Package: hovercraft (via r-universe)

November 27, 2026

Title Tooltips for 'Shiny' and 'bslib' Pages

Version 0.0.90

Description Puts a tip on any 'HTML' tag in a 'Shiny' app or a 'bslib' page. The tip shows when the pointer rests on the tag or keyboard focus arrives, and holds plain text or a named template filled in from the values of one item, such as a row of a table. Tips follow the page's light and dark mode and can be themed, with a contrast check. They are drawn by a small 'JavaScript' file that ships with the package.

License MIT + file LICENSE

Copyright inst/COPYRIGHTS

URL https://insightsurge.github.io/hovercraft/, https://github.com/insightsurge/hovercraft

BugReports https://github.com/insightsurge/hovercraft/issues

Depends R (>= 4.4.0)

Imports checkmate, cli, htmltools, jsonlite, rlang

Suggests bslib (>= 0.6.0), callr, chromote, DT, httpuv, knitr, later, lifecycle, pkgload, promises, reactable, rmarkdown, shiny, shinycssloaders, shinyjs, testthat (>= 3.1.7), withr

Config/quality-gate/coverage 70

Config/launchR/version 0.0.19

Config/testthat/edition 3

Config/testthat/parallel true

VignetteBuilder knitr

Encoding UTF-8

Language en-GB

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.1.1

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

Date/Publication 2026-11-27 16:51:17 UTC

RemoteUrl https://github.com/insightsurge/hovercraft

RemoteRef HEAD

RemoteSha 87f2e3c87c108adaf5ee6299fbf4fe5f162969d5

Contents

hovercraft_book . . . 2
tip_blocks . . . 3
tip_define . . . 5
tip_marker . . . 6
tip_more . . . 8
tip_on . . . 10
tip_output . . . 18
tip_presets . . . 20
tip_server . . . 21
tip_spec . . . 24
tip_style . . . 26
tip_table . . . 28
tip_template . . . 33
tip_theme . . . 35
use_hovercraft . . . 36

Index **40**

`hovercraft_book` *Open hovercraft book*

Description

[Experimental] Opens hovercraft book, a Shiny app that shows every feature of the package on one page, in six parts: getting started, content, interaction, look, tips from Shiny, and tables. Each section is one feature: a live figure to rest the pointer on, a sentence of choices that change it, the R code that makes what you see (ready to copy), a short list of what to test, and the Shiny inputs the page sends. The code a section shows is the code that draws its figure. The index has a search box, and the checklist (the key C) shows one test step at a time and ticks a step when the server receives the input that proves it.

Usage

```
hovercraft_book(section = NULL, ..., checklist = FALSE, run = TRUE)
```

Arguments

section	The section to open on, or NULL (the default) for the first: one of "start", "placement", "templates", "disabled", "pins", "pin_limits", "more", "more_button", "inputs", "words", "looks", "size", "colours", "markers", "server", "output", "tables", "columns", "table_pins", "in_place" or "table_server".
...	Must be empty. It makes a misspelt argument an error.
checklist	TRUE to open with the checklist on; FALSE (the default) to open without it.
run	TRUE (the default) to run the app, which needs an interactive R session; FALSE to return it.

Details

The book needs the suggested packages shiny and bslib; its table sections also need DT or reactable. Its files are in system.file("book", package = "hovercraft").

Unlike the other functions of the package, its name does not start with tip_: it opens the package's own book, not a tip.

Value

With run = FALSE, the app, a shiny.appobj. With run = TRUE, what shiny::runApp() returns when the app stops.

Examples

```
app <- hovercraft_book(section = "pins", run = FALSE)
inherits(app, "shiny.appobj")
```

---

tip_blocks *Blocks for a tip's template*

Description

[Stable] The parts a template is made of. tip_title() is the first line, tip_text() a paragraph of markdown, tip_badge() a short label, tip_fields() a list of label and value pairs, and tip_code() a piece of code shown as text. Text and values may hold placeholders, {{ field }} or {{ field | number:2 }}, filled from the record of the item the tip belongs to. A placeholder names one field, matched exactly (dots and spaces allowed), and takes one filter at most: number (digits after the point, rounded as R does) or percent (the value times 100, then digits, then a percent sign). A value that is missing or blank shows a dash. There is no way to write a literal {{ yet.

Usage

```
tip_title(text)

tip_text(text)

tip_badge(text, status = NULL)

tip_fields(...)

tip_code(field, language = NULL, copy = FALSE)
```

Arguments

text	One string, with placeholders if you like. In `tip_text()` it is markdown (strong, emphasis, code, links, short lists), and a value put in by a placeholder is never read as markdown.
status	What the badge says about its item, or `NULL` for a plain badge: one of `"success"`, `"warning"`, `"danger"` or `"info"`, the words bslib uses, or a placeholder such as `"{{ tone }}"` filled from the record. Any other fixed word is an error. When filled from a record, one of the four words colours the badge and any other value shows a plain badge. The badge is coloured from the page's Bootstrap colours for that status (with plain colours on a page without Bootstrap), and its text should say the same thing, because colour alone is not enough. A block written as a plain list, `list(type = "badge", text = "a", tone = "info")`, uses the engine's name for it, `tone`, with the same rule.
...	Named strings for `tip_fields()`: each name is a label (plain text) and each value a string that may hold placeholders, as in `tip_fields(Records = "{{ records | number:0 }}")`.
field	One string: the name of the record field that holds the code. The code is shown as text, never run, and never filled in by placeholders.
language	One string naming the code's language, or `NULL`.
copy	`TRUE` asks for a Copy button under the code, which lives in a tip that stays open: give the tag that has this template `pin = TRUE` or `pin = "click"` in `tip_on()`, or the button never shows. The button copies the code as text, also on a page that is not served over https (where the browser's clipboard is closed to a page, the copy is done the older way), and says "Copied" or "Copy failed" beside the button. With `events = "copy"` in `tip_on()`, the server reads each press as `input$<id>_copy`.

Value

A block, to give to `tip_template()`.

See Also

Other tooltips: `tip_define()`, `tip_marker()`, `tip_more()`, `tip_on()`, `tip_output()`, `tip_presets()`, `tip_server`, `tip_spec()`, `tip_style()`, `tip_table()`, `tip_template()`, `tip_theme()`, `use_hovercraft()`

Examples

```
tip_title("{{ label }}")
tip_text("**{{ differences | number:0 }}** differences")
tip_badge("{{ status }}", status = "warning")
tip_fields(Records = "{{ records | number:0 }}", Files = "{{ files }}")
tip_code("sql", language = "sql")
```

---

tip_define *Name the templates for a page*

Description

[Stable] Puts named templates on the page, so a tag can use one with `tip_on(tag, template = "name")`. Put the result in the page's UI, once. Names are page-wide. A name is lower-case letters, digits and _, starting with a letter; inside a Shiny module, prefix it with `ns()` (for example `tip_define(!!ns("row") := tip_template(...))`) so two copies of the module keep their own templates without overwriting each other. Defining the same name again replaces the earlier template under it, with a warning when the blocks differ.

Usage

```
tip_define(...)
```

Arguments

`...`	Templates, each with a name: `tip_define(diff = tip_template(...))`, or in a Shiny module `tip_define(!!ns("diff") := tip_template(...))`. A template may also be a list of blocks. Supports dynamic dots (`rlang::list2()`).

Details

The templates are also kept in R, so that `tip_on()` can check a record against them. Defined in a Shiny session (in `renderUI()` or a module's server), they belong to that session: two sessions of one app may define one name with different fields and each is checked against its own. Defined where the UI is built, they belong to the process and every session is checked against them; a session may define the name again for itself, without a warning. A `renderUI()` that is cached with `bindCache()` does not run again for the next session, so a template it defines is known in R only to the session that ran it: define a template that sessions share where the UI is built.

Value

Tags to put in the page's UI.

See Also

Other tooltips: `tip_blocks`, `tip_marker()`, `tip_more()`, `tip_on()`, `tip_output()`, `tip_presets()`, `tip_server`, `tip_spec()`, `tip_style()`, `tip_table()`, `tip_template()`, `tip_theme()`, `use_hovercraft()`

Examples

```
tip_define(
  diff = tip_template(
    tip_title("{{ label }}"),
    tip_text("{{ differences | number:0 }} differences")
  )
)
```

`tip_marker` *A marker for a tipped tag*

Description

[Experimental] Describes the visible cue on a tipped tag, to give as `marker =` to `tip_on()`, `tip_table()` or `tip_spec()`. For the common cases `marker = "underline"`, `"icon"` or `"corner"` is enough; use `tip_marker()` to choose the line of an underline or to give your own icon.

Usage

```
tip_marker(type, ..., line = NULL, icon = NULL)
```

Arguments

`type`	The kind of marker: `"underline"`, `"icon"`, `"corner"` or `"none"`. A table draws only `"corner"`.
`...`	Must be empty. It makes a misspelt argument an error.
`line`	Only for `type = "underline"`: `"dotted"`, `"dashed"` or `"wavy"`, or `NULL` (the default) to follow the tip preset in use. `"wavy"` is a long gentle wave in the text's own colour, not the browser's short zigzag.
`icon`	Only for `type = "icon"`: an icon as a tag, such as `fontawesome::fa()` or `bsicons::bs_icon()`, or `NULL` (the default) for the built-in one. It is added to the tag as a child that a screen reader skips.

Value

A `hovercraft_marker` object, to give as `marker =` to `tip_on()`, `tip_table()` or `tip_spec()`. Its fields are internal; pass the object on.

The types

- `"underline"` draws a line under the tag's words. It takes no room, so nothing moves. The line follows the tip preset in use (see `tip_style()`): dotted for `"none"`, `"sober"` and `"dense"`, dashed for `"modern"` and `"editorial"`, and a long gentle wave for `"lively"`. Choose one with `line`.
- `"icon"` adds a small icon after the words: a circle with an `i`, or the icon you give. It takes a little room.

- `"corner"` draws a small triangle in the top corner of the tag, as a spreadsheet does for a note. It takes no room and suits a table cell or a card; on a single word it can look like a stray mark, so prefer `"underline"` there.
- `"none"` draws nothing. It is what a tag has when `marker` is left out.

A marker is drawn in the tag's own text colour, so it keeps its contrast in light and dark. The pointer over a marked tag is the help cursor, unless the tip can be pinned (it is then a hand), or the tag is a button or a link, which keep their own. A tag that takes more than words, such as the `div` Shiny puts around an input, is marked as a whole: put the marker on the label tag, or on the words, when that is what you mean.

While the tip is open

A marked tag is highlighted while its tip shows, so a reader sees which marker the tip belongs to: a soft grey tint on the words of an underline and on the tag of an icon or a corner (a table cell, a label), and a glow round a `bslib::value_box()`, which has a background of its own. It goes when the tip closes. There is no argument for it, and a tag with no marker is not highlighted. It follows the tip engine's open mark on the tag, for a tip that shows on hover or focus and for one that is pinned.

