_failed	If not `NULL`, then the message to use for failed process termination. This overrides the message give when the status bar was created.
.envir	Environment to evaluate the glue expressions in. It is also used to auto-clear the status bar if `.auto_close` is `TRUE`.

See Also

Other status bar: `cli_process_start()`, `cli_status_update()`, `cli_status()`

---

cli_status_update *Update the status bar*

Description

Update the status bar

Usage

```
cli_status_update(
  id = NULL,
  msg = NULL,
  msg_done = NULL,
  msg_failed = NULL,
  .envir = parent.frame()
)
```

Arguments

id	Id of the status bar to update. Defaults to the current status bar container.
msg	Text to update the status bar with. NULL if you don't want to change it.
msg_done	Updated "done" message. NULL if you don't want to change it.
msg_failed	Updated "failed" message. NULL if you don't want to change it.
.envir	Environment to evaluate the glue expressions in.

Value

Id of the status bar container.

See Also

Other status bar: cli_process_start(), cli_status_clear(), cli_status()

---

cli_text *CLI text*

Description

It is wrapped to the screen width automatically. It may contain inline markup. (See inline-markup.)

Usage

```
cli_text(..., .envir = parent.frame())
```

Arguments

...	The text to show, in character vectors. They will be concatenated into a single string. Newlines are *not* preserved.
.envir	Environment to evaluate the glue expressions in.

Examples

```
cli_text("Hello world!")
cli_text(packageDescription("cli")$Description)

## Arguments are concatenated
cli_text("this", "that")

## Command substitution
greeting <- "Hello"
subject <- "world"
cli_text("{greeting} {subject}!")

## Inline theming
cli_text("The {.fn cli_text} function in the {.pkg cli} package")
```

```
## Use within container elements
ul <- cli_ul()
cli_li()
cli_text("{.emph First} item")
cli_li()
cli_text("{.emph Second} item")
cli_end(ul)
```

`cli_ul` *Unordered CLI list*

Description

An unordered list is a container, see containers.

Usage

```
cli_ul(
  items = NULL,
  id = NULL,
  class = NULL,
  .close = TRUE,
  .auto_close = TRUE,
  .envir = parent.frame()
)
```

Arguments

`items`	If not `NULL`, then a character vector. Each element of the vector will be one list item, and the list container will be closed by default (see the `.close` argument).
`id`	Id of the list container. Can be used for closing it with `cli_end()` or in themes. If `NULL`, then an id is generated and returned invisibly.
`class`	Class of the list container. Can be used in themes.
`.close`	Whether to close the list container if the `items` were specified. If `FALSE` then new items can be added to the list.
`.auto_close`	Whether to close the container, when the calling function finishes (or `.envir` is removed, if specified).
`.envir`	Environment to evaluate the glue expressions in. It is also used to auto-close the container if `.auto_close` is `TRUE`.

Value

The id of the new container element, invisibly.

Examples

```
## Specifying the items at the beginning
cli_ul(c("one", "two", "three"))

## Adding items one by one
cli_ul()
cli_li("one")
cli_li("two")
cli_li("three")
cli_end()

## Complex item, added gradually.
cli_ul()
cli_li()
cli_verbatim("Beginning of the {.emph first} item")
cli_text("Still the first item")
cli_end()
cli_li("Second item")
cli_end()
```

---

cli_vec *Add custom cli style to a vector*

Description

Add custom cli style to a vector

Usage

```
cli_vec(x, style = list())
```

Arguments

x	Vector that will be collapsed by cli.
style	Style to apply to the vector. It is used as a theme on a span element that is created for the vector. You can set vec_sep and vec_last to modify the sep and last arguments of glue::glue_collapse(). See an example below.

Details

You can use this function to change the default parameters of glue::glue_collapse(), see an example below.

The style is added as an attribute, so operations that remove attributes will remove the style as well.

See Also

cli_format()

Examples

```
v <- cli_vec(
  c("foo", "bar", "foobar"),
  style = list(vec_sep = " & ", vec_last = " & ")
)
cli_text("My list: {v}.")

# custom truncation
x <- cli_vec(names(mtcars), list(vec_trunc = 3))
cli_text("Column names: {x}.")
```

---

cli_verbatim *CLI verbatim text*

Description

It is not wrapped, but printed as is.

Usage

```
cli_verbatim(..., .envir = parent.frame())
```

Arguments

...	The text to show, in character vectors. Each element is printed on a new line.
.envir	Environment to evaluate the glue expressions in.

Examples

```
cli_verbatim("This has\nthree", "lines")
```

---

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(), num_ansi_colors()

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 `cli.width` option, if set. Otherwise it tries to determine the size of the terminal or console window.

Usage

```
console_width()
```

Details

These are the exact rules:

- If the `cli.width` option is set to a positive integer, it is used.
- If the `cli.width` option is set, but it is not a positive integer, and error is thrown.

Then we try to determine the size of the terminal or console window:

- If we are not in RStudio, or we are in an RStudio temrinal, then we try to use the `tty_size()` function to query the terminal size. This might fail if R is not running in a terminal, but failures are ignored.
- If we are in the RStudio build pane, then the `RSTUDIO_CONSOLE_WIDTH` environment variable is used. If the build pane is resized, then this environment variable is not accurate any more, and the output might get garbled.
- We are *not* using the `RSTUDIO_CONSOLE_WIDTH` environment variable if we are in the RStudio console.

If we cannot determine the size of the terminal or console window, then we use the `width` option. If the `width` option is not set, then we return 80L.

Value

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

Examples

```
console_width()
```

`containers` *CLI containers*

Description

Container elements may contain other elements. Currently the following commands create container elements: `cli_div()`, `cli_par()`, the list elements: `cli_ul()`, `cli_ol()`, `cli_dl()`, and list items are containers as well: `cli_li()`.

Details

Container elements need to be closed with `cli_end()`. For convenience, they have an `.auto_close` argument, which instructs the container element to be closed automatically when the function that created it terminates (either regularly, or with an error).

