Package: blockr.script (via r-universe)

November 19, 2026

Title Script and Prose Blocks for 'blockr'

Version 0.0.0.9000

Description A block for 'blockr' boards whose state is an ordinary R script. The script reads any number of upstream results and returns whatever its last statement evaluates to. A top-level assignment of a plain value becomes a control on the block, and the exported code is the script with the current values written in. A prose block holds markdown with inline R, as 'Quarto' writes it, and reads its inputs the same way.

License GPL (>= 3)

URL https://github.com/BristolMyersSquibb/blockr.script

BugReports https://github.com/BristolMyersSquibb/blockr.script/issues

Depends R (>= 4.1.0)

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

Remotes BristolMyersSquibb/blockr.core, BristolMyersSquibb/blockr.ui

Imports blockr.core (>= 0.1.4), blockr.ui, evaluate, htmltools, jsonlite, shiny, stats, tools, utils

Suggests dm, dplyr, ggplot2, grid, gt, gtsummary, knitr, lattice, reactives, testthat (>= 3.0.0)

Config/roxygen2/version 8.1.0

Config/pak/sysreqs cmake make libuv1-dev zlib1g-dev

Repository https://cynkra.r-universe.dev

Date/Publication 2026-11-17 17:32:12 UTC

RemoteUrl https://github.com/BristolMyersSquibb/blockr.script

RemoteRef HEAD

RemoteSha 97d43fbbad22046d9a2f119f1c3f94038eefb7dd

Contents

block_result_output.html_page . . . 2
block_result_update . . . 4
html_page . . . 6
inline_r . . . 8
new_prose_block . . . 9
new_script_block . . . 10
register_script_blocks . . . 14
result_head . . . 14
result_titles . . . 16
script_expr . . . 17
sentence_data . . . 18

Index **20**

`block_result_output.html_page`
Render a block result, dispatching on the result's class

Description

`block_output()` dispatches on the BLOCK, which is the right hinge for a block that always draws the same kind of thing. A function or code block draws whatever its code returned, so the thing worth dispatching on is the RESULT: a data frame, a gt table, a ggplot, a package's own table class.

Usage

```
## S3 method for class 'html_page'
block_result_output(result, block, session, ...)

block_result_output(result, block, session, ...)

## Default S3 method:
block_result_output(result, block, session, ...)
```

Arguments

`result`	The block's evaluated result.
`block`	The block object.
`session`	The block's Shiny module session.
`...`	Passed to methods.

Details

The default method is `render_dynamic_output()`, the type switch this package has always used, so adding this generic changes nothing on its own. It exists so another package can teach a function block to draw its own object without that object's renderer having to live here – and, unlike an `htmltools::as.tags()` method, a method here receives the block's module `session`. That is what a renderer needs to namespace an element id, register a `downloadHandler` on `session$output`, or observe a click on `session$input`.

For example, a package with its own table class can register a method that draws the table through blockr.viz's renderer rather than through gt.

Value

A shiny render function, as `blockr.core::block_output()` requires.

Called on every re-evaluation

blockr.core re-runs `block_output()` whenever the block's result changes (`output_render_observer()`). Assigning to `session$output` replaces the output and can be done on every call. An observer must be created once per block and not once per call, so a method keeps a note that it has made them.

That note, and any state the method keeps with it, has to end with the block. blockr.core destroys a block's session when the block is removed (`session$destroy()`). The observers, outputs and reactive values made in it are then gone, and Shiny refuses every later read or write of them. `session$userData` is shared by the whole Shiny session and is not cleared, so a note kept there is still found when a block is added again under the same id. Remove it in `session$onDestroy()`:

```
key <- paste0("my_table:", session$ns(""))
if (is.null(session$userData[[key]])) {
  session$userData[[key]] <- TRUE
  session$onDestroy(function() session$userData[[key]] <- NULL)
  shiny::observeEvent(session$input$row, my_click(session$input$row))
}
```

Updating in place

The function a method returns replaces the block's output, so the browser draws the result again on every evaluation and what the user did to it (a scroll position, a search text) is lost. A result class that can change what it has on screen also registers a `block_result_update()` method.

The head row

A method can put its own tools, such as a download button, into the block's head row and give the block's sentence a title with `result_head()`. It reads the title, subtitle and caption now on screen with `result_titles()`.