Words that wrap

For an underline, the words of a block tag (a `td`, a `p`, a `div` and so on) are put in a `span` inside the tag, so that the line goes under the words and not across the whole cell or paragraph. In a `bslib::value_box()` the title is marked. Tags inside the block that are not words are left as they are. A block that holds no words of its own, such as a `div` of other tags, has nothing to underline and stops with an error: put the marker on the tag that holds the words, or use `"corner"` or `"icon"`.

See Also

Other tooltips: `tip_blocks`, `tip_define()`, `tip_more()`, `tip_on()`, `tip_output()`, `tip_presets()`, `tip_server`, `tip_spec()`, `tip_style()`, `tip_table()`, `tip_template()`, `tip_theme()`, `use_hovercraft()`

Examples

```
tip_marker("underline")
tip_marker("underline", line = "wavy")
tip_marker("icon", icon = htmltools::tags$span("?"))

tip_on(
  htmltools::tags$span("Baseline", tabindex = "0"),
  "The first **screening** value",
  marker = tip_marker("underline", line = "dashed")
)
```

tip_more *The long form of a tip, and when it shows*

Description

[Experimental] Wraps the words of a tip's long form with the moment it shows, to give as `more` to `tip_on()` or `tip_spec()`. With no `tip_more()`, `more = "words"` is the same as `more = tip_more("words")`: the tip is short, and a "More" button in it opens the long form.

Usage

```
tip_more(
  content,
  ...,
  on = NULL,
  label = NULL,
  pin = FALSE,
  as = NULL,
  where = NULL
)
```

Arguments

`content`	The words of the long form (markdown, with `{{ field }}` placeholders filled from the tip's record or, in a table, from the row), or a `tip_template()`. `NA` and `""` mean no long form.
`...`	Must be empty. It makes a misspelt argument an error.
`on`	When the long form shows: `"button"` (a "More" button opens it), `"hover"` (at once), `"pin"` (when the tip is pinned) or `"click"` (a click on the item opens it, with no button). `NULL`, the default, is `"button"`. `"click"` is not for a table, where a click selects a row.
`label`	The words of the buttons, only with `"button"`: one string for the "More" button (the "Less" button keeps its word), or `c(more = "Show all", less = "Show less")`. `NULL` gives the built-in "More" and "Less". With `"hover"`, `"pin"` or `"click"` there is no button, so a `label` is an error. It beats the `more` and `less` words of `labels` (see `tip_on()` and `use_hovercraft()`) for this tip.
`pin`	`TRUE` makes the "More" button (and the M key) pin the tip as well as open its long form, so the long form stays. `FALSE`, the default, never pins. Only with `"button"`, and only on a tip that can pin: `tip_on()` with `pin = TRUE` and an `id`, or a row of `tip_table()` with `tip_spec(pin = TRUE)`.
`as`	How the control looks: `"button"` (a small button) or `"link"` (an italic text link). `NULL`, the default, is `"button"`. Only with `on = "button"`; with another moment there is no control, so it is an error.
`where`	Which side of the bottom of the tip the control sits on: `"end"` or `"start"`, in the reading direction. `NULL`, the default, puts a button at the end and a link at the start. Only with `on = "button"`.

Details

With on = "button" (the default) a "More" button at the bottom of the short tip opens the long form in the same tip, and a "Less" button closes it again. It does not pin (unless pin = TRUE, below), and needs no tip pin and no id. The tip is short again the next time it opens. A pinned tip gets the same buttons. The M key opens and closes the long form while the keyboard focus is on the tagged item or the pointer rests on it, except while the focus is in a text field, where a letter is a letter. When the focus is on one item and the pointer rests on another, the key acts on the item under the pointer, the tip the reader sees, not on one around it; a pinned tip that holds the focus keeps the key. A tip that a tap or tip_show() opened does not take the key from the pointer. It is a letter key that works from a hover, so a page switches it off for every reader with use_hovercraft(keys = c(more = FALSE)); there is no switch for one reader. A tag with a long form announces the key as aria-keyshortcuts. The tip stays open and on its side when it grows or shrinks, grows away from its item and scrolls when there is no more room, instead of flipping to the other side; that is the engine's own behaviour (0.58.0).

as and where choose the look and the place of the control that opens the long form (only with on = "button"). as = "button" (the default) is the small button in the bottom corner of the tip, at its end; as = "link" is an italic, underlined text link at the bottom of the tip, at its start. where says the side, "end" or "start", and follows the reading direction, so the end is the right in English and the left in Arabic. Both are the same control to a keyboard and a screen reader: the link is a real <button> styled as a link, which says "button" and "expanded" or "collapsed" to a screen reader, takes Enter and Space, and does not navigate; an <a> would need role = "button" and its own key handling to do the same. It is a tab stop in a pinned tip, M and aria-keyshortcuts work as for the button, and the words are still the ones of label and labels.

With on = "click" a click on the tagged item opens and closes the long form, and the tip has no button. The item should take keyboard focus (tabindex = "0" on a or), because the M key is the keyboard route: it does the same, and the item announces it as aria-keyshortcuts. The long form is announced to a screen reader when it opens; the item gets no aria-expanded and no role, since a click on a plain tag is not a control. The tip must be open: a click that opens the tip (a first tap on a touch screen) only opens it, short, and the next click opens or closes the long form (on a touch screen the engine closes the tip on a second tap, so use the default button there). The click never pins, so pin = "click" with it is an error. A control, or a tag that holds one (a button, an input, a link, a label ...), is refused, because a click on it has its own job, and so is a table: a click on a row selects it. Use the default button there.

With pin = TRUE the "More" button, and the M key, also pin the tip, so the long form stays; "Less" then only closes the long form and the tip stays pinned. It needs a tip that can pin, the same as on = "pin". Today's default, pin = FALSE, never pins: a pinned tip stays pinned when it opens its long form.

With on = "hover" the long form shows at once, with the short one. With on = "pin" it shows when the tip is pinned, so the tip must be able to pin: tip_on(pin = TRUE) with an id, or a row of tip_table() with tip_spec(pin = TRUE); without a pin it is an error.

A tip that expands sends no Shiny input. A pinned tip drawn again, such as one that opens its long form, sends input$<id>_pin with state = "updated"; the values of state may grow in a later version, so an app should not assume it knows them all.

Value

An object of class hovercraft_more, to give as more to tip_on() or tip_spec().

See Also

Other tooltips: tip_blocks, tip_define(), tip_marker(), tip_on(), tip_output(), tip_presets(), tip_server, tip_spec(), tip_style(), tip_table(), tip_template(), tip_theme(), use_hovercraft()

Examples

```
tip_on(
  htmltools::tags$button("Run"),
  "Runs the checks.",
  more = tip_more("Takes about a minute on the full data.")
)

# A click on the item opens the long form, and the More button pins the tip
tip_on(
  htmltools::tags$span("Run", tabindex = "0"),
  "Runs the checks.",
  more = tip_more("Takes about a minute.", on = "click")
)
tip_on(
  htmltools::tags$button("Run", id = "run"),
  "Runs the checks.",
  id = "run_tip",
  pin = TRUE,
  more = tip_more("Takes about a minute.", pin = TRUE)
)

# A text link at the start of the bottom of the tip, instead of the button
tip_on(
  htmltools::tags$button("Run"),
  "Runs the checks.",
  more = tip_more("Takes about a minute.", as = "link")
)

# The long form shows with the short one, and the buttons say other words
tip_on(
  htmltools::tags$button("Run"),
  "Runs the checks.",
  more = tip_more("Takes about a minute.", on = "hover")
)
tip_on(
  htmltools::tags$button("Run"),
  "Runs the checks.",
 more = tip_more("Takes about a minute.", label = c(more = "Details", less = "Fewer details"))
)
```

---

tip_on *Put a tip on a tag*

Description

[Stable] Gives one HTML tag a tip that shows after the pointer rests on it, or keyboard focus arrives, and hides when either leaves. The tip is plain text, or drawn from a named template (`tip_define()`) filled in from a record: the values of one item, such as a row of a table. The engine's script and CSS are attached to the tag, so it works in a bslib page, in a plain Shiny page and in output made later with `renderUI()`. A tag inside `bslib::tooltip()` or `bslib::popover()`, or with `data-bs-toggle`, keeps the tip that bslib draws and this one is left off.

Usage

```
tip_on(
  tag,
  text = NULL,
  ...,
  template = NULL,
  record = NULL,
  style = NULL,
  placement = NULL,
  marker = NULL,
  id = NULL,
  events = NULL,
  when_disabled = NULL,
  more = NULL,
  pin = NULL,
  labels = NULL
)
```

Arguments

`tag`	**[Stable]** One HTML tag, such as `htmltools::tags$tr()` or `shiny::actionButton()`. A list of tags, a tag list or text stops with an error. To tip a tab, put the tip on the tag that is its title, and give the tab a `value`: `nav_panel(title = tip_on(span("Data"), "..."), value = "data")`. Without `value`, bslib writes the tag's HTML into the tab's value.
`text`	**[Stable]** One string of markdown (strong, emphasis, code, links, short lists), or `NULL`. It is shown as written: build it with `paste0()` or `sprintf()`. `NA` and `""` count as none: the tag comes back without a tip. When `record` is given, the `{{ field }}` placeholders in the text are filled from it (as in a template, with one filter at most, `number:n` or `percent:n`), also when `template` is the name of a defined template, which is then not used; when `template` is a `tip_template()`, the text is plain words and its braces stay as written, since the text beats the template. Without a record, braces stay as written. Words with placeholders give the tag no `aria-label`, because they depend on the record.
`...`	Must be empty. It makes a misspelt argument an error.
`template`	**[Stable]** The name of a template defined with `tip_define()`, a `tip_template()` made here, or `NULL`. A `tip_template()` travels with the tag, so it needs no `tip_define()`; its fields are checked against `record`. If `text` is given too, the text is the tip and the template is not used (see `text` for its placeholders).

record **[Stable]** A named list, or a data frame of one row: the values the template's placeholders read, one value each. NA and NULL show a dash. Dates are written as YYYY-MM-DD, times in UTC as YYYY-MM-DDTHH:MM:SSZ and factors as their labels. Once tip_define() has seen the template, only the columns it names are kept, so nothing else in a row is written into the page; for a template not seen here, every column goes, so choose them with data[i, c("label", "n")].

style **[Stable]** as a preset name, **[Experimental]** as a style object. How this tag's tip looks, or NULL (the default) to take the page's. A preset name from tip_presets() ("none" clears a preset set above), a tip_style(), or one part of a style: a tip_theme() or a bslib::bs_theme(). The tag's style beats the page's, and a value set nearer wins, value by value. A list is not taken, and neither is a string that holds : or ; (style takes no CSS); see tip_style(). A list of CSS variables (tip_style(theme = list("--x" = ...))) is for the whole page, so on one tag it stops with an error: give it to use_hovercraft(), or use the roles of tip_theme().

placement **[Stable]** Where the tip opens: "auto", "top", "bottom", "left" or "right", or NULL (the default) for the page's choice, which is "auto" unless a tagged ancestor sets a side. A chosen side flips to the other when there is no room, and "left" and "right" mirror in right-to-left text. On a phone the tip always chooses its own side.

marker **[Experimental]** A visible cue that the tag holds a tip. NULL (the default) and "none" draw none; the others are "underline", "icon" or "corner", a tip_marker() to choose the line of an underline or give an icon, or a tag, which is used as the icon (marker = fontawesome::fa("circle-info") is tip_marker("icon", icon = ...)). See tip_marker() for what each draws. For the cells of a table, use the marker argument of tip_table().

id **[Stable]** A Shiny id for this tip, or NULL (the default) for none: one non-empty string with no whitespace and no :, so ns("baseline_tip") works in a module. Give it a different id from every other tip source on the page. tip_show(), tip_hide() and tip_update() find the tip by it, and events names the inputs after it.

events **[Experimental]** Which Shiny inputs this tag's tip sends, or NULL (the default) for none; one or more of "hint", "pin" and "copy". All are off unless asked for, because they send a message to the server every time a reader meets the tag. They need an id, or a tag with an id of its own; in a module the input is read without the namespace. "pin" and "copy" belong to a pinned tip, so asking for either without a pin stops with an error. Each is a list that holds the fields below.