Examples

```
## div with custom theme
d <- cli_div(theme = list(h1 = list(color = "blue",
                                    "font-weight" = "bold")))
cli_h1("Custom title")
cli_end(d)

## Close automatically
div <- function() {
  cli_div(class = "tmp", theme = list(.tmp = list(color = "yellow")))
  cli_text("This is yellow")
```

```
}
div()
cli_text("This is not yellow any more")
```

`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)
```

`faq` *Frequently Asked Questions*

Description

Frequently Asked Questions

Details

My platform supports ANSI colors, why does cli not use them?:

It is probably a mistake in cli's ANSI support detection algorithm. Please open an issue at https://github.com/r-lib/cli/issues and don't forget to tell us the details of your platform and terminal or GUI.

How do I turn off ANSI colors and styles?:

Set the NO_COLOR environment variable to a non-empty value. You can do this in your .Renviron file (use usethis::edit_r_environ()).

If you want to do this for testthat tests, then consider using the 3rd edition on testthat, which does turn off ANSI styling automatically inside test_that().

cli does not show the output before file.choose()**:**

Try calling flush.console() to flush the console, before file.choose(). If flushing does not work and you are in RStudio, then it is probably this RStudio bug: https://github.com/rstudio/rstudio/issues/8040 See more details at https://github.com/r-lib/cli/issues/151

---

format_error *Format an error, warning or diagnostic message*

Description

You can then throw this message with stop() or rlang::abort().

Usage

```
format_error(message, .envir = parent.frame())

format_warning(message, .envir = parent.frame())

format_message(message, .envir = parent.frame())
```

Arguments

message	It is formatted via a call to cli_bullets().
.envir	Environment to evaluate the glue expressions in.

Details

The messages can use inline styling, pluralization and glue substitutions.

Examples

```
## Not run:
n <- "boo"
stop(format_error(c(
        "{.var n} must be a numeric vector",
  "x" = "You've supplied a {.cls {class(n)}} vector."
)))

len <- 26
idx <- 100
stop(format_error(c(
```

```
      "Must index an existing element:",
  "i" = "There {?is/are} {len} element{?s}.",
  "x" = "You've tried to subset element {idx}."
)))

## End(Not run)
```

`format_inline` *Format and returns a line of text*

Description

You can use this function to format a line of cli text, without emitting it to the screen. It uses `cli_text()` internally.

Usage

```
format_inline(..., .envir = parent.frame())
```

Arguments

`...`	Passed to `cli_text()`.
`.envir`	Environment to evaluate the expressions in.

Value

Character scalar, the formatted string.

Examples

```
format_inline("This is a message for {.emph later}.")
```

`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. If NULL, then the default is used, which can be customized via the cli.spinner_unicode, cli.spinner_ascii and cli.spinner options. (The latter applies to both Unicode and ASCII displays. These options can be set to the name of a builting spinner, or to a list that has an entry called frames, a character vector of frames.

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")
```

---

inline-markup *CLI inline markup*

Description

CLI inline markup

Command substitution

All text emitted by cli supports glue interpolation. Expressions enclosed by braces will be evaluated as R code. See glue::glue() for details.

In addition to regular glue interpolation, cli can also add classes to parts of the text, and these classes can be used in themes. For example

```
cli_text("This is {.emph important}.")
```

adds a class to the "important" word, class "emph". Note that in this case the string within the braces is usually not a valid R expression. If you want to mix classes with interpolation, add another pair of braces:

```
adjective <- "great"
cli_text("This is {.emph {adjective}}.")
```

An inline class will always create a span element internally. So in themes, you can use the span.emph CSS selector to change how inline text is emphasized:

```
cli_div(theme = list(span.emph = list(color = "red")))
adjective <- "nice and red"
cli_text("This is {.emph {adjective}}.")
```

Classes

The default theme defines the following inline classes:

- `arg` for a function argument.
- `cls` for an S3, S4, R6 or other class name.
- `code` for a piece of code.
- `dt` is used for the terms in a definition list (`cli_dl()`).
- `dd` is used for the descriptions in a definition list (`cli_dl()`).
- `email` for an email address.
- `emph` for emphasized text.
- `envvar` for the name of an environment variable.
- `field` for a generic field, e.g. in a named list.
- `file` for a file name.
- `fun` for a function name.
- `key` for a keyboard key.
- `path` for a path (essentially the same as `file`).
- `pkg` for a package name.
- `strong` for strong importance.
- `url` for a URL.
- `var` for a variable name.
- `val` for a generic "value".

See examples below.

You can simply add new classes by defining them in the theme, and then using them, see the example below.

Highlighting weird-looking values:

Often it is useful to highlight a weird file or path name, e.g. one that starts or ends with space characters. The built-in theme does this for `.file`, `.path` and `.email` by default. You can highlight any string inline by adding the `.q` class to it.

The current highlighting algorithm

- adds single quotes to the string if it does not start or end with an alphanumeric character, underscore, dot or forward slash.
- Highlights the background colors of leading and trailing spaces on terminals that support ANSI colors.

Collapsing inline vectors

When cli performs inline text formatting, it automatically collapses glue substitutions, after formatting. This is handy to create lists of files, packages, etc.

By default cli truncates long vectors. The truncation limit is by default one hundred elements, but you can change it with the `vec_trunc` style.

See examples below.

Formatting values

The `val` inline class formats values. By default (c.f. the built-in theme), it calls the `cli_format()` generic function, with the current style as the argument. See `cli_format()` for examples.

Escaping { and }

It might happen that you want to pass a string to cli_* functions, and you do *not* want command substitution in that string, because it might contain { and } characters. The simplest solution for this is to refer to the string from a template:

```
msg <- "Error in if (ncol(dat$y)) {: argument is of length zero"
cli_alert_warning("{msg}")
```

If you want to explicitly escape { and } characters, just double them:

```
cli_alert_warning("A warning with {{ braces }}")
```

See also examples below.

