Package ‘constructive’

June 6, 2023

Title Display Idiomatic Code to Construct Most R Objects
Version 0.1.0
Description Prints code that can be used to recreate R objects. In a sense it is similar to ‘base::dput()’ or ‘base::deparse()’ but ‘constructive’ strives to use idiomatic constructors.
License MIT + file LICENSE
Encoding UTF-8
RoxygenNote 7.2.3
Imports cli, diffobj, ellipsis, rlang (>= 1.0.0), roxygen2, styler, waldo, methods
Suggests dm, pixarfilms, testthat (>= 3.0.0), dplyr, data.table, vctrs, tibble, prettycode, tidyselect, lubridate, forcats, knitr, rmarkdown, sf, ggplot2, scales
URL https://github.com/cynkra/constructive,
https://cynkra.github.io/constructive/
BugReports https://github.com/cynkra/constructive/issues
Config/testthat/edition 3
VignetteBuilder knitr
NeedsCompilation yes
Author Antoine Fabri [aut, cre]
Maintainer Antoine Fabri <antoine.fabri@gmail.com>
Repository CRAN
Date/Publication 2023-06-06 07:40:09 UTC

R topics documented:

.cstr_apply .......................................................... 3
.cstr_combine_errors ............................................. 4
.cstr_construct .................................................... 6
.cstr_fetch_opts .................................................... 6
## R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>.cstr_match_constructor</td>
<td>7</td>
</tr>
<tr>
<td>.cstr_options</td>
<td>7</td>
</tr>
<tr>
<td>.cstr_pipe</td>
<td>8</td>
</tr>
<tr>
<td>.cstr_register_constructors</td>
<td>8</td>
</tr>
<tr>
<td>.cstr_repair_attributes</td>
<td>9</td>
</tr>
<tr>
<td>.cstr_wrap</td>
<td>10</td>
</tr>
<tr>
<td>.env</td>
<td>10</td>
</tr>
<tr>
<td>.xptr</td>
<td>11</td>
</tr>
<tr>
<td>compare_options</td>
<td>11</td>
</tr>
<tr>
<td>construct</td>
<td>12</td>
</tr>
<tr>
<td>constructors</td>
<td>15</td>
</tr>
<tr>
<td>construct_diff</td>
<td>15</td>
</tr>
<tr>
<td>construct_dump</td>
<td>17</td>
</tr>
<tr>
<td>construct_issues</td>
<td>17</td>
</tr>
<tr>
<td>construct_signature</td>
<td>18</td>
</tr>
<tr>
<td>custom-constructors</td>
<td>18</td>
</tr>
<tr>
<td>deparse_call</td>
<td>19</td>
</tr>
<tr>
<td>opts_array</td>
<td>20</td>
</tr>
<tr>
<td>opts_AsIs</td>
<td>20</td>
</tr>
<tr>
<td>opts_atomic</td>
<td>21</td>
</tr>
<tr>
<td>opts_classGeneratorFunction</td>
<td>23</td>
</tr>
<tr>
<td>opts_classPrototypeDef</td>
<td>23</td>
</tr>
<tr>
<td>opts_classRepresentation</td>
<td>24</td>
</tr>
<tr>
<td>opts_constructive_options</td>
<td>24</td>
</tr>
<tr>
<td>opts_data.frame</td>
<td>25</td>
</tr>
<tr>
<td>opts_data.table</td>
<td>26</td>
</tr>
<tr>
<td>opts_Date</td>
<td>27</td>
</tr>
<tr>
<td>opts_dm</td>
<td>28</td>
</tr>
<tr>
<td>opts_dots</td>
<td>28</td>
</tr>
<tr>
<td>opts_environment</td>
<td>29</td>
</tr>
<tr>
<td>opts_externalptr</td>
<td>31</td>
</tr>
<tr>
<td>opts_factor</td>
<td>32</td>
</tr>
<tr>
<td>opts_formula</td>
<td>33</td>
</tr>
<tr>
<td>opts_function</td>
<td>34</td>
</tr>
<tr>
<td>opts_grouped_df</td>
<td>35</td>
</tr>
<tr>
<td>opts_language</td>
<td>35</td>
</tr>
<tr>
<td>opts_Layer</td>
<td>36</td>
</tr>
<tr>
<td>opts_list</td>
<td>37</td>
</tr>
<tr>
<td>opts_matrix</td>
<td>38</td>
</tr>
<tr>
<td>opts_mts</td>
<td>39</td>
</tr>
<tr>
<td>opts_numeric_version</td>
<td>39</td>
</tr>
<tr>
<td>opts_ordered</td>
<td>40</td>
</tr>
<tr>
<td>opts_package_version</td>
<td>41</td>
</tr>
<tr>
<td>opts_pairlist</td>
<td>41</td>
</tr>
<tr>
<td>opts_POSIXct</td>
<td>42</td>
</tr>
<tr>
<td>opts_POSIXlt</td>
<td>43</td>
</tr>
<tr>
<td>opts_quosure</td>
<td>43</td>
</tr>
<tr>
<td>opts_quosures</td>
<td>44</td>
</tr>
</tbody>
</table>
Description