"hint" sends input$<id>_hint each time the tip opens or closes: state ("shown" or "hidden"), text (the tip's plain words, cut at 200 characters), by ("reader", "host" for tip_show() and tip_hide(), or "data" when the tag left the page), via ("pointer", "focus", "tap", "key", "command", or NULL) and reason (why it closed, NULL on "shown": "leave", "blur", "escape", "outside", "pinned" when the hint became a pin, "replaced", "gone" or "host"). "pin" sends input$<id>_pin each time the tip is pinned, closed or drawn again while pinned: state ("pinned", "closed" or "updated"), text, by ("reader", "host" or "data"), via ("click", "key", "command", or NULL) and reason (why it closed: "select", "escape", "button", "outside", "replaced", "gone",

"below" when a layer around it closed, "host" or "redraw" when it was drawn again with other host slots (by is "data"); NULL on "pinned"). A "state" of "updated" is not a pin: an app that tests only for "closed" should check it. "pin" also keeps input$<id>_pinned, TRUE while the tip is pinned and FALSE after, sent in the same batch as _pin and as a plain value, so an equal value does not fire an observer again; read it with isTRUE(input$run_pinned). The values of state may grow in a later version, so an app should not assume it knows them all. Every one of these inputs is NULL until its first event; in shiny::testServer() set them with session$setInputs(), and in shinytest2 with set_inputs(..., allow_no_input_binding_ = TRUE). On a page without Shiny (R Markdown, Quarto, a static page) events does nothing. "copy" sends input$<id>_copy each time the Copy button of a code block in the pinned tip is pressed: field (the record field the code came from), block (the code block's place among the tip's code blocks, from 0), ok (TRUE when the browser let the copy happen), by ("reader") and via ("click" or "key").

when_disabled **[Experimental]** The reason the control is switched off, as one string of plain text with a word in it, or NULL (the default) for none. While the control is disabled, its tip says "Not available:" and the reason, with a small lock before it; when the control is enabled again the reason goes and the tag's own tip (text or template) is shown as usual. text and template may be left out, for a tag that only has a reason; then a marker is left off with a warning, since it would show on an enabled control that has no tip. The tag must hold a control that can be disabled: a button, input, select, textarea or fieldset. It works when the page is built with the control disabled and when it is switched off later, by the update*(disabled =) functions of Shiny or by script, on a Bootstrap 5 page (bslib) and on the Bootstrap 3 page of a plain fluidPage(); a control disabled by aria-disabled alone is not covered. tip_update() changes the control's own tip, also while it is disabled: the new words show once the control is enabled.

A disabled control takes no keyboard focus and, under Bootstrap 5, no pointer, so tip_on() puts it in a small wrapper that does: a , or a for a tag that is a block, such as the div Shiny puts around an input. While the control is disabled the wrapper is one tab stop, and the control gets no pointer, so the reason is the wrapper's. A button's wrapper has the role of a button; any other control is a group named by the control's label, so a screen reader still says what the field is. The wrapper is not a tab stop when the control is enabled. A control that has the focus when it is switched off hands it to the wrapper, and takes it back when it is switched on. The tip's id and events stay on the control, and the wrapper answers to them while the control is disabled, so tip_show() and the hint input name the control either way. The value returned is the wrapper; calling tip_on() on it again replaces the tip whole, the reason included, and keeps one wrapper when a new when_disabled is given; without one you get the bare control back, with the layout it had before it was wrapped.

The wrapper takes the control's place in a layout: a width, a share of a flex row and a side written in its style, or as the classes w-25, w-50, w-75, w-100, flex-fill, flex-grow-*, col-*, order-*, align-self-* and ms-auto, move to the wrapper, and the control fills the wrapper. A d-block control and the cells of a d-grid keep their width. Not covered: a control that is a direct child of

a btn-group, btn-group-vertical or input-group, since the wrapper then comes between the control and the joined borders; put the tip on the group instead, and do not give a control in a group a reason. A proposed fix is a reason that needs no wrapper, which asks the tip engine for a keyboard stop of its own.

more **[Stable]** as words or a template, **[Experimental]** as a tip_more(). Extra content for the long form of the tip, or NULL (the default) for none: words (markdown, optionally with {{ field }} placeholders filled from record), a tip_template(), or a tip_more() that also says when the long form shows. NA and "" count as none. Words and a template are the same as tip_more() with its default: the tip is short, and a "More" button in it (and the M key) opens the long form in the same tip, with no pin and no id. See tip_more() for "hover", "pin", "click" (a click on the item opens it; not on a control), pin = TRUE (the button pins the tip too), as and where (a text link, and the side the control sits on) and the buttons' words; with "pin" or pin = TRUE the tag must be able to pin and have an id (either id = or the tag's own HTML id), otherwise tip_on() stops with an error.

pin **[Stable]** Whether a reader can keep this tip open, one at a time: NULL (the default) or FALSE for a tip that only shows on hover or focus (FALSE is "never", whatever a level above says), TRUE for a tip that is pinned by the small pin button in the corner of the tip, or by the pin key, and "click" for a tip that a click on the tagged item also pins. "click" is for a tag that has nothing to click on its own: on a control, or on a tag that holds one, it stops with an error, because the click would pin the tip instead of using the control. Use TRUE there: a plain button and a div around an input take it. With TRUE, a click, a key or a tap on the item never pins, so an item that does something when it is clicked, such as a row that selects itself, keeps doing it; a tap shows the tip. A tip that is pinned stays open, with the engine's own close button, until it is closed (the button, Escape, or a click elsewhere), and a code block with tip_code(copy = TRUE) shows its Copy button there. A pin needs an id (the id argument or the tag's own), which names the tip in events, and a tag that the engine can pin: not a link, a submit or reset button, an input, select, textarea, summary, label or editable tag; put the tip on a or around it.

The pin button is drawn over the corner of the tip as a separate element, so a screen reader still gets a tip that it only reads; the first block of the tip keeps room for it, so a one-sentence tip has the button at the end of its line. The pin key is P, which the tagged item announces as aria-keyshortcuts: it pins the tagged item while the focus is on the item or inside it, or while the pointer rests on it, and does nothing while the focus is in a field or while neither holds. When the focus is on one item and the pointer rests on another, the key pins the item under the pointer, the tip the reader sees, not an item around it; an item that is pinned and holds the focus keeps the key. A tip that a tap or tip_show() opened does not take the key from the pointer. The key works from a hover, so a page switches it off for every reader with use_hovercraft(keys = FALSE); there is no switch for one reader. A screen reader that is in browse mode may take the letter for its own navigation, so check the key with the screen reader you support. Escape closes a pinned tip. Pointer, keyboard and tap are tested in Chrome only.

labels **[Experimental]** The words the tip shows itself, such as "Copy", "Close", "More"

and the pin button's name, for this tag's tip and every tip inside the tag, or NULL (the default) to take the page's words (see use_hovercraft()). A named character vector, such as c(close = "Dismiss"); a word left out keeps its built-in one. The keys, with their built-in words, are copy (Copy), copied (Copied), copy_failed (Copy failed), close (Close), not_available (Not available:), pinned ({label}: details pinned. Tab to reach them.), code_line (code: {first}), code_line_empty (code, 1 line), code_lines (code, {n} lines, starting {firs code_lines_empty (code, {n} lines), more (More), more_pinned (More when pinned), less (Less), pin (Pin this tip). Four keys take placeholders, which the page fills in: pinned takes {label} (the name of the item), code_line takes {first} (the first line of the code) and code_lines and code_lines_empty take {n} (the number of lines; code_lines takes {first} too); other braces stay as written. An unknown key, an empty or blank word, or a value that is not words stops with an error; the unknown key's error lists the keys. The nearest words win, key by key: a tag's labels beat the page's, and a tag inside another tag with labels uses the nearer one. tip_more(label =) still sets the words of the "More" and "Less" buttons of one tip, and beats labels there. The words go with the tag, so labels needs a tip on the call (text, template or when_disabled); a second tip_on() on the tag replaces them with its own. With when_disabled, they are also on the wrapper, so "Not available:" changes while the control is off. For a tip_table(), give the words to use_hovercraft(): a table is marked by its own script, and only a tag that has a tip of its own can carry labels around it.

Details

A tip shows on keyboard focus as well as on a pointer, so a tag that cannot take focus (a table row, a div) gives a keyboard user no way to the tip. tip_on() looks at the tag and at everything inside it, so the div that Shiny puts around an input is fine: the input takes focus. It warns once about a tag with nothing focusable in it, and never adds a tabindex itself, because a table of 3,000 rows would become 3,000 Tab stops: give the tags worth a stop tabindex = "0", or put the tip on a button or a link.

A button with no visible words, such as one that holds only an icon, has no name for a screen reader. When it also has no aria-label or aria-labelledby of its own, and the tip is plain text, tip_on() gives it an aria-label made from the text, with the markdown marks taken out. A tip drawn from a template gives no label, because its words depend on the record: set aria-label yourself. The engine reads the tip once: when a tag's name, from aria-label, aria-labelledby or its own words, is the same as the tip's words, the tip is not also given to a screen reader as a description.

If both text and template are given, the text is shown for this tag. A template whose name has been defined in this session is checked against the record here: a field it names that the record does not hold is an error. A name that has not been defined is left to the page, which says so once in the browser's console.

A second tip_on() on a tag that already has a tip replaces the old tip whole: its words, look, id, events, pin and when_disabled all go, and the new call's arguments are the tip. When the old tip had an id, events or a pin and the new call gives none of them, it warns once, with the class hovercraft_warning_retip, so a wrapper that tips a tag again on purpose can silence it with suppressWarnings(classes =).

A tag that already has a `title` attribute keeps it, and the browser then shows its own tooltip beside the tip. `tip_on()` warns once, with the class `hovercraft_warning_title`, and says to remove the `title`; it does not remove it itself, because on an icon it may be the only name.

Value

The tag, with the tip's attributes and the engine's dependency added; with a marker, its attributes, and the marker's own dependency; with `when_disabled`, inside a wrapper that holds the reason.