Pluralization

All cli commands that emit text support pluralization. Some examples:

```
cli_alert_info("Found {ndirs} diretor{?y/ies} and {nfiles} file{?s}.")
cli_text("Will install {length(pkgs)} package{?s}: {.pkg {pkgs}}")
```

See pluralization for details.

Wrapping

Most cli containers wrap the text to width the container's width, while observing margins requested by the theme.

To avoid a line break, you can use the UTF_8 non-breaking space character: \u00a0. cli will not break a line here.

To force a line break, insert a form feed character: \f or \u000c. cli will insert a line break there.

Examples

```
## Some inline markup examples
cli_ul()
cli_li("{.emph Emphasized} text")
cli_li("{.strong Strong} importance")
cli_li("A piece of code: {.code sum(a) / length(a)}")
cli_li("A package name: {.pkg cli}")
cli_li("A function name: {.fn cli_text}")
cli_li("A keyboard key: press {.kbd ENTER}")
cli_li("A file name: {.file /usr/bin/env}")
cli_li("An email address: {.email bugs.bunny@acme.com}")
cli_li("A URL: {.url https://acme.com}")
cli_li("An environment variable: {.envvar R_LIBS}")
```

```
cli_end()

## Adding a new class
cli_div(theme = list(
  span.myclass = list(color = "lightgrey"),
  "span.myclass" = list(before = "["),
  "span.myclass" = list(after = "]")))
cli_text("This is {.myclass in brackets}.")
cli_end()

## Collapsing
pkgs <- c("pkg1", "pkg2", "pkg3")
cli_text("Packages: {pkgs}.")
cli_text("Packages: {.pkg {pkgs}}")

## Custom truncation, style set via cli_vec
nms <- cli_vec(names(mtcars), list(vec_trunc = 5))
cli_text("Column names: {nms}.")

## Classes are collapsed differently by default
x <- Sys.time()
cli_text("Hey {.var x} has class {.cls {class(x)}}")

## Escaping
msg <- "Error in if (ncol(dat$y)) {: argument is of length zero"
cli_alert_warning("{msg}")

cli_alert_warning("A warning with {{ braces }}")
```

`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 = "auto")
```

Arguments

stream	The stream to inspect or manipulate, 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().

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 = "auto")
```

Arguments

stream	The stream to inspect or manipulate, 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

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 the 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,
  header = "",
  footer = "",
  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.
`header`	Text to show on top border of the box. If too long, it will be cut.
`footer`	Text to show on the bottom border of the box. If too long, it will be cut.
`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 console_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.

Usage

```
make_ansi_style(..., bg = FALSE, grey = FALSE, colors = num_ansi_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 `...` style argument can be any of the following:

- A cli ANSI style function of class `ansi_style`. This is returned as is, without looking at the other arguments.
- 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_ansi_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.

Value

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

See Also

Other ANSI styling: `ansi-styles`, `combine_ansi_styles()`, `num_ansi_colors()`

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 = "auto",
  template = "{spin}",
  static = c("dots", "print", "print_line", "silent")
)
```

Arguments

`which`	The name of the chosen spinner. If `NULL`, then the default is used, which can be customized via the `cli.spinner_unicode`, `cli.spinner_ascii` and `cli.spinner` options. (The latter applies to both Unicode and ASCII displays. These options can be set to the name of a builting spinner, or to a list that has an entry called `frames`, a character vector of frames.
`stream`	The stream to use for the spinner. Typically this is standard error, or maybe the standard output stream. 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()`.
`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$spin(); Sys.sleep(0.05) })
  sp3$finish()
}
ansi_with_hidden_cursor(fun_with_spinner3())
```

See Also

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

---

no *Pluralization helper functions*

Description

Pluralization helper functions

Usage

```
no(expr)

qty(expr)
```

Arguments

expr	For `no()` it is an expression that is printed as "no" in cli expressions, it is interpreted as a zero quantity. For `qty()` an expression that sets the pluralization quantity without printing anything. See examples below.

See Also

Other pluralization: `pluralization`, `pluralize()`

---

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")
```

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 env var 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 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 stderr() and there is an active sink for it, then 1L is returned.
9. If R is running inside RGui on Windows, or R.app on macOS, then we return 1L.
10. If R is running inside RStudio, with color support, then the appropriate number of colors is returned, usually 256L.
11. If R is running on Windows, inside an Emacs version that is recent enough to support ANSI colors, then 8L is returned. (On Windows, Emacs has isatty(stdout()) == FALSE, so we need to check for this here before dealing with terminals.)
12. If stream is not a terminal, then 1L is returned.
13. If R is running on Unix, inside an Emacs version that is recent enough to support ANSI colors, then 8L is returned.
14. If stream is not the standard output or error, then 1L is returned.
15. If we are on Windows, under ComEmu or cmder, or ANSICON is loaded, then 8L is returned.
16. Otherwise if we are on Windows, return 1L.
17. 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, except if the TERM environment variable is xterm and tput returned 8L, we return 256L, because xterm compatible terminals tend to support 256 colors.
18. If tput colors fails, we try to guess. If COLORTERM is set to any value, we return 8L.
19. If TERM is set to dumb, we return 1L.
20. If TERM starts with screen, xterm, or vt100, we return 8L.
21. If TERM contains color, ansi, cygwin or linux, we return 8L.
22. Otherwise we return 1L.

Value

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

See Also

Other ANSI styling: ansi-styles, combine_ansi_styles(), make_ansi_style()

Examples

```
num_ansi_colors()
```

`pluralization` *CLI pluralization*

Description

CLI pluralization

Introduction

cli has tools to create messages that are printed correctly in singular and plural forms. This usually requires minimal extra work, and increases the quality of the messages greatly. In this document we first show some pluralization examples that you can use as guidelines. Hopefully these are intuitive enough, so that they can be used without knowing the exact cli pluralization rules.