See Also

`block_result_update()`

Examples

```
## Not run:
# In a package with its own table class `my_table`:
block_result_output.my_table <- function(result, block, session, ...) {
  shiny::renderUI(draw_my_table(result))
}

## End(Not run)
```

`block_result_update` *Update a block result that is on screen*

Description

`block_result_output()` draws a result from scratch: the function it returns replaces the block's output, and the browser builds the result again. A table then loses its scroll position, its search text and its collapsed rows on every change of a control. A class that can change what it has drawn registers a method here. The method sends the new result to the browser itself, for example with `session$sendCustomMessage()` or with the update function of the widget it drew, and returns `TRUE`. The block then leaves its output as it is.

Usage

```
block_result_update(result, block, session, ...)

## Default S3 method:
block_result_update(result, block, session, ...)
```

Arguments

`result`	The block's evaluated result.
`block`	The block object.
`session`	The block's Shiny module session.
`...`	Passed to methods.

Value

`TRUE` when the result on screen was updated. Anything else has the block draw the result from scratch.

When it is called

The block calls the method when the result it drew last is still on screen and the new result has the same class vector. In every other case it calls `block_result_output()`:

- the first result of a block, also of a block that was removed and added again under the same id;
- a result of another class;

- the first result after an error or after an input was unavailable;
- a result whose predecessor has not reached the browser: a result evaluated twice while the block was hidden, or a second evaluation in the flush that drew the first;
- after a method here returned anything but TRUE, or failed.

So a method can return FALSE for a change it cannot apply in place, such as a table with other columns, and the result is drawn from scratch.

A class without a method is drawn as before: the function its block_result_output() method returns is the block's output, unchanged.

What the method has to do

The output keeps the HTML the first render sent, and the render function of that render does not run again. So everything that follows the result has to be sent by this method: the data, and a caption the result draws itself. Use ids that are the same in every call, such as session$ns("table").

result_head() and result_titles() work as in block_result_output(). The head row starts empty before the call, so call result_head() with the same arguments as in the first render. result_titles(), called in the method, gives the sentence as it is now, and the block calls the method again when the sentence changes.

What the method can rely on

When the method is called, the HTML that block_result_output() drew for the last result was sent to the browser in an earlier flush. A message the method sends therefore arrives after that HTML, and after every message an earlier call sent.

It cannot rely on more than that. The block does not know whether the browser has the element in the page at this moment: a layout can take a hidden card out of the document. A message for an element that is not there is lost unless the widget keeps it, so a widget that can be hidden this way should apply the last message when it is attached again. If the browser attaches the output element again without a new evaluation, it draws the HTML of the first render.

The block also knows only the class of what it drew last, not which method drew it. A method that sometimes hands the result on with NextMethod() has to note that itself, and return FALSE here while the drawing in the page is not its own.

See Also

block_result_output()

Examples

```
## Not run:
# A table widget with an update function of its own.
block_result_output.my_table <- function(result, block, session, ...) {
  id <- session$ns("table")
  shiny::renderUI(my_table_widget(result$data, id = id))
}

block_result_update.my_table <- function(result, block, session, ...) {
  update_my_table_widget(session, session$ns("table"), result$data)
```

```
  TRUE
}

## End(Not run)
```

`html_page` *A web page as a block result*

Description

`html_page()` returns an object that draws as a sandboxed iframe. A script block that ends with it shows the page on its card. Give it either `html`, a page you write, or `url`, a page that lives elsewhere.

Usage

```
html_page(
  html = NULL,
  url = NULL,
  data = NULL,
  height = "auto",
  sandbox = NULL,
  max_rows = 1000L
)
```

Arguments

`html`	The page, a single string or a character vector of lines.
`url`	A URL.
`data`	A data frame, a list or a vector that scripts in an `html` page read as `blockr.data`. `NULL` for none.
`height`	Height of the frame: a number of pixels or a CSS length such as `"400px"` or `"50vh"`. `"auto"` fits an `html` page to its content; a URL page cannot be measured and defaults to `"600px"`.
`sandbox`	The iframe's `sandbox` attribute, a single string. `NULL` takes the default for the mode.
`max_rows`	Rows of `data` passed to the page. More are dropped with a warning. `Inf` for no limit.

Value

An object of class `html_page`.