Stable and experimental

`tag`, `text`, `template`, `record`, `placement`, `id`, `pin` (one pinned tip at a time, with its pin button and the P key), `more` as words or a template (a "More" button opens the long form in the same tip) and a `style` that is a preset name are `r lifecycle::badge("stable")`: their names, values and defaults do not change in 0.1.x. What the long form's button looks like and the keys beyond the pin's are not part of that promise: the look and place of the "More" button, the M key and the `keys` argument of `use_hovercraft()` are `r lifecycle::badge("experimental")`, like `events` and the fields of the inputs it turns on, `when_disabled`, `marker`, a `style` that is a `tip_style()`, a `tip_theme()` or a bslib theme, a `tip_more()`, and keeping more than one pin (`max_pins` of `use_hovercraft()`). They may change.

Known limits

- A tag inside `bslib::tooltip()` or `bslib::popover()`, or with `data-bs-toggle`, keeps the tip that bslib draws.
- A pin closes (reason `"gone"`) when its item is removed, or hidden by a tab, a conditional panel or a sidebar, and it does not come back with the item. A pin whose item only scrolls out of view is hidden and returns when the item does.
- `tip_on()` costs about a millisecond of R time for each tag, and the page's script some time on every change of the page: for a long table, page it.
- Reconnecting an app (`allowReconnect`) is untested.

See Also

Other tooltips: `tip_blocks`, `tip_define()`, `tip_marker()`, `tip_more()`, `tip_output()`, `tip_presets()`, `tip_server`, `tip_spec()`, `tip_style()`, `tip_table()`, `tip_template()`, `tip_theme()`, `use_hovercraft()`

Examples

```
tip_on(htmltools::tags$button("Run"), "Runs the **check**")

# The reason a control is switched off
tip_on(
  htmltools::tags$button("Run", disabled = NA),
  "Runs the **check**",
  when_disabled = "Pick a dataset first"
)

# A tip that a reader can pin, by the button in its corner or by the P key
tip_on(
```

```
  htmltools::tags$button("Run", id = "run"),
  "Runs the **check**",
  id = "run_tip",
  pin = TRUE,
  events = c("hint", "pin")
)

# A short tip with a "More" button that opens the long form in the same tip
tip_on(
  htmltools::tags$button("Run"),
  "Runs the **check**",
  more = "Checks every column and prints a summary."
)

# The long form shows with the short one
tip_on(
  htmltools::tags$button("Run"),
  "Runs the **check**",
  more = tip_more("Checks every column and prints a summary.", on = "hover")
)

# A look for this tag only
tip_on(htmltools::tags$button("Run"), "Runs the **check**", style = "sober")
tip_on(
  htmltools::tags$button("Run"),
  "Runs the **check**",
  style = tip_theme(light = list(bg = "#1b3a6b", text = "#ffffff"))
)

# A visible cue that a word holds a tip
tip_on(
  htmltools::tags$span("Baseline", tabindex = "0"),
  "The first **screening** value",
  marker = "underline"
)

tip_define(diff = tip_template(
  tip_title("{{ label }}"),
  tip_text("{{ differences | number:0 }} differences")
))
rows <- data.frame(label = c("ADAE", "ADSL"), differences = c(3, 0))
lapply(seq_len(nrow(rows)), function(i) {
  tip_on(
    htmltools::tags$tr(htmltools::tags$td(rows$label[[i]]), tabindex = "0"),
    template = "diff",
    record = rows[i, ]
  )
})

# The built-in words of the tip, with other words, for this tip
tip_on(
  htmltools::tags$button("Run"),
  "Runs the **check**",
```

```
  pin = TRUE,
  id = "run",
  labels = c(close = "Dismiss")
)

# An icon-only button is named from its tip
tip_on(htmltools::tags$button(htmltools::tags$i(class = "icon-gear")), "Open the **settings**")
```

`tip_output` *A live Shiny output inside a tip template*

Description

[Experimental] Puts any Shiny output — a plot, a `DT` or `reactable` table, a `gt` table, a chart or `uiOutput()` — inside a `tip_template()`. Give `output` as the Shiny output id (with `ns()` inside a module, such as `ns("detail_plot")`), or as the output tag itself (such as `shiny::plotOutput(ns("detail_plot"))` or one wrapped in a loader like `shinycssloaders::withSpinner()`). It needs Shiny: on a page without a server (R Markdown, Quarto, a static page) the output stays empty.

Usage

```
tip_output(output, keep_alive = TRUE)
```

Arguments

`output`	The Shiny id of the output (one non-empty string with no whitespace, so `ns("detail_plot")` works in a module), or an output tag (such as `shiny::plotOutput("detail_plot")` or a loader-wrapped output tag).
`keep_alive`	`TRUE` (the default) to park the bound output in a hidden holder when the tip closes so it stays alive; `FALSE` to unbind and destroy it on close.

Details

When `output` is given as an id string, place the matching `*Output()` tag in your UI (plain or wrapped in a loader such as `shinycssloaders`); hovercraft parks it out of sight until a tip opens and moves it into whichever item's tip is open. When `output` is given as an output tag, `tip_define()` places it in the hidden holder for you together with its HTML dependencies.

With `keep_alive = TRUE` (the default), closing the tip parks the bound output in a hidden holder instead of destroying it, so reopening the tip shows it at once without recomputing unless its reactive inputs changed. When a parked output is shown again, hovercraft refreshes its visibility and size state so suspended outputs update. With `keep_alive = FALSE`, closing the tip unbinds and removes the output from the page, and the next open builds it again.

While the output computes (including async renders), hovercraft shows a plain loading placeholder unless the output already uses an external loader. `shinycssloaders` is tested; `waiter` and `bones` are expected to work but are untested. If the render errors, the tip shows the error with a Retry button. If the output returns `NULL` or empty text, the tip shows an empty state.

Value

A block, to give to `tip_template()`.

Retry

Retry sets the input `input$<output>_retry` (for an output with the id `report_table`, `input$report_table_retry`). Shiny runs a render again only when something it reads changes, so **the render must read that input** to run again on Retry: put `input$report_table_retry` on a line of its own at the top of the render, as in the example below. If the server neither starts nor finishes the output within a few seconds of the click, the tip goes back to the error state with a message that says the server did not run the output again, and Retry stays available.

One output for several items

One output has one place in the page, so a template that many items share shows one output, drawn for one item at a time. Your server decides for which: ask the items for the `"hint"` input (`tip_on(events = "hint")`) and read `input$<id>_hint` in the render, as in the second example below; its `state` is `"shown"` while the item's hint is open. When a tip opens for a different item than the one the output was last shown for, the tip shows the loading state and hides the old content until the server's new value arrives (a server that never reads the open item sends none, and the old content stays after about a second and a half). A pinned tip sends `input$<id>_pin` (with `events = "pin"`), which the render can read in the same way, and the output of several open pins sits in the tip that opened last: the others say "Shown in another tip." When the tip that holds the output closes, the output goes back to the one that held it before.

The tip's own markup is plain attributes and template JSON with no random or per-session ids, so `shiny::renderUI() |> shiny::bindCache()` and `render*() |> shiny::bindCache()` work across sessions.

See Also

Other tooltips: `tip_blocks`, `tip_define()`, `tip_marker()`, `tip_more()`, `tip_on()`, `tip_presets()`, `tip_server`, `tip_spec()`, `tip_style()`, `tip_table()`, `tip_template()`, `tip_theme()`, `use_hovercraft()`

Examples

```
tip_template(
  tip_title("{{ label }}"),
  tip_output("detail_plot")
)

tip_template(
  tip_title("{{ label }}"),
  tip_output("detail_table", keep_alive = FALSE)
)

# Retry: the render reads the retry input, so a click on Retry runs it again.
library(shiny)
ui <- fluidPage(
  tip_define(report = tip_template(tip_title("{{ label }}"), tip_output("report_text"))),
  textOutput("report_text"),
```

```
  tip_on(
    actionButton("run", "Report"),
    template = "report",
    record = list(label = "Report"),
    id = "run"
  )
)
server <- function(input, output, session) {
  tries <- 0
  output$report_text <- renderText({
    input$report_text_retry
    tries <<- tries + 1
    if (tries == 1) stop("The first try fails")
    "The report is ready"
  })
}
shinyApp(ui, server)

# One output for two items: the render reads the hint input to learn which item is open.
ui <- fluidPage(
  tip_define(detail = tip_template(tip_title("{{ label }}"), tip_output("detail_text"))),
  textOutput("detail_text"),
  tip_on(
    actionButton("a", "Item A"),
    template = "detail",
    record = list(label = "A"),
    id = "a",
    events = "hint"
  ),
  tip_on(
    actionButton("b", "Item B"),
    template = "detail",
    record = list(label = "B"),
    id = "b",
    events = "hint"
  )
)
server <- function(input, output, session) {
  open_item <- reactiveVal("none")
  observeEvent(input$a_hint, if (input$a_hint$state == "shown") open_item("A"))
  observeEvent(input$b_hint, if (input$b_hint$state == "shown") open_item("B"))
  output$detail_text <- renderText(paste("Details of item", open_item()))
}
shinyApp(ui, server)
```

`tip_presets` *The presets*

Description

[Stable] Lists the presets the tip engine ships, to give as `style =` to `tip_on()` and `use_hovercraft()`, or as the first argument of `tip_style()`. The name of a preset is stable; its values may be tuned between releases.

Usage

```
tip_presets()
```

Value

A data frame with one row for each preset and the columns `name`, `label` (a person's name for it) and `about` (one sentence on what it is for).

See Also

Other tooltips: `tip_blocks`, `tip_define()`, `tip_marker()`, `tip_more()`, `tip_on()`, `tip_output()`, `tip_server`, `tip_spec()`, `tip_style()`, `tip_table()`, `tip_template()`, `tip_theme()`, `use_hovercraft()`

Examples

```
tip_presets()
```

---

tip_server *Show, hide and change a tip from the server*

Description

[Experimental] `tip_show()` opens the tip of a tag, `tip_hide()` closes it, and `tip_update()` changes what the tip says, from a Shiny server function. The tag is found by the `id` given to `tip_on()`, or, when `tip_on()` had none, by the tag's own `id`. For a tip of a `tip_table()` give the table's `id` and the address of the tip: `row` alone is a row, `row` and `column` a cell, and `column` alone a header cell. For a tip on a tag, leave both out.