If you need pluralization without the semantic cli functions, see the `pluralize()` function.

Examples

Pluralization markup:

In the simplest case the message contains a single `{}` glue substitution, which specifies the quantity that is used to select between the singular and plural forms. Pluralization uses markup that is similar to glue, but uses the {? and } delimiters:

```
library(cli)
nfile <- 0; cli_text("Found {nfile} file{?s}.")
```

```
#> Found 0 files.
```

```
nfile <- 1; cli_text("Found {nfile} file{?s}.")
```

```
#> Found 1 file.
```

```
nfile <- 2; cli_text("Found {nfile} file{?s}.")
```

```
#> Found 2 files.
```

Here the value of `nfile` is used to decide whether the singular or plural form of `file` is used. This is the most common case for English messages.

Irregular plurals:

If the plural form is more difficult than a simple `s` suffix, then the singular and plural forms can be given, separated with a forward slash:

```
ndir <- 1; cli_text("Found {ndir} director{?y/ies}.")
```

```
#> Found 1 directory.
```

```
ndir <- 5; cli_text("Found {ndir} director{?y/ies}.")
```

```
#> Found 5 directories.
```

Use "no" instead of zero:

For readability, it is better to use the `no()` helper function to include a count in a message. `no()` prints the word "no" if the count is zero, and prints the numeric count otherwise:

```
nfile <- 0; cli_text("Found {no(nfile)} file{?s}.")
```

```
#> Found no files.
```

```
nfile <- 1; cli_text("Found {no(nfile)} file{?s}.")
```

```
#> Found 1 file.
```

```
nfile <- 2; cli_text("Found {no(nfile)} file{?s}.")
```

```
#> Found 2 files.
```

Use the length of character vectors:

With the auto-collapsing feature of cli it is easy to include a list of objects in a message. When cli interprets a character vector as a pluralization quantity, it takes the length of the vector:

```
pkgs <- "pkg1"
cli_text("Will remove the {.pkg {pkgs}} package{?s}.")
```

```
#> Will remove the pkg1 package.
```

```
pkgs <- c("pkg1", "pkg2", "pkg3")
cli_text("Will remove the {.pkg {pkgs}} package{?s}.")
```

```
#> Will remove the pkg1, pkg2, and pkg3 packages.
```

Note that the length is only used for non-numeric vectors (when `is.numeric(x)` return `FALSE`). If you want to use the length of a numeric vector, convert it to character via `as.character()`.

You can combine collapsed vectors with "no", like this:

```
pkgs <- character()
cli_text("Will remove {?no/the/the} {.pkg {pkgs}} package{?s}.")
```

```
#> Will remove no packages.
```

```
pkgs <- c("pkg1", "pkg2", "pkg3")
cli_text("Will remove {?no/the/the} {.pkg {pkgs}} package{?s}.")
```

```
#> Will remove the pkg1, pkg2, and pkg3 packages.
```

When the pluralization markup contains three alternatives, like above, the first one is used for zero, the second for one, and the third one for larger quantities.

Choosing the right quantity:

When the text contains multiple glue `{}` substitutions, the one right before the pluralization markup is used. For example:

```
nfiles <- 3; ndirs <- 1
cli_text("Found {nfiles} file{?s} and {ndirs} director{?y/ies}")
```

```
#> Found 3 files and 1 directory
```

This is sometimes not the the correct one. You can explicitly specify the correct quantity using the `qty()` function. This sets that quantity without printing anything:

```
nupd <- 3; ntotal <- 10
cli_text("{nupd}/{ntotal} {qty(nupd)} file{?s} {?needs/need} updates")
```

```
#> 3/10 files need updates
```

Note that if the message only contains a single `{}` substitution, then this may appear before or after the pluralization markup. If the message contains multiple `{}` substitutions *after* pluralization markup, an error is thrown.

Similarly, if the message contains no `{}` substitutions at all, but has pluralization markup, an error is thrown.

Rules

The exact rules of cli's pluralization. There are two sets of rules. The first set specifies how a quantity is associated with a {?} pluralization markup. The second set describes how the {?} is parsed and interpreted.

Quantities:

1. `{}` substitutions define quantities. If the value of a `{}` substitution is numeric (when `is.numeric(x)` holds), then it has to have length one to define a quantity. This is only enforced if the `{}` substitution is used for pluralization. The quantity is defined as the value of `{}` then, rounded with `as.integer()`. If the value of `{}` is not numeric, then its quantity is defined as its length.
2. If a message has {?} markup but no `{}` substitution, an error is thrown.
3. If a message has exactly one `{}` substitution, its value is used as the pluralization quantity for all {?} markup in the message.
4. If a message has multiple `{}` substitutions, then for each {?} markup cli uses the quantity of the `{}` substitution that precedes it.
5. If a message has multiple `{}` substitutions and has pluralization markup without a preceding `{}` substitution, an error is thrown.

Pluralization markup:

1. Pluralization markup starts with {? and ends with }. It may not contain { and } characters, so it may not contain `{}` substitutions either.
2. Alternative words or suffixes are separated by `/`.
3. If there is a single alternative, then *nothing* is used if `quantity == 1` and this single alternative is used if `quantity != 1`.
4. If there are two alternatives, the first one is used for `quantity == 1`, the second one for `quantity != 1` (including `quantity == 0`).
5. If there are three alternatives, the first one is used for `quantity == 0`, the second one for `quantity == 1`, and the third one otherwise.

See Also

Other pluralization: `no()`, `pluralize()`

pluralize *String templating with pluralization*

Description

`pluralize()` is similar to `glue::glue()`, with two differences:

- It supports cli's pluralization syntax, using {?} markers.
- It collapses substituted vectors into a comma separated string.

Usage

```
pluralize(
  ...,
  .envir = parent.frame(),
  .transformer = glue::identity_transformer
)
```

Arguments

`..., .envir, .transformer`
: All arguments are passed to `glue::glue()`.