Exported for custom constructor design. If `recurse` is `TRUE` (default), we recurse to construct `args` and insert their construction code in a `fun(...)` call returned as a character vector. If `args` already contains code rather than object to construct one should set `recurse` to `FALSE`.

Usage

```r
cstr_apply(
  args,
  fun = "list",
  ..., trailingcomma = FALSE,
  recurse = TRUE,
  implicit_names = FALSE,
  new_line = TRUE,
  one_liner = FALSE
)
```

Arguments

- **args**
  A list of arguments to construct recursively, or code if `recurse` = `FALSE`. If elements are named, the arguments will be named in the generated code.

- **fun**
  The function name to use to build code of the form "fun(...)"

- **...**
  options passed recursively to the further methods

- **trailingcomma**
  leave a trailing comma after the last argument if the code is multiline, some constructors allow it (e.g. tibble::tibble()) and it makes for nicer diffs in version control.

- **recurse**
  Whether to recursively generate the code to construct `args`. If `FALSE` arguments are expected to contain code.

- **implicit_names**
  When data is provided, compress calls of the form `f(a = a)` to `f(a)`

- **new_line**
  passed to wrap to remove add a line after "fun(" and before ")", forced to `FALSE` if `one_liner` is `TRUE`

- **one_liner**
  Whether to return a one line call.
Value

A character vector of code

Examples

```r
a <- 1
.cstr_apply(list(a=a), "foo")
cstr_apply(list(a=a), "foo", data = list(a=1))
cstr_apply(list(a=a), "foo", data = list(a=1), implicit_names = TRUE)
cstr_apply(list(b=a), "foo", data = list(a=1), implicit_names = TRUE)
cstr_apply(list(a="c(1,2)"), "foo")
cstr_apply(list(a="c(1,2)"), "foo", recurse = FALSE)
```

Description

Exported for custom constructor design. This function allows combining independent checks so information is given about all failing checks rather than the first one. All parameters except ... are forwarded to rlang::abort()

Usage

```r
.cstr_combine_errors(
  ..., 
  class = NULL,
  call,
  header = NULL,
  body = NULL,
  footer = NULL,
  trace = NULL,
  parent = NULL,
  use_cli_format = NULL,
  .internal = FALSE,
  .file = NULL,
  .frame = parent.frame(),
  .trace_bottom = NULL
)
```

Arguments

... check expressions
class Subclass of the condition.
call  The execution environment of a currently running function, e.g. `call = caller_env()`.
    The corresponding function call is retrieved and mentioned in error messages as the source of the error.

    You only need to supply `call` when throwing a condition from a helper function which wouldn’t be relevant to mention in the message.