Usage

```
tip_show(
  id,
  ...,
  row = NULL,
  column = NULL,
  session = shiny::getDefaultReactiveDomain()
)

tip_hide(
  id,
  ...,
```

```
  row = NULL,
  column = NULL,
  session = shiny::getDefaultReactiveDomain()
)

tip_update(
  id,
  text = NULL,
  ...,
  template = NULL,
  record = NULL,
  placement = NULL,
  row = NULL,
  column = NULL,
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments

`id`	The tip's id: the `id` given to `tip_on()`, or, with `row` or `column`, the `id` given to `tip_table()`, without the module's namespace. One non-empty string with no whitespace and no `:`.
`...`	Must be empty. It makes a misspelt argument an error.
`row`, `column`	The address of a table's tip: `row` the row's position in the data, `column` a data column's name or position. Both `NULL` (the default) for a tip on a tag.
`session`	The Shiny session; the default is the one the call runs in.
`text`, `template`, `record`, `placement`	For `tip_update()`: the new values, as in `tip_on()`. `NULL` (the default) leaves a value as it is and `NA` takes it away; `""` clears the `text`. Give at least one. For a `template` that R has not seen in this process (it was defined in another session or process), `record` sends every column, since R cannot know which ones the template names: choose the columns yourself.

Details

`row` is the position of the row in the data the table has now, counting from 1: after `replaceData()` or a redraw with other data, row 3 is whatever is third. It is not a record's id, so do not keep it to find a record later. `column` is the data column, by its name in the data frame (not the header's words) or its position among the data columns, without the row-names column. A name is read from the table that `tip_table()` drew in this session, so the table has to have been drawn: a name that is not a column of a table already drawn stops with an error that lists the columns; a name given before the table's first draw is read when the call is sent, and if it is still not a column then, the call warns (class `hovercraft_warning_table`) and sends nothing. A position needs no table. A column that the widget hides has no tip. A row that is not on the table's current page has no element yet, in any paged table (`DT` in client or server mode, `reactable`), so nothing happens until it is on the page.

A call reaches only a tag that is on the page and drawn at that moment; nothing is queued, and a tag in a hidden tab opens nothing. On a page without Shiny (R Markdown, Quarto, a static page) the calls stop with an error. Every call is sent when the flush that follows it is done, and asks for that

flush, so `tip_show()` in the same observer as a `renderUI()` that draws the tag reaches the new tag, and a call from a callback of `later` is sent too.

A tip opened by `tip_show()` stays open until `tip_hide()` closes it, or until the reader opens another tip, presses Escape or clicks away. `tip_hide()` closes a hint, the tip that shows on hover or focus, and never a pin: when the tip of this id is not the hint that is open (another tip's hint is, or this one is pinned or closed), it does nothing. No server call pins a tip or closes a pin.

`tip_update()` rewrites the tag's tip attributes. It takes the values of `tip_on()` that change a tip (`text`, `template`, `record` and `placement`) and changes only what is named: `NULL` leaves a value as it is, `NA` takes it away, and for `text`, `""` clears the words. `template` is the name of a template or a `tip_template()`, which is sent with the call (not one that holds a `tip_output()`, which needs the tag's own `tip_on()`); `text` with `{{ field }}` placeholders is filled from `record`, as in `tip_on()`. An open tip draws its new words, template or record at once; a new `placement` is used the next time the tip opens. A `record` needs a `template`, or `text` with placeholders, in the same call, so that only the columns it names are sent to the page; to change the template alone, leave `record` out and the tag keeps the record it has. A `template` given without `text` also takes the tag's plain text away, since plain text beats a template. `text` and `template` in one call: the text beats the template, so the tip shows the words and the template is not used, and the words are plain text, with their braces as written; to fill `{{ field }}` placeholders from `record`, give the words alone. `tip_update()` leaves a `style` or `marker` alone, and it leaves the tag's `aria-label` as it is, so a button named from its tip keeps the old name; give such a button its own `aria-label`. The new tip lasts until the tag is drawn again, for example by `renderUI()`.

On a control that has `when_disabled`, `tip_update()` changes the control's own tip, also while the control is disabled. A disabled control still says only "Not available:" and its reason, in the new `placement`; the new `text`, `template` and `record` show once the control is enabled.

Inside a module, give the id as the module's own (`"save"`); the call adds the module's namespace, as the `update*()` functions of Shiny do. In the module's UI, give `tip_on()` the id as `ns("save")`.

Value

`NULL`, invisibly. These functions are called for their effect on the page.

See Also

Other tooltips: `tip_blocks`, `tip_define()`, `tip_marker()`, `tip_more()`, `tip_on()`, `tip_output()`, `tip_presets()`, `tip_spec()`, `tip_style()`, `tip_table()`, `tip_template()`, `tip_theme()`, `use_hovercraft()`

Examples

```
if (interactive() && requireNamespace("shiny", quietly = TRUE)) {
  ui <- shiny::fluidPage(
    tip_on(shiny::actionButton("run", "Run"), "Runs the **check**", id = "run_tip"),
    shiny::actionButton("show", "Show the tip"),
    shiny::actionButton("hide", "Hide it"),
    shiny::actionButton("change", "Change it")
  )
  server <- function(input, output, session) {
    shiny::observeEvent(input$show, tip_show("run_tip"))
    shiny::observeEvent(input$hide, tip_hide("run_tip"))
   shiny::observeEvent(input$change, tip_update("run_tip", text = "Runs the **new check**"))
```

```
  }
  shiny::shinyApp(ui, server)
}
```

`tip_spec` *A tip with settings of its own, for a table*

Description

[Experimental] Wraps the words of one `tip_table()` target (`rows`, one column of `cells`, or one column of `headers`) together with settings that only that target needs. A setting left as `NULL` is taken from `tip_table()`. The API shape may change in upcoming releases: later versions add more settings.

Usage

```
tip_spec(
  content,
  ...,
  style = NULL,
  placement = NULL,
  to = NULL,
  marker = NULL,
  more = NULL,
  pin = NULL,
  events = NULL
)
```

Arguments

`content`	The words of the tip (markdown, with `{{ column }}` placeholders for `rows` and `cells`), or a `tip_template()`. `NA` and `""` mean no tip.
`...`	Must be empty. It makes a misspelt argument an error.
`style`	How this target's tips look, or `NULL` to take the table's: whatever `style` takes in `tip_on()` (a preset name, a `tip_style()`, a `tip_theme()` or a `bslib::bs_theme()`). A row's own style also reaches the cells in that row that have none of their own, because the tip engine reads the nearest tagged element. A list of CSS variables (`tip_style(theme = list("--x" = ...))`) is for the whole page, so a spec refuses it with an error: give it to `use_hovercraft()`, or use the roles of `tip_theme()`.
`placement`	Where this tip shows: `"auto"`, `"top"`, `"bottom"`, `"left"` or `"right"`, or `NULL` to take the table's. A cell or header cell with no placement of its own follows the table's, not its row's.
`to`	**[Experimental]** Where this target's pinned tip is drawn: `NULL` or `"float"` (the default) for a tip that floats beside the row, or `"in_place"` for a panel under the row that holds what any pinned tip holds (its blocks, the long form, a

tip_output()). The hover tip still floats at placement. In place needs a pin (pin = TRUE), the rows target and the table's id (tip_table() says which is missing), and it opens by the pin handle or the pin key only. Any other string is an error. See tip_table() for what it does when the table sorts, filters and pages, and for what is not covered.

marker — The visible cue for this target's cells or header cells: "corner", "none" to take the marker off a column when the table has one, or NULL (the default) to take the table's. A spec for rows takes none: a row is not marked. See the marker argument of tip_table().

more — Extra content for the long form of this tip, or NULL (the default) for none: words (markdown, with {{ column }} placeholders for rows and cells), a tip_template(), or a tip_more() that also says when the long form shows. NA and "" mean no long form. Words and a template are the same as tip_more() with its default: a "More" button in the tip opens the long form, in a row, a cell and a header cell alike. tip_more(on = "pin") and tip_more(pin = TRUE) work only in the spec for rows of a tip_table() with pin = TRUE, the one target that can pin; anywhere else in a table they stop with an error. tip_more(on = "click") stops with an error in a table, where a click selects a row. tip_more(as = , where =) choose the look and side of the control, a button or a text link, as in tip_on(). In a table a cell does not inherit the long form of its row: a cell with no more of its own shows its short tip only, and a cell's own more replaces the row's.

pin — Whether a reader can keep this target's tip open: NULL (the default) or FALSE for no pin, or TRUE for a pin handle in the corner of the tip and a pin key, as in tip_on(). Only the tips of rows pin for now: a spec with a pin given to cells or headers stops with "only rows pin for now". A click on a row selects it in both widgets, so the "click" pin of tip_on() is refused in a table. A pin needs the table's id. With pin = TRUE, tip_more(on = "pin") opens the long form when the row is pinned.

events — The Shiny inputs this target's tips send: one or more of "hint", "pin" and "copy", or NULL (the default) for none. "pin" and "copy" belong to a pinned tip, so asking for either without a pin stops with an error, as in tip_on(). Only the tips of rows send them for now ("only rows send events for now"), and they need the table's id. The inputs belong to the table, one per event, not one per row: see the "Shiny inputs" section of tip_table().

Details

Give a spec as rows, or as an entry of a named list given to cells or headers. Plain words and a tip_template() are accepted in all three places too, and mean "the table's settings".

Value

An object of class hovercraft_spec, to give to tip_table().

See Also

Other tooltips: tip_blocks, tip_define(), tip_marker(), tip_more(), tip_on(), tip_output(), tip_presets(), tip_server, tip_style(), tip_table(), tip_template(), tip_theme(), use_hovercraft()

Examples

```
scores <- data.frame(team = c("Owls", "Foxes"), wins = c(7, 5), note = c("Top", "Late"))
DT::datatable(scores) |>
  tip_table(
    rows = "**{{ team }}**: {{ wins | number:0 }} wins",
    cells = list(
      note = tip_spec(
        "{{ team }}",
        more = tip_more("{{ note }}", on = "hover"),
        style = "sober",
        placement = "right",
        marker = "corner"
      )
    )
  )

# A row that can be pinned, and a Shiny input that says which row was pinned
DT::datatable(scores) |>
  tip_table(
   rows = tip_spec("**{{ team }}**: {{ wins | number:0 }} wins", pin = TRUE, events = "pin"),
    id = "scores_tips"
  )
```

---

tip_style *How the tips look*

Description

[Experimental] Bundles what makes a tip look the way it does: a preset (a named look), a theme (colours, corner and font), a class on the tip box, and how it enters and how big it is. Give the result as `style =` to `tip_on()` for one tag, or to `use_hovercraft()` for the whole page. A part left out is left alone: it is inherited from the page, or from the page's own bslib variables.

Usage