Details

See pluralization and some examples below.

See Also

Other pluralization: `no()`, `pluralization`

Examples

```
# Regular plurals
nfile <- 0; pluralize("Found {nfile} file{?s}.")
nfile <- 1; pluralize("Found {nfile} file{?s}.")
nfile <- 2; pluralize("Found {nfile} file{?s}.")

# Irregular plurals
ndir <- 1; pluralize("Found {ndir} director{?y/ies}.")
ndir <- 5; pluralize("Found {ndir} director{?y/ies}.")

# Use 'no' instead of zero
nfile <- 0; pluralize("Found {no(nfile)} file{?s}.")
nfile <- 1; pluralize("Found {no(nfile)} file{?s}.")
nfile <- 2; pluralize("Found {no(nfile)} file{?s}.")

# Use the length of character vectors
pkgs <- "pkg1"
```

```
pluralize("Will remove the {pkgs} package{?s}.")
pkgs <- c("pkg1", "pkg2", "pkg3")
pluralize("Will remove the {pkgs} package{?s}.")

pkgs <- character()
pluralize("Will remove {?no/the/the} {pkgs} package{?s}.")
pkgs <- c("pkg1", "pkg2", "pkg3")
pluralize("Will remove {?no/the/the} {pkgs} package{?s}.")

# Multiple quantities
nfiles <- 3; ndirs <- 1
pluralize("Found {nfiles} file{?s} and {ndirs} director{?y/ies}")

# Explicit quantities
nupd <- 3; ntotal <- 10
cli_text("{nupd}/{ntotal} {qty(nupd)} file{?s} {?needs/need} updates")
```

`progress-c` *The cli progress C API*

Description

The cli progress C API

The cli progress C API

`CLI_SHOULD_TICK`**:**

A macro that evaluates to (int) 1 if a cli progress bar update is due, and to (int) 0 otherwise. If the timer hasn't been initialized in this compilation unit yet, then it is always 0. To initialize the timer, call `cli_progress_init_timer()` or create a progress bar with `cli_progress_bar()`.

`cli_progress_add()`**:**

```
void cli_progress_add(SEXP bar, int inc);
```

Add a number of progress units to the progress bar. It will also trigger an update if an update is due.

- `bar`: progress bar object.
- `inc`: progress increment.

`cli_progress_bar()`**:**

```
SEXP cli_progress_bar(int total, SEXP config);
```

Create a new progress bar object. The returned progress bar object must be `PROTECT()`-ed.

- `total`: Total number of progress units. Use `NA_INTEGER` if it is not known.
- `config`: R named list object of additional parameters. May be `NULL` (the C NULL~) or R_NilValue(the RNULL‘) for the defaults.

`config` may contain the following entries:

- name: progress bar name.
- status: (intiial) progress bar status.
- type: progress bar type.
- total: total number of progress units.
- show_after: show the progress bar after the specified number of seconds. This overrides the global show_after option.
- format: format string, must be specified for custom progress bars.
- format_done: format string for successful termination.
- format_failed: format string for unsuccessful termination.
- clear: whether to remove the progress bar from the screen after termination.
- auto_terminate: whether to terminate the progress bar when the number of current units equals the number of total progress units.

Example:

```
#include <cli/progress.h>
SEXP progress_test1() {
  int i;
  SEXP bar = PROTECT(cli_progress_bar(1000, NULL));
  for (i = 0; i < 1000; i++) {
    cli_progress_sleep(0, 4 * 1000 * 1000);
    if (CLI_SHOULD_TICK) cli_progress_set(bar, i);
  }
  cli_progress_done(bar);
  UNPROTECT(1);
  return Rf_ScalarInteger(i);
}
```

`cli_progress_done()`:

```
void cli_progress_done(SEXP bar);
```

Terminate the progress bar.

- bar: progress bar object.

`cli_progress_init_timer()`:

```
void cli_progress_init_timer();
```

Initialize the cli timer without creating a progress bar.

`cli_progress_num()`:

```
int cli_progress_num();
```

Returns the number of currently active progress bars.

`cli_progress_set()`:

```
void cli_progress_set(SEXP bar, int set);
```

Set the progress bar to the specified number of progress units.

- bar: progress bar object.

- `set`: number of current progress progress units.

`cli_progress_set_clear()`:

```
void cli_progress_set_clear(SEXP bar, int clear);
```

Set whether to remove the progress bar from the screen. You can call this any time before `cli_progress_done()` is called.

- `bar`: progress bar object.
- `clear`: whether to remove the progress bar from the screen, zero or one.

`cli_progress_set_format()`:

```
void cli_progress_set_format(SEXP bar, const char *format, ...);
```

Set a custom format string for the progress bar. This call does not try to update the progress bar. If you want to request an update, call `cli_progress_add()`, `cli_progress_set()` or `cli_progress_update()`.

- `bar`: progress bar object.
- `format`: format string.
- `...`: values to substitute into `format`.

`format` and `...` are passed to `vsnprintf()` to create a format string.

Format strings may contain glue substitutions, referring to progress variables, pluralization, and cli styling.

`cli_progress_set_name()`:

```
void cli_progress_set_name(SEXP bar, const char *name);
```

Set the name of the progress bar.

- `bar`; progress bar object.
- `name`: progress bar name.

`cli_progress_set_status()`:

```
void cli_progress_set_status(SEXP bar, const char *status);
```

Set the status of the progress bar.

- `bar`: progress bar object.
- `status` : progress bar status.

`cli_progress_set_type()`:

```
void cli_progress_set_type(SEXP bar, const char *type);
```

Set the progress bar type. Call this function right after creating the progress bar with `cli_progress_bar()`. Otherwise the behavior is undefined.

- `bar`: progress bar object.
- `type`: progress bar type. Possible progress bar types: `iterator`, `tasks`, `download` and `custom`.