HTML

`html` goes into the frame as it is. To put values from R into it, build the string in the script, with `paste0()`, `sprintf()` or a package such as glue. `htmltools::htmlEscape()` escapes a value that may hold < or &.

Scripts in the page read `data` as JSON from `blockr.data`, an array of row objects for a data frame, or `null` without data. Missing values are `null`. Date-times are written in UTC (`"2026-01-02T03:04:05Z"`).

The data travels to the browser with the page, again each time the block re-renders. `max_rows` limits it.

The page runs in an iframe with `sandbox = "allow-scripts"`: its CSS and JS cannot reach the board. The sandbox does not keep the page from sending its data to other hosts. A script, an image or a style sheet in the page can load from anywhere and pass along what the page holds. With `height = "auto"` the iframe grows to fit the page, up to 20000 pixels.

URL

The URL has to start with `http://` or `https://`. A value from R goes in with `paste0()`; `utils::URLencode()` with `reserved = TRUE` encodes it. The other site has to allow framing: a site that sends `X-Frame-Options: DENY` or `SAMEORIGIN`, or a CSP `frame-ancestors` that does not list the board's host, stays blank. A link above the frame opens the page in a new tab.

The frame keeps `allow-same-origin`, because an external app needs its own cookies and storage. The sandbox therefore only protects the board from pages on another host. A URL on the board's own host can reach the board and everything the viewer can do on it.

Outside a board

At the console, printing the object opens the page in the viewer. In a script run with `Rscript` it prints one line. In an R Markdown or Quarto document with HTML output the frame is part of the document.

Examples

```
title <- "Hello"
html_page(paste0("<h1>", title, "</h1>"))

bands <- data.frame(name = c("Mick", "John"), band = c("Stones", "Beatles"))
html_page(
  paste0(
    "<ul>",
    paste0("<li>", bands$name, ": ", bands$band, "</li>", collapse = ""),
    "</ul>"
  )
)

html_page(
  "<ul id='l'></ul><script>
  blockr.data.forEach(function (r) {
    document.getElementById('l').insertAdjacentHTML(
      'beforeend', '<li>' + r.name + '</li>'
    );
```

```
  });
  </script>",
  data = bands
)

q <- "blockr"
html_page(
  url = paste0("https://example.org/search?q=", utils::URLencode(q, TRUE))
)
```

`inline_r` *Inline R in markdown*

Description

Writes the value of every `` `r expr` `` in a markdown string into the text, as knitr does for inline code: each expression is evaluated with the objects in `data` in scope, and its value replaces the code.

Usage

```
inline_r(text, data = list())
```

Arguments

`text`	A markdown string.
`data`	A named list of objects the expressions can use.

Details

Left alone: code in fenced blocks, other inline code, and inline code written between double backticks to show the syntax (`` `` `r x` `` ``). Code in a block indented by four spaces is evaluated, as knitr does.

A value has to be a vector of at most 100 elements. A number is written in full (`1000000`, not `1e+06`) and several values are joined by commas. `NULL` writes nothing. When the code holds several expressions, the value of the last one is written.

Each expression is evaluated in a fresh environment whose parent is the global environment. `data` is always bound there (`NULL` when the list has none), so it never resolves to `utils::data`. `<<-` writes to the global environment, which all sessions of a Shiny app share.

Value

The markdown string with the values written in.

Examples

```
inline_r("There are `r nrow(data)` cars.", list(data = mtcars))
```

`new_prose_block` *Prose block*

Description

Text that can compute. The block holds markdown, edited as it reads (bold as bold, no syntax to learn), and R inside it the way Quarto writes inline code: `` `r nrow(data)` ``. Typing `` `r `` and a space in the editor opens a small code field that suggests the inputs, their columns and a few common functions; Enter computes it, and the text shows the value, marked with a dotted underline. Pointing at a value shows its code, a click edits it.

Usage

```
new_prose_block(text = character(), ...)
```

Arguments

`text`	Markdown string, with inline R as `` `r expr` ``.
`...`	Forwarded to `blockr.core::new_block()`.

Details

The markdown with its code is the block's setting, in its control. Its result is the markdown with the values written in, which the next block takes as text (a report, a slide). Because the code is Quarto's own inline code, the same text runs in a Quarto document where the inputs are objects of the same names.

An expression that fails marks its value in the editor; in the result it stops the block with the error.

