Package 'cli'

January 9, 2020

Title Helpers for Developing Command Line Interfaces

Version 2.0.1

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 integrates with the 'crayon' package to support 'ANSI' terminal colors.

License MIT + file LICENSE

LazyData true

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

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

RoxygenNote 7.0.2

Depends R (>= 2.10)

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

Suggests callr, covr, htmlwidgets, knitr, mockery, rmarkdown, rstudioapi, prettycode (>= 1.1.0), testthat, withr

Encoding UTF-8

VignetteBuilder cli

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2020-01-08 23:01:45 UTC

R topics documented:

`ansi-styles` *ANSI colored text*

Description

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

Usage

```
bg_black(...)

bg_blue(...)

bg_cyan(...)

bg_green(...)

bg_magenta(...)

bg_red(...)

bg_white(...)

bg_yellow(...)

col_black(...)

col_blue(...)

col_cyan(...)

col_green(...)

col_magenta(...)

col_red(...)

col_white(...)

col_yellow(...)

col_grey(...)
```

```
col_silver(...)

style_dim(...)

style_blurred(...)

style_bold(...)

style_hidden(...)

style_inverse(...)

style_italic(...)

style_reset(...)

style_strikethrough(...)

style_underline(...)
```

Arguments

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

Details

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

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

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

- `style_reset()` to remove any style, including color,
- `style_bold()` for boldface / strong text, although some terminals show a bright, high intensity text instead,
- `style_dim()` (or `style_blurred()` reduced intensity text.
- `style_italic()` (not widely supported).
- `style_underline()`,
- `style_inverse()`,
- `style_hidden()`,
- ‘style_strikethrough() (not widely supported).

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

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

Value

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

See Also

Other ANSI styling: `combine_ansi_styles()`, `make_ansi_style()`

Examples

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

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

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

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

---

ansi_hide_cursor *Hide/show cursor in a terminal*

Description

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

Usage

```
ansi_hide_cursor(stream = stderr())

ansi_show_cursor(stream = stderr())

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

Arguments

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

Details

`ansi_hide_cursor()` hides the cursor.

`ansi_show_cursor()` shows the cursor.

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

---

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_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_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_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 retuned 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 = list(), ...)

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

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

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

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 {.cal ...} expressions.

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 retuned 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 retuned 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_fone() 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("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 compuration. 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.x
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_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 `crayon::num_colors()`.
- `console_with`: detected console width.

Usage

```
cli_sitrep()
```

Value

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

Examples

```
cli_sitrep()
```

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")
)
```

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 then 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 then 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 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 startus 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 retuned 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_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()`

Examples

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

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

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

console_width *Determine the width of the console*

Description

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

Usage

```
console_width()
```

Value

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

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 are have an `.auto_close` argument, which allows automatically closing a container element, 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)
```

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

Description

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

Usage

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

Arguments

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

Value

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

See Also

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

Examples

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

`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:

- `emph` for emphasized text.
- `strong` for strong importance.
- `code` for a piece of code.
- `pkg` for a package name.

- `fun` for a function name.
- `arg` for a function argument.
- `key` for a keyboard key.
- `file` for a file name.
- `path` for a path (essentially the same as `file`).
- `email` for an email address.
- `url` for a URL.
- `var` for a variable name.
- `envvar` for the name of an environment variable.
- `val` for a "value".

See examples below.

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

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. See examples below.

Formatting values

The `val` inline class formats values. By default (c.f. the builtin 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 referring 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.

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}}")

## 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 = stderr())
```

Arguments

stream	The stream to check.

Value

TRUE or FALSE.

See Also

Other terminal capabilities: is_dynamic_tty()

Examples

```
is_ansi_tty()
```

---

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

Description

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

Usage

```
is_dynamic_tty(stream = cli_output_connection())
```

Arguments

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

Details

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

The detection mechanism is as follows:

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

4. If R_CLI_DYNAMIC is not empty and set to anything else, FALSE is returned.
5. If the stream is a terminal, then TRUE is returned.
6. If the stream is 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,
  border_style = "single",
  padding = 1,
  margin = 0,
  float = c("left", "center", "right"),
  col = NULL,
  background_col = NULL,
  border_col = col,
  align = c("left", "center", "right"),
  width = console_width()
)
```

Arguments

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

About fonts and terminal settings

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

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

Examples

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

## All border styles
list_border_styles()

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

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

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

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

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

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

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

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

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

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

```
)
```

`list_spinners` *List all available spinners*

Description

List all available spinners

Usage

```
list_spinners()
```

Value

Character vector of all available spinner names.

See Also

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

Examples

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

`make_ansi_style` *Create a new ANSI style*

Description

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

Usage

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

Arguments

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

Details

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

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

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

Value

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

See Also

Other ANSI styling: ansi-styles, combine_ansi_styles()

Examples

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

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

---

make_spinner *Create a spinner*

Description

Create a spinner

Usage

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

Arguments

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

Value

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

`cli_spinner` methods:

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

All methods return the spinner object itself, invisibly.

The spinner is automatically throttled to its ideal update frequency.

Examples

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

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

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

```
  sp2$finish()
}
ansi_with_hidden_cursor(fun_with_spinner3())
```

See Also

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

`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`

`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.

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 `{}` substituions at all, but has pluralization markup, and 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 (i.e. `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 with a preceding {} substitution, and error is thrown.

Pluralization markup:

1. Pluralization markup start with {? and ends with }. It may not contain { and } characters, so it may not contains {} 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` (include 0).
5. If there are three alternatives, the first one is used for `quantity == 0`, the second for `quantity == 1`, and the third otherwise.

See Also

Other pluralization: `no()`

`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()
```

---

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.

---

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")
```

`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 builtin 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.

More properties might be adder later.

Please see the example themes and the source code for now for the details.

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