`cli_progress_update()`:

```
void cli_progress_update(SEXP bar, int force, int add, int set);
```

Update the progress bar. Unlike the simpler `cli_progress_add()` and `cli_progress_set()` function, it can force an update if `force` is set to 1.

- `bar`: progress bar object.
- `set`: the number of current progress units. It is ignored if negative.
- `inc`: increment to add to the current number of progres units. It is ignored if `set` is not negative.
- `force`: whether to force an update, even if no update is due.

To force an update without changing the current number of progress units, supply `set = -1`, `inc = 0` and `force = 1`.

`progress-variables` *Progress bar variables*

Description

Progress bar variables

Details

These variables can be used in cli progress bar format strings.

- `pb_bar` creates a visual progress bar. If the number of total units is unknown, then it will return an empty string.
- `pb_current` is the number of current progress units.
- `pb_current_bytes` is the number of current progress utils formatted as bytes. The output has a constant width of six characters.
- `pb_elapsed` is the elapsed time since the start of the progress bar. The time is measured since the progress bar is created with `cli_progress_bar()` or similar.
- `pb_elapsed_clock` is the elapsed time, in hh::mm::ss format.
- `pb_elapsed_raw` is the number of seconds since the start of the progress bar.
- `pb_eta` is the estimated time until the end of the progress bar, in human readable form.
- `pb_eta_raw` is the estimated time until the end of the progress bar, in seconds.
- `pb_eta_str` is the estimated time until the end of the progress bar. It includes the `"ETA:"` prefix. It is only shown if the time can be estimated, otherwise it is the empty string.
- `pb_extra` can be used to access extra data, see the `extra` argument of `cli_progress_bar()` and `cli_progress_update()`.
- `pb_id` is the id of the progress bar. The id has the format cli-<pid>-<counter> where <pid> is the process id, and <counter> is an integer counter that is incremented every time cli needs a new unique id.
- `pb_name` is the name of the progress bar. This is supplied by the developer, and it is by default the empty string. A space character is added to non-empty names.

- pb_percent is the percentage of the progress bar, always formatted in three characters plus the percentage sign. If the total number of units is unknown, then it is " NA%".
- pb_pid is the integer process id of the progress bar.
- pb_rate is the progress rate, in number of units per second, formatted in a string.
- pb_rate_raw is the raw progress rate, in number of units per second.
- pb_rate_bytes is the progress rate, formatted as bytes per second, in human readable form.
- pb_spin is a spinner. The default spinner is selected via a get_spinner() call.
- pb_status is the status string of the progress bar. By default this is an empty string, but it is possible to set it in cli_progress_bar() and 'cli_progress_update()].
- pb_timestamp is a time stamp in ISO 8601 format.
- pb_total is the total number of progress units, or NA if the number of units is unknown.
- pb_total_bytes is the total number of progress units, formatted as bytes, in a human readable format.

Examples

```
# pb_bar and pb_percent
cli_progress_demo(
  format = "Progress bar: {cli::pb_bar} {cli::pb_percent}",
  total = 100
)

# pb_current and pb_total
cli_progress_demo(
  format = "[{cli::pb_current}/{cli::pb_total}]",
  total = 248
)

# pb_current_bytes, pb_total_bytes
cli_progress_demo(
  format = "[{cli::pb_current_bytes}/{cli::pb_total_bytes}]",
  total = 102800,
  at = seq(0, 102800, by = 1024)
)
```

---

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")
```

`simple_theme` *A simple CLI theme*

Description

Note that this is in addition to the builtin theme. To use this theme, you can set it as the `cli.theme` option:

Usage

```
simple_theme(dark = getOption("cli_theme_dark", "auto"))
```

Arguments

dark	Whether the theme should be optiomized for a dark background. If "auto", then cli will try to detect this. Detection usually works in recent RStudio versions, and in iTerm on macOS, but not on other platforms.

Details

```
options(cli.theme = cli::simple_theme())
```

and then CLI apps started after this will use it as the default theme. You can also use it temporarily, in a div element:

```
cli_div(theme = cli::simple_theme())
```

See Also

themes, builtin_theme().

Examples

```
cli_div(theme = cli::simple_theme())

cli_h1("Heading 1")
cli_h2("Heading 2")
cli_h3("Heading 3")

cli_alert_danger("Danger alert")
cli_alert_warning("Warning alert")
cli_alert_info("Info alert")
cli_alert_success("Success alert")
cli_alert("Alert for starting a process or computation",
  class = "alert-start")

cli_text("Packages and versions: {.pkg cli} {.version 1.0.0}.")
cli_text("Time intervals: {.timestamp 3.4s}")

cli_text("{.emph Emphasis} and  {.strong strong emphasis}")

cli_text("This is a piece of code: {.code sum(x) / length(x)}")
cli_text("Function names: {.fn cli::simple_theme}")

cli_text("Files: {.file /usr/bin/env}")
cli_text("URLs: {.url https://r-project.org}")

cli_h2("Longer code chunk")
cli_par(class = "code R")
cli_verbatim(
  '# window functions are useful for grouped mutates',
  'mtcars %>%',
  '  group_by(cyl) %>%',
  '  mutate(rank = min_rank(desc(mpg)))')
cli_end()
```

```
cli_h2("Even longer code chunk")
cli_par(class = "code R")
cli_verbatim(format(ls))
cli_end()

cli_end()
```

---

spark_bar *Draw a sparkline bar graph with unicode block characters*

Description

Rendered using block elements. In most common fixed width fonts these are rendered wider than regular characters which means they are not suitable if you need precise alignment.

Usage

```
spark_bar(x)
```

Arguments

x	A numeric vector between 0 and 1

Details

You might want to avoid sparklines on non-UTF-8 systems, because they do not look good. You can use `is_utf8_output()` to test for support for them.

See Also

`spark_line()`

Examples

```
x <- seq(0, 1, length = 6)
spark_bar(x)
spark_bar(sample(x))

spark_bar(seq(0, 1, length = 8))