```
tip_style(
  preset = NULL,
  ...,
  theme = NULL,
  class = NULL,
  motion = NULL,
  gap = NULL,
  max_width = NULL,
  pin_width = NULL
)
```

Arguments

preset	A preset name from tip_presets(), "none" to clear an inherited preset, or NULL (the default) to inherit. A string with : or ; is not a preset: style is not CSS.
...	Must be empty. It makes a misspelt argument an error.
theme	A tip_theme(), a bslib::bs_theme(), a named list of roles (list(bg = "#1b3a6b")), a named list of CSS variables, or NULL (the default).
class	Class names for the tip box, "" to clear an inherited class, or NULL (the default) to inherit. See the section on the class.
motion	How a tip enters: "fade", "slide" or "none", or NULL (the default) to inherit. A page that sets no motion has none. See the section on size and motion.
gap	The space between the tag and the tip, in pixels: a whole number from 0 to 24, or NULL (the default) to inherit.
max_width	The widest a hint grows, in pixels: a whole number from 160 to 640, or NULL (the default) to inherit. It is also kept inside the window.
pin_width	The widest a pinned tip grows, in pixels: a whole number from 240 to 720, or NULL (the default) to inherit.

Details

A value set nearer the tip wins: a tag's style beats the page's. A theme is merged role by role, so a tag that sets only bg keeps the page's text. A preset is replaced as a whole by the nearer one, and "none" clears an inherited preset. A theme beats a preset set at the same place or further out, role by role, so tip_style("sober", theme = ...) is sober with those roles changed; a preset set nearer the tip than a theme beats that theme.

Value

A hovercraft_style object, to give as style = to tip_on() or use_hovercraft(). Its fields are internal; pass the object on.

The class

class puts class names on the tip box, so your own CSS can restyle it. Give class names as a vector (c("wide", "audit-tip")) or as text with spaces ("wide audit-tip"), "" to clear a class the page or a parent set, or NULL to inherit. A name has letters, digits, - and _ and does not start with a digit, and must not start with hc-tip, which the tip's own classes use; R stops on a bad name, where the browser would only drop it with a line in its console. The nearest class wins as a whole and the classes are not added together: a tag's class = "wide" replaces a page's class = "brand".

Size and motion

motion is how a tip enters: "fade", "slide" (it moves four pixels from its side while it fades) or "none". gap is the space between the tag and the tip, max_width the widest a hint grows, and pin_width the widest a pinned tip grows, all in pixels and all whole numbers. R checks them against the ranges in the tip engine's page contract, so a value outside one stops with an error that names it. A tip is also kept inside the window.

They are written the way a theme is, on the page or on one tag, and a value set nearer the tip wins, one value at a time: a tag that sets only `gap` keeps the page's `motion`. They apply with a preset or without one. A reader's reduced-motion setting always wins over `"fade"` and `"slide"`; the tip engine does this, and nothing here switches it off.

There is no `delay` argument: the tip engine reads the delay before a hint shows only from the file of a preset.

Themes on one tag

A theme given to a tag sets the roles `bg`, `text`, `border`, `code_bg`, `radius`, `shadow` and `font`. CSS variables given as a list of names that start with `--` reach the tip only from the page, not from a tag, because a tip is drawn away from its tag.

See Also

Other tooltips: `tip_blocks`, `tip_define()`, `tip_marker()`, `tip_more()`, `tip_on()`, `tip_output()`, `tip_presets()`, `tip_server`, `tip_spec()`, `tip_table()`, `tip_template()`, `tip_theme()`, `use_hovercraft()`

Examples

```
tip_style("sober")
tip_style("sober", theme = tip_theme(light = list(bg = "#1b3a6b", text = "#ffffff")))
tip_style(class = "wide")
tip_style(class = "") # clears a class set above
tip_style("sober", motion = "slide", max_width = 280)
use_hovercraft(style = tip_style(motion = "fade", gap = 10))
tip_on(
  htmltools::tags$button("Run"),
  "Runs the **check**",
  style = tip_style("dense", class = "audit-tip")
)
```

`tip_table` *Put tips on the rows, cells and headers of a table*

Description

[Experimental] Built on top of the tip engine, and experimental: the API may change in upcoming releases. Gives the rows of a table made by `DT::datatable()` or `reactable::reactable()` a tip drawn from the row's own values, and optionally a tip of their own to chosen cells and header cells. The tips follow the data row, so sorting, filtering and paging need no code of yours: each row is its own record, whatever place it has on the screen.

Usage

```
tip_table(
  widget,
  ...,
  rows = NULL,
  cells = NULL,
  headers = NULL,
  style = NULL,
  placement = NULL,
  marker = NULL,
  id = NULL,
  max_pins = NULL
)
```

Arguments

widget	A table made by DT::datatable() or by reactable::reactable().
...	Must be empty. It makes a misspelt argument an error.
rows	One string, tip_template() or tip_spec() for every row, or NULL. NA and "" are no tip. A tip_spec() here may also set pin and events. A tip_template() is given as an object: the name of a template made with tip_define() stops with an error that says so.
cells	A named vector or list, one entry per column (by the column's name; a name given twice is an error), for the cells of that column. Each entry is a string, a tip_template() or a tip_spec(); NA and "" give that column no tip. Use a list to mix them: list(status = tip_spec("{{ note }}", placement = "right"), wins = "{{ wins }} wins").
headers	The same for the header cells; a header has no row to read from, so a tip_spec() here holds words or a tip_template() with no placeholders.
style	How every tip in the table looks, or NULL (the default) to take the page's: whatever style takes in tip_on() (a preset name, a tip_style(), a tip_theme() or a bslib::bs_theme()). A tip_spec() with its own style beats it for that target. A row's own style also reaches the cells in that row that have none of their own, because the tip engine reads the nearest tagged element. A list of CSS variables (tip_style(theme = list("--x" = ...))) is for the whole page, so a table refuses it with an error: give it to use_hovercraft(), or use the roles of tip_theme().
placement	Where the tips show: "auto", "top", "bottom", "left" or "right", or NULL for the page's choice. A tip_spec() with its own placement beats it. A cell's tip follows this, not its row's.
marker	The visible cue on every cell and header cell that has a tip, so a reader can see where the tips are: "corner" (a small triangle in the top corner, as a spreadsheet shows a note), or NULL or "none" for no marker (the default). A tip_spec() with its own marker beats it. A row takes no marker, and a table draws only the corner for now: "underline", "icon" and an icon of your own stop with an error. A cell with a marker is highlighted while its tip is open. See tip_marker().

id — The id this table's Shiny inputs hang on (`input$<id>_pin`), and the base of its tips' ids: one non-empty string with no whitespace and no `:`. It is the table's id, not one tip's. Inside `renderDT()` or `renderReactable()` in a module's server function, the plain id is enough: `tip_table()` adds the module's prefix from the session the table is drawn in, as `tip_show()` does, so `id = "scores"` in module `"mod1"` is `input[["mod1-scores_pin"]]`. In a module's UI function, or anywhere else with no session (a static page), nothing adds the prefix: give `ns("scores")` there. An id that already has the prefix (`session$ns("scores")`) is left alone. Give it a different id from every other tip source on the page: two tables with one id make the page warn in the browser console. With an `id`, every marked row, cell and header cell gets its own tip id derived from it, which follows the data row through sorting, filtering and paging; how that id is built is not part of the API: `tip_show()`, `tip_hide()` and `tip_update()` take the table's `id` with `row` and `column` instead. `NULL` gives the tips no id. A pin or events given to `rows` (see `tip_spec()`) need an `id`.

max_pins — **[Experimental]** How many of this table's rows can be pinned at once, a whole number from 1 to 10, or `NULL` (the default). `NULL` counts the table's pins in the app's limit (`use_hovercraft(max_pins =)`): a row pin can close a pin elsewhere, and the other way round. A number gives the table its own count, outside the app's: a row pinned past it closes the table's oldest pin and no other, and a pin elsewhere never closes one of the table's rows. It needs a `rows` spec that pins (`tip_spec()` with `pin = TRUE`) and the table's `id`. All the pins open together, the app's and every table's, cannot pass 10, the engine's cap. On a viewport narrower than 600 pixels the engine keeps one pin at a time, whatever the counts say.

Details

Known limits: `row` is the position in the data the table has now, so a pin follows the row number after `replaceData()` or a redraw; a `DT` table in server mode has only the rows of the current page on the screen; and the table's script does some work for each row on the screen on every change of the data, so page a long table. `tip_table()` reads the table's data once in R (for a `reactable` table it parses all the rows) to check the columns and values the tips name, so its R time grows with the rows.

A cell's tip wins over its row's, and a row's tip opens at the pointer. The tip of a cell or a header cell points at its text, not at the whole cell, so a right-aligned number in a wide column, or a right-aligned header, gets a tip beside the text and not far to its left; there is no argument for it. A target that needs its own settings (style, placement, marker or `more`) takes a `tip_spec()` in place of words. A tip is made of words (markdown) with `{{ column }}` placeholders, or of a `tip_template()`; every placeholder is checked against the table's columns, and a miss lists them. Placeholders that start with a dot are kept for hovercraft. A header tip has no row to read from, so it takes words, or a `tip_template()` with no placeholders.

Tips also show for the keyboard. A row, a cell or a header cell cannot take focus by itself, so `tip_table()` gives the table one tab stop, and the arrow keys move between the rows, cells and header cells that have a tip: left and right inside a row (the row first, then its cells), up and down between rows to the same column, Home and End to the ends of the row, and Ctrl with Home or End to the ends of the table. The tip shows for what has focus. A header that its widget already

makes focusable (the sortable headers of DT) keeps its own tab stop. Enter and Space do not select a row. A click on a row selects it in both widgets, so a row's tip does not pin on click.

Pins: with `tip_spec(pin = TRUE)` as `rows`, a reader can keep a row's tip open. The tip that shows on hover or focus gets a small pin handle in its corner, the same as `tip_on()`'s (`pin = TRUE`): a click or tap on it pins the row's tip, and the tip's own close button, `Escape` or a click away closes it (a click away does not close a panel that opens in place, see below). The pin key `P` pins the tip of the row that has focus or that the pointer rests on, or of the row a focused cell belongs to; a cell or a header cell has no pin of its own ("only rows pin for now"), so `P` while the pointer rests on a cell that has a tip of its own pins nothing. A long form (`more`) opens with its "More" button, or with the `M` key when the focus is on the row, a cell or a header cell, or the pointer rests on it. A tap on a row shows its tip and a tap on the handle pins it. The handle is not a column and not a tab stop, so the table keeps its one tab stop and its columns; keyboard readers use the key, which each row that can pin announces as `aria-keyshortcuts`. A pin follows its row. It stays on a row when `DT` sorts (client and server mode), and on the same row number when `DT` draws the table again or `DT::replaceData()` or `reactable::updateReactable(data = )` give it new data, with the new values in its tip. It closes when its row leaves the page (a filter or another page in `DT`, another page in `reactable`), when `reactable` shows another row in the row's place, when the table goes, and when a click lands elsewhere. The input then says why: `DT` closes a pin of a row that left the page itself (`reason "gone"`, `by "data"`); `reactable` keeps its row nodes and gives them other rows, so the layer closes the pin through the tip engine's host command (`reason "host"`, `by "host"`, `via "command"`). A server that acts on `by == "data"` sees the two widgets differently.

Value

The widget, with the tips added.

Shiny inputs

`tip_spec(events = )` on `rows` turns on inputs that belong to the table, one per event and not one per row, named after the table's `id`: `input$<id>_hint`, `input$<id>_pin` and `input$<id>_copy`. Each is `NULL` until its first event. Its value is the event that just happened, with the fields of `tip_on()`'s (`state`, `text`, `by`, `via` and `reason`), and says which tip it was:

- `target`: `"row"` (`"cell"` and `"header"` are kept for later; only rows send events for now);
- `row`: the row's place in the data the widget has now, counted from 1, in client mode and in server mode alike, and not the row's place on the screen: after a sort, the third data row is still 3. It is `NULL` for a header. After `DT::replaceData()`, `reactable::updateReactable(data = )` or a re-render with other data, row 3 is whatever is third now, so do not keep it as a record's id;
- `column`: the data column's name (the name in the data frame, not the header's text), or `NULL` for a row;
- `id`: the tip's id. How it is written is not part of the API.

`input$<id>_pinned` (with `events = "pin"`) is what is pinned now: a list of such addresses (`target`, `row`, `column`, `id`), the oldest pin first. It is `NULL` until the first pin event, and an empty list once nothing is pinned. It is sent in the same batch as `_pin`, and a value equal to the one before does not fire an observer again. Read it with `length(input$scores_pinned)`: it holds as many addresses as the table has pins open (see `max_pins`). For a tag of `tip_on()` with `events = "pin"`, `input$<id>_pinned` is `TRUE` or `FALSE` (read it with `isTRUE(input$run_pinned)`).

input$<id>_pin and input$<id>_hint are told apart by state, whose values may grow in a later version. A pin's state is "pinned", "closed" or "updated" (the tip is pinned and was drawn again because its words changed: an app that only tests for "closed" should not take it for a pin); a hint's is "shown" or "hidden". The values of by are "reader", "data" (the page changed under the tip) and "host" (a command of the page did it). The values of reason (the same for the inputs of tip_on()) are, for a pin, "select" (the same element again), "escape", "button" (the tip's close button), "outside" (a click away), "replaced" (another tip was pinned), "gone" (the row left the page), "below" (a layer around it closed), "host" (a command of the page closed it) and "redraw" (the tip was drawn again with other host slots, by = "data"); for a hint, "leave" (the pointer left), "blur" (the focus left), "escape", "outside", "pinned" (the hint became a pin), "replaced", "gone" and "host". reason is NULL when a tip opens.

In place

[Experimental] With tip_spec(to = "in_place", pin = TRUE) as rows (and the table's id), a row's pinned tip is drawn as a panel under its row, as wide as the table's visible part, in DT and in reactable. The hover tip still floats. The pin handle or the pin key opens it, and it holds what any pinned tip holds: its blocks, the long form with its "More" button, a tip_output(). The widget's own inputs (rows_selected, cell_clicked) are not touched by a click in it.

It follows the pin: DT moves the panel with its row when the table sorts, and puts it back after every draw, in server mode too; a filter or another page that takes the row off the page closes it (reason "gone"). DT::replaceData() keeps it on the same row number. reactable closes it when it shows other rows in the row's node (reason "host"), as for a floating pin, and keeps it when it is given new data with updateReactable(data =). A server-mode DT drawn again by renderDT() closes it, as it does a floating pin. A panel counts in use_hovercraft(max_pins =) and tip_table(max_pins =) like any pin: the oldest closes (reason "replaced") and its panel goes with it.

A click away does not close a panel: it closes by its close button, Escape, the pin limit, or its row going. The pin key on a row whose panel is open does nothing. Escape on a row that has no pin of its own closes the newest pin, as for floating pins. With scrollX the panel stays at the left edge of the visible part. The FixedColumns extension of DT (it copies the rows) stops with an error; responsive and RowGroup of DT, reactable's groupBy, and screen readers are not covered. A click inside a DT panel is stopped at the panel's row, so a handler that the page delegates on document does not see it.

What works, each checked in a browser:

- DT, in client mode and in server mode under Shiny, with scrollX, scrollY, fixedHeader, colReorder and responsive. With responsive, a column folded into the details row has no tip there; the row's tip is unchanged.
- reactable, with sorting, filtering, paging, row selection (a click selects the row and its tip stays), groupBy (the rows inside a group have tips, the group's own row has none), details (the details panel has none) and filterable = TRUE (the tip of a header cell stays on the header cell; the filter box under it has no tip and is not described twice). DT's filter row has none either.
- An id inside a module: give the plain id inside renderDT() or renderReactable(), and ns() in the module's UI function (see id).

- Tables that Shiny updates with new data: `DT::replaceData()` (it works in server mode) with the user's own data frame, and `reactable::updateReactable(data = )`. The tips follow the new rows, as they do after sorting, filtering and paging.

Not supported: a `DT` table with its own `createdRow` or `initComplete` option; a `reactable` column whose `headerClass` is a JavaScript function; any other widget.

Large tables: `tip_table()` adds nothing to the data of the page, since each tip reads its values from the table's own data in the browser. A `reactable` row carries the values its tips read in its class, as well as in the engine's attribute, so they sit twice in the page for every row that is drawn: page the table. A table with every row drawn at once is slow in `DT` when cells or headers have tips (a table of 2,000 rows with a row tip and three cell tips took about a second to draw, five times as long as without tips; thousands of rows take seconds); page it, or tip its rows only.

Give `tip_table()` the finished widget, last in the pipe or at least after the columns are settled. `DT::datatable()` keeps its columns and its inputs as they are: `tip_table()` adds no column and changes no event (a table with its own `createdRow` or `initComplete` option is refused for now). `reactable::reactable()` gets its own `rowClass`, `class` and `headerClass` added to yours; the class of a row holds the values its tips read, so it changes with the data.

See Also

Other tooltips: `tip_blocks`, `tip_define()`, `tip_marker()`, `tip_more()`, `tip_on()`, `tip_output()`, `tip_presets()`, `tip_server`, `tip_spec()`, `tip_style()`, `tip_template()`, `tip_theme()`, `use_hovercraft()`

Examples