    Can also be NULL or a defused function call to respectively not display any call or hard-code a code to display.

    For more information about error calls, see Including function calls in error messages.

header  An optional header to precede the errors

body, footer  Additional bullets.

trace  A trace object created by `trace_back()`.

parent Supply `parent` when you rethrow an error from a condition handler (e.g. with `try_fetch()`).

    • If `parent` is a condition object, a chained error is created, which is useful when you want to enhance an error with more details, while still retaining the original information.

    • If `parent` is `NA`, it indicates an unchained rethrow, which is useful when you want to take ownership over an error and rethrow it with a custom message that better fits the surrounding context.

    Technically, supplying `NA` lets `abort()` know it is called from a condition handler. This helps it create simpler backtraces where the condition handling context is hidden by default.

    For more information about error calls, see Including contextual information with error chains.

use_cli_format  Whether to format `message` lazily using cli if available. This results in prettier and more accurate formatting of messages. See `local_use_cli()` to set this condition field by default in your package namespace.

    If set to TRUE, `message` should be a character vector of individual and unformatted lines. Any newline character "\n" already present in `message` is reformatted by cli’s paragraph formatter. See Formatting messages with cli.

.internal  If TRUE, a footer bullet is added to `message` to let the user know that the error is internal and that they should report it to the package authors. This argument is incompatible with footer.

.file  A connection or a string specifying where to print the message. The default depends on the context, see the stdout vs stderr section.

.frame  The throwing context. Used as default for .trace_bottom, and to determine the internal package to mention in internal errors when .internal is TRUE.

.trace_bottom  Used in the display of simplified backtraces as the last relevant call frame to show. This way, the irrelevant parts of backtraces corresponding to condition handling (`tryCatch()`, `try_fetch()`, `abort()`, etc.) are hidden by default. Defaults to call if it is an environment, or .frame otherwise. Without effect if trace is supplied.
Description

Exported for custom constructor design. `.cstr_construct()` is basically a naked `construct()`, without the checks, the style, the object post processing etc..

Usage

`.cstr_construct(x, ..., data = NULL)`

Arguments

- `x` An object, for `construct_multi()` a named list or an environment.
- `...` Constructive options built with the `opts_*()` family of functions. See the "Constructive options" section below.
- `data` Named list or environment of objects we want to detect and mention by name (as opposed to deparsing them further). Can also contain unnamed nested lists, environments, or package names, in the latter case package exports and datasets will be considered. In case of conflict, the last provided name is considered.

Value

A character vector

Description

Exported for custom constructor design.

Usage

`.cstr_fetch_opts(class, ..., template = NULL)`

Arguments

- `class` A string. An S3 class.
- `..., template` Parameters generally forwarded through the dots of the caller function

Value

An object of class `c(paste0("constructive_options_", class), "constructive_options")`
.cstr_match_constructor

*Validate a constructor*

**Description**

Fails if the chosen constructor doesn’t exist.

**Usage**

`.cstr_match_constructor(constructor, class)`

**Arguments**

- **constructor**
  a String (or character vector but only the first item will be considered)
- **class**
  A string

**Value**

A string, the first value of `constructor` if it is the name of an existing constructor or "next".

---

.cstr_options

*Create constructive options*

**Description**

Exported for custom constructor design.

**Usage**

`.cstr_options(class, ...)`

**Arguments**

- **class**
  A string. An S3 class.
- **...**
  Options to set

**Value**

An object of class `c(paste0("constructive_options_", class), "constructive_options")`
.cstr_pipe

**Insert a pipe between two calls**

**Description**

Exported for custom constructor design.