# NAs are left out
spark_bar(c(0, NA, 0.5, NA, 1))
```

spark_line *Draw a sparkline line graph with Braille characters.*

Description

You might want to avoid sparklines on non-UTF-8 systems, because they do not look good. You can use `is_utf8_output()` to test for support for them.

Usage

```
spark_line(x)
```

Arguments

x	A numeric vector between 0 and 1

See Also

`spark_bar()`

Examples

```
x <- seq(0, 1, length = 10)
spark_line(x)
```

start_app *Start, stop, query the default cli application*

Description

`start_app` creates an app, and places it on the top of the app stack.

Usage

```
start_app(
  theme = getOption("cli.theme"),
  output = c("auto", "message", "stdout", "stderr"),
  .auto_close = TRUE,
  .envir = parent.frame()
)

stop_app(app = NULL)

default_app()
```

Arguments

theme	Theme to use.
output	How to print the output.
.auto_close	Whether to stop the app, when the calling frame is destroyed.
.envir	The environment to use, instead of the calling frame, to trigger the stop of the app.
app	App to stop. If NULL, the current default app is stopped. Otherwise we find the supplied app in the app stack, and remote it, together with all the apps above it.

Details

stop_app removes the top app, or multiple apps from the app stack.

default_app returns the default app, the one on the top of the stack.

Value

start_app returns the new app, default_app returns the default app. stop_app does not return anything.

---

style_hyperlink *Terminal Hyperlinks*

Description

Terminal Hyperlinks

Usage

```
style_hyperlink(text, url)

ansi_has_hyperlink_support()
```

Arguments

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

Details

ansi_hyperlink() creates an ANSI hyperlink.

ansi_has_hyperlink_support() checks if the current stdout() supports hyperlinks.

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

Value

Styled ansi_string for style_hyperlink(). Logical scalar for ansi_has_hyperlink_support().

Examples

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

---

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

Description

Various handy symbols to use in a command line UI

Usage

```
symbol

list_symbols()
```

Format

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

Details

On Windows they have a fallback to less fancy symbols.

list_symbols() prints a table with all symbols to the screen.

Examples

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

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

`test_that_cli` *Test cli output with testthat*

Description

Use this function in your testthat test files, to test cli output. It requires testthat edition 3, and works best with snapshot tests.

Usage

```
test_that_cli(desc, code, configs = NULL)
```

Arguments

`desc`	Test description, passed to `testthat::test_that()`, after appending the name of the cli configuration to it.
`code`	Test code, it is modified to set up the cli config, and then passed to `testthat::test_that()`
`configs`	cli configurations to test `code` with. The default is `NULL`, which includes all possible configurations. It can also be a character vector, to restrict the tests to some configurations only. See available configurations below.

Details

`test_that_cli()` calls `testthat::test_that()` multiple times, with different cli configurations. This makes it simple to test cli output with and without ANSI colors, with and without Unicode characters.

Currently available configurations:

- `plain`: no ANSI colors, ASCII characters only.
- `ansi`: ANSI colors, ASCII characters only.
- `unicode`: no ANSI colors, Unicode characters.
- `fancy`; ANSI colors, Unicode characters.

See examples below and in cli's own tests, e.g. in https://github.com/cran/cli/blob/master/tests/testthat and the corresponding snapshots at https://github.com/cran/cli/tree/master/tests/testthat/_snaps

Importtant note regarding Windows:

Because of base R's limitation to record Unicode characters on Windows, we suggest that you record your snapshots on Unix, or you restrict your tests to ASCII configurations.

Unicode tests on Windows are automatically skipped by testthat currently.

Examples

```
# testthat cannot record or compare snapshots when you run these
# examples interactively, so you might want to copy them into a test
# file

# Default configurations
cli::test_that_cli("success", {
  testthat::local_edition(3)
  testthat::expect_snapshot({
    cli::cli_alert_success("wow")
  })
})

# Only use two configurations, because this output does not have colors
cli::test_that_cli(configs = c("plain", "unicode"), "cat_bullet", {
  testthat::local_edition(3)
  testthat::expect_snapshot({
    cli::cat_bullet(letters[1:5])
  })
})