Values refresh eagerly as you type, outside the DAG. The text itself commits on blur or Ctrl+Enter, never on a keystroke, so typing does not recompute the blocks downstream.

Value

A `prose_block` (also a `text_block`).

Who can run code

Whoever can edit the text can run R code on the server, as with the script block. The code runs in a fresh environment whose parent is the global environment, so `<<-` and `assign(..., envir = globalenv())` write to a place that all sessions of the app share.

Inputs

The block takes any number of inputs, including none, named as in `new_script_block()`:

- The first link into the block is `data`.
- Every further link is read by its link name, or as `.arg2`, `.arg3`, ... when the link has none.
- With no input connected the text is plain text, or computes with values that need no input.

On export each input the text uses is replaced by the id of the upstream block.

Examples

```
if (interactive()) {
  library(blockr.core)
  serve(
    new_board(
      blocks = list(
        data = new_dataset_block("iris", "datasets"),
        note = new_prose_block("## Iris\n\nThere are **`r nrow(data)`** rows.")
      ),
      links = links(from = "data", to = "note", input = "data")
    )
  )
}
```

`new_script_block` *Script block*

Description

A block whose state is an ordinary R script. The script reads its inputs by name and returns whatever its last statement evaluates to: a data frame for the blocks downstream, a ggplot, a gt table, or any other object.

Usage

```
new_script_block(
  script = "n <- 6\n\nutils::head(data, n)",
  values = list(),
  ...
)
```

Arguments

`script`	R code, either a single string with embedded newlines or a character vector of lines, which is joined with `\n`.
`values`	Named list of current control values, restored with a saved board. Values for names the script no longer declares are dropped.
`...`	Two arguments are read here, and the rest is passed to `blockr.core::new_block()`. • `starters`: the scripts "Start from" offers in the editor, as a list of `list(label, hint, script)`, where `hint` may be left out. `NULL`, the default, keeps the built-in ones. • `class`: classes to put in front of `script_block`, for a block built on this one. Neither is part of the block's state, which is why they are not named arguments: blockr.core saves every named argument of a constructor. A restored block gets both from its constructor again.

Details

A script that does not parse, or that has no statement returning a result, still gives a block, with a warning. The block then shows that reason as its error. A saved board that holds such a script loads, and the script can be repaired in the editor.

Value

A block of class `script_block`.

Inputs

The block takes any number of inputs, including none.

- The first link into the block is `data`. A script written for one input reads `data`, as it would in a function.
- Every link also reads by its own name: the link name, or `.arg1`, `.arg2`, ... for a link that has none. Only the links without a name are counted, so next to one named link the first unnamed one is `.arg1`. Links are named where they are drawn: "Edit link" or "Edit inputs" in the dock.
- The block is variadic (`function(id, ...args)`), so the order of the links decides which one is `data`. A link that is removed and drawn again comes last, and `data` is then another input. A script with more than one input should read each by its own name, which does not move. A link named `data` is always `data`.
- With no input connected, `data` is `NULL` and the script is a source: it reads a file, calls an API or builds a data frame itself.

On export each input name is replaced by the id of the upstream block, so the exported code reads like a script someone wrote by hand.

Controls

A top-level assignment whose right-hand side is a **plain value** becomes a control on the block, meaning a literal, a `c()` of literals, or a call to `factor()` or `as.Date()`. Everything else is code.

```
.vars <- intersect(c("AGE", "SEX", "TRT"), names(data))  # private, no control

vars <- factor(.vars, levels = .vars)               # a multi-select
species <- factor("setosa", levels(data$Species))   # a dropdown
n <- 10                                             # a number box
desc <- TRUE                                        # a checkbox

data |>
  dplyr::filter(Species %in% species) |>
  dplyr::slice_head(n = n)
```

A **factor** is how a select is expressed: its levels are the choice list, and the length of its value decides single or multiple. For a text column write `sort(unique(data$col))` as the levels. `unique()` alone follows the order of the rows, and the control is drawn again whenever that changes. A pick whose level is no longer in the data is not used: the control goes back to the value the script wrote.

A name starting with a dot is never a control. A name the script assigns again is not one either, since only the last assignment could be.