**Usage**

`cstr_pipe(x, y, pipe, one_liner)`

**Arguments**

- **x**
  - A character vector. The code for the left hand side call.
- **y**
  - A character vector. The code for the right hand side call.
- **pipe**
  - A string. The pipe to use, "plus" is useful for ggplot code.
- **one_liner**
  - A boolean. Whether to paste x, the pipe and y together on a same line (provided that x and y are strings and one liners themselves)

**Value**

A character vector

**Examples**

```
cstr_pipe("iris", "head(2)", pipe = "base", one_liner = FALSE)
cstr_pipe("iris", "head(2)", pipe = "base", one_liner = TRUE)
```

---

.cstr_register_constructors

**Register constructors**

**Description**

Use this function to register a custom constructor. See vignette for more information.

**Usage**

`cstr_register_constructors(class, ...)`

**Arguments**

- **class**
  - A string
- **...**
  - named constructors
Value

Returns NULL invisibly, called for side effects.

Description

Exported for custom constructor design. In the general case an object might have more attributes than given by the idiomatic construction. `.cstr_repair_attributes()` sets some of those attributes and ignores others.

Usage

```r
.cstr_repair_attributes(
  x,
  code,
  ..., 
  pipe = "base",
  ignore = NULL,
  idiomatic_class = NULL,
  remove = NULL,
  one_liner = FALSE
)
```

Arguments

- `x` The object to construct
- `code` The code constructing the object before attribute reparation
- `...` Forwarded to `.construct_apply()` when relevant
- `pipe` Which pipe to use, either "base" or "magrittr"
- `ignore` The attributes that shouldn’t be repaired, i.e. we expect them to be set by the constructor already in code
- `idiomatic_class` The class of the objects that the constructor produces, if `x` is of class `idiomatic_class` there is no need to repair the class.
- `remove` Attributes that should be removed, should rarely be useful.
- `one_liner` Boolean. Whether to collapse the output to a single line of code.

Value

A character vector
.cstr_wrap  
Wrap argument code in function call

**Description**

Exported for custom constructor design. Generally called through .cstr_apply().

**Usage**

`.cstr_wrap(args, fun, new_line = FALSE)`

**Arguments**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
<td>A character vector containing the code of arguments.</td>
</tr>
<tr>
<td>fun</td>
<td>A string. The name of the function to use in the function call. Use <code>fun = &quot;&quot;</code> to wrap in parentheses.</td>
</tr>
<tr>
<td>new_line</td>
<td>Boolean. Whether to insert a new line between &quot;fun(“ and the closing ”)&quot;.</td>
</tr>
</tbody>
</table>

**Value**

A character vector.

.env  
Fetch environment from memory address

**Description**

This is designed to be used in constructed output. The parents and ... arguments are not processed and only used to display additional information. If used on an improper memory address the output might be erratic or the session might crash.

**Usage**

`.env(address, parents = NULL, ...)`

**Arguments**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>Memory address of the environment</td>
</tr>
<tr>
<td>parents, ...</td>
<td>ignored</td>
</tr>
</tbody>
</table>

**Value**

The environment that the memory address points to.
Description

Base R doesn’t provide utilities to build or manipulate external pointers (objects of type "externalptr"), so we provide our own. Be warned that objects defined with `.xptr()` are not stable across sessions, however this is the best we can do.

Usage

`.xptr(address)`

Arguments

| address | Memory address |

Value

The external pointer (type "externalptr") that the memory address points to.

---

**compare_options**  
Options for `waldo::compare`

Description

Builds options that will be passed to `waldo::compare()` down the line.

Usage

```r
compare_options(
  ignore_srcref = TRUE,
  ignore_attr = FALSE,
  ignore_function_env = FALSE,
  ignore_formula_env = FALSE
)
```

Arguments

| ignore_srcref | Ignore differences in function srcrefs? TRUE by default since the srcref does not change the behaviour of a function, only its printed representation. |
ignore_attr  Ignore differences in specified attributes? Supply a character vector to ignore differences in named attributes. By default the "waldo_opts" attribute is listed in ignore_attr so that changes to it are not reported; if you customize ignore_attr, you will probably want to do this yourself.