```
scores <- data.frame(team = c("Owls", "Foxes"), wins = c(7, 5), losses = c(3, 5))
DT::datatable(scores) |>
  tip_table(
    rows = tip_spec(
      "**{{ team }}**: {{ wins | number:0 }} wins, {{ losses | number:0 }} losses",
      pin = TRUE,
      events = "pin"
    ),
    headers = c(wins = "Games won this season"),
    id = "scores_tips"
  )
```

`tip_template` *A template: the blocks a tip is drawn from*

Description

[Stable] A template is a list of blocks, shown in order. Name it with `tip_define()` and use the name in `tip_on()`, or give the template itself to `template =` of `tip_on()`. A block can be made with `tip_title()` and the other block functions, or written as a plain list such as `list(type = "text", text = "...")`; both are checked in the same way.

Usage

```
tip_template(..., head = NULL, body = NULL, foot = NULL)
```

Arguments

`...`	One or more blocks in order (cannot be named; use `head`, `body` and `foot` for slots).
`head`	Optional block or list of blocks drawn at the top of the tip (for example a `tip_title()` and a `tip_badge()`).
`body`	Optional block or list of blocks drawn in the middle of the tip. Cannot be combined with `...`.
`foot`	Optional block or list of blocks drawn at the bottom of the tip below a subtle divider.

Details

Blocks can be given in order through `...`, or placed into the named slots `head`, `body` and `foot`. Each slot takes one block or a list of blocks; `foot` is drawn below a thin divider in muted text, suited for a source line or footnote. Do not pass both positional blocks in `...` and `body`.

Value

A template, to give to `tip_define()` or to `template =` of `tip_on()`.

See Also

Other tooltips: `tip_blocks`, `tip_define()`, `tip_marker()`, `tip_more()`, `tip_on()`, `tip_output()`, `tip_presets()`, `tip_server`, `tip_spec()`, `tip_style()`, `tip_table()`, `tip_theme()`, `use_hovercraft()`

Examples

```
tip_template(
  tip_title("{{ label }}"),
  tip_text("{{ differences | number:0 }} differences"),
  list(type = "badge", text = "{{ status }}")
)

tip_template(
  head = list(tip_title("{{ label }}"), tip_badge("{{ status }}", status = "{{ tone }}")),
  body = tip_fields(Records = "{{ records | number:0 }}"),
  foot = tip_text("Source: {{ source }}")
)
```

`tip_theme` *A theme for the tips*

Description

[Experimental] Sets how the tips look: the fill, words, border and code background, the corner, the shadow and the typeface, for light and for dark. A role left out keeps what the page gives: the tip reads Bootstrap's variables where the page has them (bslib), so a page that already has a `bslib::bs_theme()` needs no theme, and its tips follow light and dark on their own. Use a theme when the tips must look different from the page, or when the page has no bslib. Give the result to `use_hovercraft()` once, in the page's UI, or to `tip_on()` for one tag, as `style =`.

Usage

```
tip_theme(..., light = NULL, dark = NULL, mode = "auto")
```

Arguments

`...`	Must be empty. It makes a misspelt argument an error.
`light, dark`	Named lists of roles, or `NULL`. Roles: `bg`, `text`, `border`, `code_bg`, `radius`, `shadow`, `font`. `radius` is a number of pixels or a length such as `"0.5rem"`; `shadow` is CSS `box-shadow` text or `"none"`; `font` is CSS `font-family` text. Values may only hold letters, digits, spaces and `, . ' " ( ) # % _ + * -`.
`mode`	`"auto"` (the default) follows the page: the `data-bs-theme` attribute that `input_dark_mode()` in bslib sets on the page, and with none, the visitor's own setting. `"light"` and `"dark"` use that mode whatever the page says. A tip is drawn at the page level, so a `data-bs-theme` on a smaller part of the page does not reach it.

Details

The roles are `bg` (the tip's fill), `text` (its words), `border` (its border, seen in forced-colours mode and in a tip that stays open), `code_bg` (behind code), `radius` (the corner), `shadow` and `font`. Colours may be names, hex, `grDevices::rgb()` output, `"rgba(0, 0, 0, 0.8)"` text or three numbers from 0 to 255. A value that is a CSS variable, `"--team-blue"` or `"var(--team-blue)"`, is read by the browser where the page defines it; R cannot check it. A theme can also be a `bslib::bs_theme()` or a plain list, see `tip_style()`.

A role set in `light` and not in `dark` is used in dark mode too, so a brand colour set once holds in both modes.

The tip has a small arrow pointing at its tag. It takes the colour of the tip, so it is drawn only when the `bg` is opaque: with a translucent `bg` (`"rgba(0, 0, 0, 0.8)"`) the tip has no arrow.

Value

A `hovercraft_theme` object, to give as `style =` to `use_hovercraft()` or `tip_on()`, or to `tip_style()`. Its fields are internal; pass the object on.

Contrast warnings

The words of a tip must be readable on its fill. For each mode, the `text` on the `bg` (and on the `code_bg`, if you give one) is measured, and a ratio below 4.5:1, the usual level for text, gives a warning of class `hovercraft_contrast`; the colours are kept. A role you leave out is taken to be what Bootstrap's default theme gives, a shade of the page: Bootstrap's tertiary background with the page's text colour (a `#f8f8f8` fill with `#1d1f21` words in light mode, `#292d34` with `#dee2e6` in dark mode). CSS variables cannot be measured and are skipped. A translucent `bg` is measured over the page's colour (white, or `#1d1f21` in dark mode), and a translucent `code_bg` over the `bg`. Set `options(hovercraft.contrast_warnings = FALSE)` to turn the warnings off, or silence only them:

```
withCallingHandlers(
  tip_theme(light = list(bg = "#ffffff", text = "#dddddd")),
  hovercraft_contrast = function(w) invokeRestart("muffleWarning")
)
```

See Also

Other tooltips: `tip_blocks`, `tip_define()`, `tip_marker()`, `tip_more()`, `tip_on()`, `tip_output()`, `tip_presets()`, `tip_server`, `tip_spec()`, `tip_style()`, `tip_table()`, `tip_template()`, `use_hovercraft()`

Examples

```
tip_theme()
tip_theme(light = list(bg = "#1b3a6b", text = "#ffffff", radius = 8))
tip_theme(
  light = list(bg = "#1b3a6b", text = "#ffffff"),
  dark = list(bg = "#9ec5fe", text = "#0b1f3a"),
  mode = "auto"
)
tip_theme(light = list(bg = "--brand-blue", font = "Georgia, serif"))
```

`use_hovercraft` *The tip engine as an HTML dependency*

Description

[Stable] for the page's `style` given as a preset name; `max_pins`, `keys` and `labels` are **[Experimental]**. The engine's script and CSS, ready to add to a page. `tip_on()` and `tip_define()` attach it for you; use this when you build the page's head yourself, or to give the page a look. The dependency is versioned with the engine, not with this package. The engine reads Bootstrap's colours where the page has them (bslib), and plain colours otherwise. Which names on the page your own CSS and JavaScript may rely on, and which are internal, is in the section "What is public on the page" of the README.

Usage

```
use_hovercraft(style = NULL, max_pins = 1, keys = TRUE, labels = NULL)
```

Arguments

`style` **[Stable]** as a preset name, **[Experimental]** as a style object. How the tips look on the whole page, or `NULL` (the default) to take the page's own bslib variables, so a page with a `bslib::bs_theme()` follows it, light and dark, with nothing more. One of:

- a preset name from `tip_presets()`, such as `"sober"`;
- a `tip_style()`, which bundles a preset, a theme, a class for the tip box, and its motion and size;
- one part of a style, read as `tip_style(<part>)`: a `tip_theme()`, or a `bslib::bs_theme()`, read through its variables (bslib 0.6.0 or later). The tip is a shade of the page: its fill is the theme's tertiary background and its words the theme's text colour, with the border colour, corner and font.

A tag's own `style` (see `tip_on()`) beats the page's, and a value set nearer wins, value by value. A named list is not taken: give `tip_theme()` the roles, or `tip_style(theme = list(...))` a list of roles or of CSS variables such as `list("--hovercraft-tip-bg" = "#1b3a6b")`. A string that holds `:` or `;` is not a style: it looks like CSS, and `style` takes no CSS.

`max_pins` **[Experimental]** How many pinned tips the app keeps open, a whole number from 1 to 10; the default, `1`, is one at a time. At the limit the oldest pin closes when a new one opens. Give it once, in the app's UI: a second `use_hovercraft()` with another number is seen by the page, which warns in the browser's console and keeps the first (a call that leaves `max_pins` out says nothing, so give it explicitly, even `1`, to be seen). A `tip_table()` with its own `max_pins` has its own count, outside this one. All the pins open together, the app's and the tables', cannot pass 10, the engine's cap. On a viewport narrower than 600 pixels the engine shows a pin as a bottom sheet and keeps one at a time, so `max_pins = 3` means one there.

`keys` **[Experimental]** The letter keys of a tip: `TRUE` (the default) for all, `FALSE` for none, or a named logical vector with one switch per key, `c(pin = FALSE)` or `c(more = FALSE)`; a key left out stays on. `pin` is the key that pins the tagged item (P), which works while the focus is on the item or the pointer rests on it, and never inside a field; `more` is the key that opens a tip's long form (M), which `tip_more()` adds. Both work from a hover, so this is the page's switch: it turns a key off for every reader, and there is no switch for one reader. Escape always closes a tip. Switching `pin` off takes the key's announcement (`aria-keyshortcuts`) off the items too; the pin button stays. Give it once: a second `use_hovercraft(keys = )` that differs warns in the console and the first holds.

`labels` **[Experimental]** The words the tips show themselves on the whole page, or `NULL` (the default) for the built-in ones. A named character vector, such as `c(copy = "Copy code", close = "Dismiss")`; a word left out keeps its built-in one. The keys, with their built-in words, are `copy` (`Copy`), `copied` (`Copied`), `copy_failed` (`Copy failed`), `close` (`Close`), `not_available` (`Not available:`), `pinned` (`{label}: details pinned. Tab to reach them.`), `code_line` (`code: {first}`), `code_line_empty` (`code, 1 line`), `code_lines` (`code, {n} lines, starting {first}`), `code_lines_empty` (`code, {n} lines`), `more` (`More`), `more_pinned` (`More when pinned`),

less (Less), pin (Pin this tip). Four keys take placeholders, which the page fills in: pinned takes {label} (the name of the item), code_line takes {first} (the first line of the code) and code_lines and code_lines_empty take {n} (the number of lines; code_lines takes {first} too); other braces stay as written. An unknown key, an empty or blank word, or a value that is not words stops with an error; the unknown key's error lists the keys. A tag's own labels (see tip_on()) beat the page's, key by key, and tip_more(label =) beats both for the buttons of one tip. Unlike keys, where the first call holds, a second use_hovercraft(labels =) adds its words to the first, and a key that both give takes the later one. Useful for a page in another language.

Value

A list of htmltools::htmlDependency() objects: the engine's script and style sheet, then, with a style that sets something, hovercraft-style, which holds the page's theme CSS and sets the page's preset and class as the engine's attributes on the page, with a max_pins, keys or labels that is given one of their own, and last hovercraft-runtime, which starts the engine and the layers (the names of these dependencies are internal and may change). Put it once in the page's UI, for example bslib::page_fluid(use_hovercraft(style = "sober"), ...).

Nothing in them is inline script or style: they are files and data, so a page served with Content-Security-Policy: script-s works. A style sheet that is made from a theme is data too, which a script applies as a constructed style sheet; it comes after the page's own style sheets in the cascade. The scripts read their names and tip_define()'s templates from JSON, which a browser does not run, and set styles through the browser's style object, which a policy on inline markup does not cover. Every dependency is a folder of the installed package, so a cached document or a shared cache can use them in another R process.

What such a page still needs, because it is not hovercraft's markup:

- a policy that has default-src or img-src must also allow data: images, or three small icons are not drawn: the lock before a reason (when_disabled), the pin button (pin) and the built-in icon of marker = "icon". The words of the first two still show; the marker has no words, so an icon marker is then invisible (an icon of your own, given as a tag, is not affected);
- a style attribute written in markup, such as the width that shiny::actionButton() and shiny::plotOutput() write, is blocked by style-src 'self' whoever writes it. style-src-attr 'unsafe-inline allows them, and nothing else, or the page can use classes and a style sheet instead.

See Also

Other tooltips: tip_blocks, tip_define(), tip_marker(), tip_more(), tip_on(), tip_output(), tip_presets(), tip_server, tip_spec(), tip_style(), tip_table(), tip_template(), tip_theme()

Examples

```
use_hovercraft()
use_hovercraft(style = "modern")
use_hovercraft(style = tip_theme(light = list(bg = "#1b3a6b", text = "#ffffff")))
use_hovercraft(style = tip_style("sober", theme = list("--hovercraft-tip-radius" = "8px")))
use_hovercraft(max_pins = 3, keys = c(pin = FALSE))
use_hovercraft(
```

```
  labels = c(copy = "Copy code", close = "Dismiss", more_pinned = "Open to read more")
)
```

Index

∗ **tooltips**
    tip_blocks, 3
    tip_define, 5
    tip_marker, 6
    tip_more, 8
    tip_on, 10
    tip_output, 18
    tip_presets, 20
    tip_server, 21
    tip_spec, 24
    tip_style, 26
    tip_table, 28
    tip_template, 33
    tip_theme, 35
    use_hovercraft, 36

bslib::bs_theme(), *12*, *24*, *27*, *29*, *35*, *37*
bslib::popover(), *11*, *16*
bslib::tooltip(), *11*, *16*
bslib::value_box(), *7*

grDevices::rgb(), *35*

hovercraft_book, 2
htmltools::htmlDependency(), *38*

paste0(), *11*

shiny::actionButton(), *38*
shiny::plotOutput(), *38*
shiny::runApp(), *3*
sprintf(), *11*

tip_badge *(*tip_blocks*)*, 3
tip_badge(), *34*
tip_blocks, 3, *5*, *7*, *10*, *16*, *19*, *21*, *23*, *25*, *28*, *33*, *34*, *36*, *38*
tip_code *(*tip_blocks*)*, 3
tip_define, 5
tip_define(), *4*, *7*, *10–12*, *16*, *18*, *19*, *21*, *23*, *25*, *28*, *29*, *33*, *34*, *36*, *38*
tip_fields *(*tip_blocks*)*, 3
tip_hide *(*tip_server*)*, 21
tip_hide(), *12*, *30*
tip_marker, 6
tip_marker(), *4*, *5*, *10*, *12*, *16*, *19*, *21*, *23*, *25*, *28*, *29*, *33*, *34*, *36*, *38*
tip_more, 8
tip_more(), *4*, *5*, *7*, *14*, *16*, *19*, *21*, *23*, *25*, *28*, *33*, *34*, *36*, *38*
tip_on, 10
tip_on(), *4–10*, *19*, *21–29*, *31–38*
tip_output, 18
tip_output(), *4*, *5*, *7*, *10*, *16*, *21*, *23*, *25*, *28*, *32–34*, *36*, *38*
tip_presets, 20
tip_presets(), *4*, *5*, *7*, *10*, *12*, *16*, *19*, *23*, *25*, *27*, *28*, *33*, *34*, *36–38*
tip_server, *4*, *5*, *7*, *10*, *16*, *19*, 21, *21*, *25*, *28*, *33*, *34*, *36*, *38*
tip_show *(*tip_server*)*, 21
tip_show(), *12*, *13*, *30*
tip_spec, 24
tip_spec(), *4–10*, *16*, *19*, *21*, *23*, *28–30*, *33*, *34*, *36*, *38*
tip_style, 26
tip_style(), *4–7*, *10*, *12*, *16*, *19*, *21*, *23–25*, *29*, *33–38*
tip_table, 28
tip_table(), *4–10*, *12*, *15*, *16*, *19*, *21–25*, *28*, *34*, *36–38*
tip_template, 33
tip_template(), *4*, *5*, *7*, *8*, *10*, *11*, *14*, *16*, *18*, *19*, *21*, *23–25*, *28–30*, *33*, *36*, *38*
tip_text *(*tip_blocks*)*, 3
tip_theme, 35
tip_theme(), *4*, *5*, *7*, *10*, *12*, *16*, *19*, *21*, *23–25*, *27–29*, *33*, *34*, *37*, *38*
tip_title *(*tip_blocks*)*, 3
tip_title(), *33*, *34*

tip_update (tip_server), 21
tip_update(), *12, 13*, *30*

use_hovercraft, 36
use_hovercraft(), *4, 5*, *7, 8*, *10*, *12*, *15, 16*, *19*, *21*, *23–29*, *33–36*