The script reads a control as its current value, written into the code as a literal. Three things follow, all by design:

- A select's pick is a string (a character vector for a multi-select), not a factor. `levels(x)` or `as.integer(x)` on a control do not give what they give for the factor on the declaration line.
- Every read of the control's name is replaced, also inside data-masking calls. With a control `n`, `dplyr::filter(n > 5)` compares the control and not a column called `n`. Write `.data$n` for the column. `.env$n` does not reach the control: its line is no longer in the code that runs.
- A later declaration reads an earlier control at its current value, so the choices of one dropdown can depend on the pick in another: `col <- factor("SEX", names(data))` followed by `val <- factor(character(0), sort(unique(data[[col]])))`.

The optional `#|` annotation carries what a value cannot say, such as a numeric range or a label: `n <- 10 #| number(min = 1, max = 50)`.

Sentence

`#| title:`, `#| subtitle:` and `#| caption:` lines give the block a sentence whose words are its controls, as blockr.viz's blocks do:

```
arm <- factor("Placebo", levels = sort(unique(data$TRT01A)))
top_n <- 8  #| number(min = 3, max = 15)
serious <- FALSE  #| label = "serious events only"
#| title: Top {@top_n} adverse events in {@arm}
#| subtitle: Subjects per term[, {@serious}]
```

`{@arg}` prints a control's value and makes the word clickable: a select opens its list, a number or text becomes a field, a checkbox toggles. A clause in `[ ... ]` leaves when its value is empty, and its control is offered as a + `label` chip instead. `{n}`, `{label(col)}` and `{col}` read the first input, as the data frame `sentence_data()` makes of it. The sentence sits in the row with the gear; the controls it names leave the grid, the others stay. A date can be named but stays a control in the grid. The sentence is shown above any result and is not part of the exported code.

Exported code

The current values are written into the body as literals, so the exported code has no assignment left for a control.

blockr.core fills the input names in with `bquote()`, which also evaluates every `.()` and `..()` the script calls itself, such as data.table's `.(x)`. Write `list(x)` there. The editor footer names the line.

Saved as lines

A saved board writes the script as a JSON array, one element per line, so a change to the script shows up in a git diff as the lines that changed.

Building on the script block

Another package can build a block on this one. Its constructor calls `new_script_block()` with a `class`, its own default script and, if it likes, its own `starters`:

```
new_listing_block <- function(script = "my_listing(data)", values = list(),
                              ...) {
  blockr.script::new_script_block(
    script, values,
    starters = my_starters(),
    class = "listing_block",
    ...
  )
}
```

The block has the classes `c("listing_block", "script_block", "block")`, so every method of the script block applies to it. A saved board restores it through `new_listing_block()`, which therefore takes `script`, `values` and `...`, no other named argument, and passes `...` on. Register the constructor with `blockr.core::register_blocks()` as any other block.

The constructor is found by its arguments: it is the nearest function in the chain of calls that takes exactly `script`, `values` and `...`. It may therefore reach `new_script_block()` through a helper of its own. Where there is no such function, as for `new_script_block(class = "x")` called at top level, the block is saved with a constructor that keeps the `class` and the `starters` it was built with.

What such a block draws is decided by the class of the script's result, not by the class of the block: see `block_result_output()`, `result_head()` and `result_titles()`. For an input that is not a data frame, see `sentence_data()`.

Examples

```
blk <- new_script_block("n <- 6\n\nutils::head(data, n)")

# the same script, one element per line
blk <- new_script_block(c("n <- 6", "", "utils::head(data, n)"))

# no input: the script is the source
blk <- new_script_block("data.frame(x = 1:3, y = c(\"a\", \"b\", \"c\"))")

if (interactive()) {
  library(blockr.core)
  serve(
    new_board(
      blocks = c(
        a = new_dataset_block("band_members", package = "dplyr"),
        b = new_dataset_block("band_instruments", package = "dplyr"),
        s = new_script_block("merge(data, instruments, by = \"name\")")
      ),
      links = links(
        from = c("a", "b"),
        to = c("s", "s"),
        input = c("data", "instruments")
```

```
      )
    )
  )
}
```

`register_script_blocks` *Register the script and prose blocks*

Description

Called on load. Exported so an app that unregistered it can add it back.

Usage

```
register_script_blocks()
```

Value

`NULL`, invisibly.