For backward compatibility with all.equal(), you can also use TRUE, to all ignore differences in all attributes. This is not generally recommended as it is a blunt tool that will ignore many important functional differences.

ignore_function_env, ignore_formula_env
Ignore the environments of functions and formulas, respectively? These are provided primarily for backward compatibility with all.equal() which always ignores these environments.

Value
A list

```
construct  Build code to recreate an object

Description
construct() builds the code to reproduce one object, construct_multi() builds the code to reproduce objects stored in a named list or environment.

Usage
construct(
  x,
  ..., compression
  data = NULL,
  pipe = c("base", "magrittr"),
  check = NULL,
  compare = compare_options(),
  one_liner = FALSE,
  template = getOption("constructive_opts_template")
)

construct_multi(
  x,
  ..., compression
  data = NULL,
  pipe = c("base", "magrittr"),
  check = NULL,
  compare = compare_options(),
  one_liner = FALSE,
  template = getOption("constructive_opts_template")
)```
Arguments

- **x**: An object, for `construct_multi()` a named list or an environment.
- **...**: Constructive options built with the `opts_*()` family of functions. See the "Constructive options" section below.
- **data**: Named list or environment of objects we want to detect and mention by name (as opposed to deparsing them further). Can also contain unnamed nested lists, environments, or package names, in the latter case package exports and datasets will be considered. In case of conflict, the last provided name is considered.
- **pipe**: Which pipe to use, either "base" or "magrittr".
- **check**: Boolean. Whether to check if the created code reproduces the object using `waldo::compare()`.
- **compare**: Parameters passed to `waldo::compare()`, built with `compare_options()`.
- **one_liner**: Boolean. Whether to collapse the output to a single line of code.
- **template**: A list of constructive options built with `opts_*()` functions, they will be overriden by .... Use it to set a default behavior for `constructive`.

Value

An object of class 'constructive'.

Constructive options

Constructive options provide a way to customize the output of `construct()`. We can provide calls to `opts_*()` functions to the `...` argument. Each of these functions targets a specific type or class and is documented on its own page.

- `opts_AsIs(constructor = c("I", "next", "atomic"), ...)
- `opts_Date(constructor = c("as.Date", "as_date", "date", "new_date", "as.Date.numeric", "as_date.numeric", "next", "atomic"), ..., origin = "1970-01-01")
- `opts_Layer(constructor = c("default", "layer", "environment"), ...)
- `opts_POSIXct(constructor = c("as.POSIXct", ".POSIXct", "as_datetime", "as.POSIXct.numeric", "as_datetime.numeric", "next", "atomic"), ..., origin = "1970-01-01")
- `opts_POSIXlt(constructor = c("as.POSIXlt", "next", "list"), ...)
- `opts_R_system_version(constructor = c("R_system_version", "next", "atomic"), ...)
- `opts_S4(constructor = c("new"), ...)
- `opts_array(constructor = c("array", "next"), ...)
- `opts_atomic(..., trim = NULL, fill = c("default", "rlang", "+", "...", "none"), compress = TRUE, unicode_representation = c("ascii", "latin", "character", "unicode"), escape = FALSE)
- `opts_classGeneratorFunction(constructor = c("setClass"), ...)
- `opts_classPrototypeDef(constructor = c("prototype"), ...)
- `opts_classRepresentation(constructor = c("getClassDef"), ...)
- `opts_constructive_options(constructor = c("opts", "next"), ...)


construct

- `opts_data.frame`(constructor = c("data.frame", "read.table", "next", "list"), ...)
- `opts_data.table`(constructor = c("data.table", "next", "list"), ..., selfref = FALSE)
- `opts_dm`(constructor = c("dm", "next", "list"), ...)
- `opts_dots`(constructor = c("default"), ...)
- `opts_environment`(constructor = c(".env", "list2env", "as.environment", "new.env", "topenv", "new_environment"), ..., recurse = FALSE, predefine = FALSE)
- `opts_externalptr`(constructor = c("default"), ...)
- `opts_factor`(constructor = c("factor", "as_factor", "new_factor", "next", "atomic"), ...)
- `opts_formula`(constructor = c("~", "formula", "as.formula", "new_formula"), ..., environment = TRUE)
- `opts_function`(constructor = c("function", "as.function", "new_function"), ..., environment = TRUE, srcref = FALSE, trim = NULL)
- `opts_grouped_df`(constructor = c("default", "next", "list"), ...)
- `opts_language`(constructor = c("default"), ...)
- `opts_list`(constructor = c("list", "list2"), ..., trim = NULL, fill = c("vector", "new_list", "+", "...", "none"))
- `opts_matrix`(constructor = c("matrix", "array", "next”, "atomic"), ...)
- `opts_mts`(constructor = c("ts", "next", "atomic"), ...)
- `opts_numeric_version`(constructor = c("numeric_version", "next", "atomic"), ...)
- `opts_ordered`(constructor = c("ordered", "factor", "new_ordered", "next", "atomic"), ...)
- `opts_package_version`(constructor = c("package_version", "next", "atomic"), ...)
- `opts_pairlist`(constructor = c("pairlist", "pairlist2"), ...)
- `opts_quosure`(constructor = c("new_quosure", "next", "language"), ...)
- `opts_quosures`(constructor = c("new_quosures", "next", "list"), ...)
- `opts_rowwise_df`(constructor = c("default", "next", "list"), ...)
- `opts_tbl_df`(constructor = c("tibble", "tribble", "next", "list"), ..., trailing_comma = TRUE)
- `opts_ts`(constructor = c("ts", "next", "atomic"), ...)
- `opts_vctrs_list_of`(constructor = c("list_of", "list"), ...)
- `opts_weakref`(constructor = c("new_weakref"), ...)

Examples