# You often need to evaluate all cli calls of a test case in the same
# environment. Use `local()` to do that:
cli::test_that_cli("theming", {
  testthat::local_edition(3)
  testthat::expect_snapshot(local({
    cli::cli_div(theme = list(".alert" = list(before = "!!! ")))
    cli::cli_alert("wow")
  }))
})
```

`themes` *CLI themes*

Description

CLI elements can be styled via a CSS-like language of selectors and properties. Only a small subset of CSS3 is supported, and a lot visual properties cannot be implemented on a terminal, so these will be ignored as well.

Adding themes

The style of an element is calculated from themes from four sources. These form a stack, and the themes on the top of the stack take precedence, over themes in the bottom.

1. The cli package has a built-in theme. This is always active. See `builtin_theme()`.
2. When an app object is created via `start_app()`, the caller can specify a theme, that is added to theme stack. If no theme is specified for `start_app()`, the content of the `cli.theme` option is used. Removed when the corresponding app stops.

3. The user may speficy a theme in the `cli.user_theme` option. This is added to the stack *after* the app's theme (step 2.), so it can override its settings. Removed when the app that added it stops.
4. Themes specified explicitly in `cli_div()` elements. These are removed from the theme stack, when the corresponding `cli_div()` elements are closed.

Writing themes

A theme is a named list of lists. The name of each entry is a CSS selector. Only a subset of CSS is supported:

- Type selectors, e.g. `input` selects all <input> elements.
- Class selectors, e.g. `.index` selects any element that has a class of "index".
- ID selector. #toc will match the element that has the ID "toc".
- The descendant combinator, i.e. the space, that selects nodes that are descendants of the first element. E.g. div span will match all elements that are inside a element.

The content of a theme list entry is another named list, where the names are CSS properties, e.g. `color`, or `font-weight` or `margin-left`, and the list entries themselves define the values of the properties. See `builtin_theme()` and `simple_theme()` for examples.

Formatter callbacks

For flexibility, themes may also define formatter functions, with property name `fmt`. These will be called once the other styles are applied to an element. They are only called on elements that produce output, i.e. *not* on container elements.

Supported properties

Right now only a limited set of properties are supported. These include left, right, top and bottom margins, background and foreground colors, bold and italic fonts, underlined text. The `before` and `after` properties are supported to insert text before and after the content of the element.

The current list of properties:

- `after`: A string literal to insert after the element. It can also be a function that returns a string literal. Supported by all inline elements, list items, alerts and rules.
- `background-color`: An R color name, or HTML hexadecimal color. It can be applied to most elements (inline elements, rules, text, etc.), but the background of containers is not colored properly currently.
- `before`: A string literal to insert before the element. It can also be a function that returns a string literal. Supported by all inline elements, list items, alerts and rules.
- `class-map`: Its value can be a named list, and it specifies how R (S3) class names are mapped to cli class names. E.g. `list(fs_path = "file")` specifies that `fs_path` objects (from the fs package) should always print as `.file` objects in cli.
- `color`: Text color, an R color name or a HTML hexadecimal color. It can be applied to most elements that are printed.

- `collapse`: Specifies how to collapse a vector, before applying styling. If a character string, then that is used as the separator. If a function, then it is called, with the vector as the only argument.
- `digits`: Number of digits after the decimal point for numeric inline element of class `.val`.
- `fmt`: Generic formatter function that takes an input text and returns formatted text. Can be applied to most elements. If colors are in use, the input text provided to `fmt` already includes ANSI sequences.
- `font-style`: If `"italic"` then the text is printed as cursive.
- `font-weight`: If `"bold"`, then the text is printed in boldface.
- `line-type`: Line type for `cli_rule()`.
- `list-style-type`: String literal or functions that returns a string literal, to be used as a list item marker in un-ordered lists.
- `margin-bottom`, `margin-left`, `margin-right`, `margin-top`: Margins.
- `padding-left`, `padding-right`: This is currently used the same way as the margins, but this might change later.
- `start`: Integer number, the first element in an ordered list.
- `string_quote`: Quoting character for inline elements of class `.val`.
- `text-decoration`: If `"underline"`, then underlined text is created.
- `text-exdent`: Amound of indentation from the second line of wrapped text.
- `transform`: A function to call on glue substitutions, before collapsing them. Note that `transform` is applied prior to implementing color via ANSI sequences.
- `vec_last`: The last separator when collapsing vectors.
- `vec_sep`: The separator to use when collapsing vectors.
- `vec_trunc`: Vectors longer than this will be truncated. Defaults to 100.

More properties might be added later. If you think that a property is not applied properly to an element, please open an issue about it in the cli issue tracker.

Examples

Color of headings, that are only active in paragraphs with an 'output' class:

```
list(
  "par.output h1" = list("background-color" = "red", color = "#e0e0e0"),
  "par.output h2" = list("background-color" = "orange", color = "#e0e0e0"),
  "par.output h3" = list("background-color" = "blue", color = "#e0e0e0")
)
```

Create a custom alert type:

```
list(
  ".alert-start" = list(before = symbol$play),
  ".alert-stop"  = list(before = symbol$stop)
)
```

`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(),
  trim = FALSE
)
```

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()`.
`trim`	Whether to avoid traversing the same nodes multiple times. If `TRUE` and `data` has a `trimmed` column, then that is used for printing repeated noded.

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")

# Trimming
pkgdeps <- list(
  "dplyr@0.8.3" = c("assertthat@0.2.1", "glue@1.3.1", "magrittr@1.5",
    "R6@2.4.0", "Rcpp@1.0.2", "rlang@0.4.0", "tibble@2.1.3",
    "tidyselect@0.2.5"),
  "assertthat@0.2.1" = character(),
  "glue@1.3.1" = character(),
  "magrittr@1.5" = character(),
  "pkgconfig@2.0.3" = character(),
  "R6@2.4.0" = character(),
  "Rcpp@1.0.2" = character(),
  "rlang@0.4.0" = character(),
  "tibble@2.1.3" = c("cli@1.1.0", "crayon@1.3.4", "fansi@0.4.0",
     "pillar@1.4.2", "pkgconfig@2.0.3", "rlang@0.4.0"),
  "cli@1.1.0" = c("assertthat@0.2.1", "crayon@1.3.4"),
  "crayon@1.3.4" = character(),
  "fansi@0.4.0" = character(),
  "pillar@1.4.2" = c("cli@1.1.0", "crayon@1.3.4", "fansi@0.4.0",
     "rlang@0.4.0", "utf8@1.1.4", "vctrs@0.2.0"),
  "utf8@1.1.4" = character(),
  "vctrs@0.2.0" = c("backports@1.1.5", "ellipsis@0.3.0",
     "digest@0.6.21", "glue@1.3.1", "rlang@0.4.0", "zeallot@0.1.0"),
  "backports@1.1.5" = character(),
  "ellipsis@0.3.0" = c("rlang@0.4.0"),
  "digest@0.6.21" = character(),
  "glue@1.3.1" = character(),
  "zeallot@0.1.0" = character(),
  "tidyselect@0.2.5" = c("glue@1.3.1", "purrr@1.3.1", "rlang@0.4.0",
```

```
    "Rcpp@1.0.2"),
  "purrr@0.3.3" = c("magrittr@1.5", "rlang@0.4.0")
)

pkgs <- data.frame(
  stringsAsFactors = FALSE,
  name = names(pkgdeps),
  deps = I(unname(pkgdeps))
)

tree(pkgs)
tree(pkgs, trim = TRUE)

# Mark the trimmed nodes
pkgs$label <- pkgs$name
pkgs$trimmed <- paste(pkgs$name, " (trimmed)")
tree(pkgs, trim = TRUE)
```

Index