`result_head` *Fill the head row of a script block from a result method*

Description

A script block has one head row: the sentence on the left, the gear on the right. A `block_result_output()` method calls `result_head()` to put its own tools into that row, left of the gear, and to give the sentence titles for the lines the script does not declare.

Usage

```
result_head(session, tools = NULL, titles = NULL, caption = TRUE)
```

Arguments

`session`	The block's session, as the method received it.
`tools`	Tags for the head row, or `NULL` for none.
`titles`	A list with any of `title`, `subtitle` and `caption`, each one plain string. The sentence shows them for the lines the script does not declare. A line the script declares wins, and a script line with nothing after the colon (`#| title:`) switches that line off. The strings are shown as they are; `{...}` is not read in them.
`caption`	`FALSE` when the result draws the caption itself.

Details

The head row starts empty before every render. A method that does not call `result_head()` leaves the row as the script block draws it, so the tools of one result class are gone as soon as the script returns another. The method is called on every re-evaluation; calling `result_head()` again with the same arguments redraws nothing. The tools are hidden while the block has no result on screen, for example after an error.

Value

`NULL`, invisibly.

Tools

`tools` is drawn as it is. A direct child that is a link or a button gets the size and look of an icon button (26 px square), so give it an icon as its content and a `title` as its name. Ids are namespaced with `session$ns()` and their outputs and inputs live on `session`, as anywhere in the method. Assigning a `downloadHandler` to `session$output` on every call is fine. An observer on `session$input` must be created once per block and not once per call, see `block_result_output()`.

Caption

The script block draws a `#| caption:` line in the head row, under the subtitle. A result that draws the caption itself, such as a table with the caption under it, passes `caption = FALSE` and reads the text from `result_titles()`. The head row then leaves the caption out, so there is one copy on screen. A control that only the caption names stays in the grid of controls, because the head row has no word for it.

See Also

`result_titles()` for the sentence as text, `block_result_output()` for the method this is called from.

Examples

```
## Not run:
block_result_output.my_table <- function(result, block, session, ...) {
  session$output$csv <- shiny::downloadHandler(
    filename = "table.csv",
    content = function(file) utils::write.csv(result$data, file)
  )
  result_head(
    session,
    tools = shiny::downloadLink(
      session$ns("csv"), label = shiny::icon("download"),
      title = "Download"
    ),
    titles = list(title = "My table"),
    caption = FALSE
  )
  shiny::renderUI(
```

```
    shiny::tagList(
      draw_my_table(result),
      shiny::tags$p(result_titles(session)$caption)
    )
  )
}

## End(Not run)
```

`result_titles` *The sentence of a script block as text*

Description

The title, subtitle and caption a script block shows, as the plain text now on screen. For a `block_result_output()` method that needs them outside the head row: in a downloaded file, or for a caption drawn under a table.

Usage

```
result_titles(session, ui = TRUE)
```

Arguments

`session`	The block's session, as the method received it.
`ui`	`TRUE` for the sentence as on screen, `FALSE` for the sentence without the script's `ui_*` lines.

Details

A line the script declares is resolved with the current control values and the data tokens (`{n}`, `{label(col)}`). A line it does not declare is the title the method gave to `result_head()`. A line that is neither, or that the script switched off, is `NULL`.

The sentence can change without the block being evaluated again, for example when a control that only the title names is changed. So read it where it is used and do not keep a copy: inside a render function the call takes a reactive dependency and the output follows the sentence, inside the `content` function of a `downloadHandler` it gives the current text.

Value

A list with

- `title`, `subtitle` and `caption`, each one string or `NULL`;
- `filters`, the filter trail that `{filters}` prints, as a character vector (empty when the input carries none);
- `declared`, a character vector with those of `"title"`, `"subtitle"` and `"caption"` that the script has a line for;
- `templates`, a list keyed by the names in `declared`, each the line's text as written, `""` for a line that is switched off.

Lines for the app only

A script can carry `#| ui_title:`, `#| ui_subtitle:` and `#| ui_caption:` lines, which are meant for the app: a `ui_title` or `ui_caption` replaces the plain line on screen and a `ui_subtitle` is added to it. With `ui = TRUE` the result is the sentence as on screen. A download or a report passes `ui = FALSE` and gets the sentence built from the plain lines only. Both forms follow the controls.

What the script declares

`declared` names the lines the script has, also one it switched off with nothing after the colon. So for `caption`:

- in `declared` and a string: the script's caption;
- in `declared` and `NULL`: the script switched the caption off, or it resolved to nothing;
- not in `declared` and a string: the title the method gave;
- not in `declared` and `NULL`: there is none.

`templates` holds the declared lines as the script wrote them, before anything is filled in, for a rule that depends on the wording, such as a caption that is exactly `{filters}`.

See Also

`result_head()`

Examples

```
## Not run:
block_result_output.my_table <- function(result, block, session, ...) {
  session$output$csv <- shiny::downloadHandler(
    filename = "table.csv",
    content = function(file) {
      # without the lines the script wrote for the app only
      writeLines(c(result_titles(session, ui = FALSE)$title, ""), file)
      utils::write.table(result$data, file, append = TRUE, sep = ",")
    }
  )
  shiny::renderUI(draw_my_table(result))
}

## End(Not run)
```

`script_expr` *A script as one expression*

Description

The code a script block runs, built from the script alone: the controls are replaced by their values and the lines that declared them are gone. It is for a place that has the script but no block and no input, such as the call a report makes.

Usage

```
script_expr(script, values = list(), data = "data")
```

Arguments

script	The script, as for new_script_block().
values	Named list of control values, as a block's state holds them.
data	The name the expression reads the first input by. Every read of data in the script becomes a read of this name.

Details

A control takes its value from values, and the script's own value where values has none. Without an input, a declaration that reads the input, such as factor("a", levels = unique(data$col)), cannot be worked out. When values holds a value for it, that value is written in. When it does not, the line stays in the expression as code and is evaluated with the rest.

Value

An expression, or NULL when the script does not parse or has no statement left.

Examples

```
script_expr("n <- 6\n\nutils::head(data, n)", list(n = 3))

# the levels need the input, the value comes from `values`
script_expr(
  "arm <- factor(\"A\", levels = unique(data$ARM))\nsubset(data, ARM == arm)",
  values = list(arm = "B"),
  data = "adsl"
)
```

---

sentence_data *The data frame the sentence reads*

Description

The data tokens of a sentence ({n}, {label(col)}, {col}, {filters}) and the column labels in the menu of a word read a data frame. The script block calls this generic on its first input to get one, so a package whose blocks hand on something else can say what the sentence sees.

Usage

```
sentence_data(x, script = NULL, ...)

## Default S3 method:
sentence_data(x, script = NULL, ...)
```

Arguments

x	The block's first input.
script	The block's script, one string. A method can read it to answer for the tables the script names.
...	Not used by the default method.

Details

The default method returns a data frame as it is. A dm gives one data frame with the columns of all its tables side by side and no rows, with the column labels kept. Where two tables have a column of the same name, the first table's is used. {label(col)} then works for a column of any table, and {n} is 0. Anything else gives NULL: the data tokens are empty then, and a clause in [...] that holds one leaves.

A method returns a data frame or NULL. A character vector in the attribute blockr_filters of that data frame is the filter trail that {filters} prints and that result_titles() returns as filters.

Value

A data frame, or NULL when the sentence has nothing to read.

Examples

```
sentence_data(datasets::iris[1:3, ])
sentence_data("not a data frame")

## Not run:
# A package whose blocks hand on a dm with a filter trail
sentence_data.dm <- function(x, script = NULL, ...) {
  out <- NextMethod()
  attr(out, "blockr_filters") <- my_filter_trail(x)
  out
}

## End(Not run)
```

Index

block_result_output
        *(*block_result_output.html_page*)*, 2
block_result_output(), *4*, *5*, *13–16*
block_result_output.html_page, 2
block_result_update, 4
block_result_update(), *3*
blockr.core::block_output(), *3*
blockr.core::new_block(), *9*, *10*
blockr.core::register_blocks(), *13*

html_page, 6
htmltools::htmlEscape(), *7*

inline_r, 8

new_prose_block, 9
new_script_block, 10
new_script_block(), *9*, *18*

paste0(), *7*

register_script_blocks, 14
result_head, 14
result_head(), *3*, *5*, *13*, *16*, *17*
result_titles, 16
result_titles(), *3*, *5*, *13*, *15*, *19*

script_expr, 17
sentence_data, 18
sentence_data(), *12*, *13*
sprintf(), *7*

utils::URLencode(), *7*