```
construct(head(cars))
construct(head(cars), opts_data.frame("read.table"))
construct(head(cars), opts_data.frame("next"))
construct(iris$Species)
construct(iris$Species, opts_atomic(compress = FALSE), opts_factor("new_factor"))
```
constructors

Description
A nested environment containing constructor functions for the package constructive

Usage
constructors

Format
An object of class environment of length 36.

---

construct_diff

Display diff of object definitions

Description
Display diff of object definitions

Usage

construct_diff(
  target,
  current,
  ..., .
  data = NULL,
  pipe = c("base", "magrittr"),
  check = TRUE,
  compare = compare_options(),
  one_liner = FALSE,
  template = getOption("constructive_opts_template"),
  mode = c("sidebyside", "auto", "unified", "context"),
  interactive = TRUE
)

Arguments

  target the reference object
  current the object being compared to target
  ... Constructive options built with the opts_*() family of functions. See the "Constructive options" section below.
construct_diff

**data**
Named list or environment of objects we want to detect and mention by name (as opposed to deparsing them further). Can also contain unnamed nested lists, environments, or package names, in the latter case package exports and datasets will be considered. In case of conflict, the last provided name is considered.

**pipe**
Which pipe to use, either "base" or "magrittr"

**check**
Boolean. Whether to check if the created code reproduces the object using waldo::compare().

**compare**
Parameters passed to waldo::compare(), built with compare_options().

**one_liner**
Boolean. Whether to collapse the output to a single line of code.

**template**
A list of constructive options built with opts_*() functions, they will be overriden by .... Use it to set a default behavior for {constructive}.

**mode, interactive**
Passed to diffobj::diffChr()

---

**Value**

Returns NULL invisibly, called for side effects

---

**Examples**

```r
## Not run:
# some object print the same though they're different
# `construct_diff()` shows how they differ:
df1 <- data.frame(a=1, b = "x")
df2 <- data.frame(a=1L, b = "x", stringsAsFactors = TRUE)
attr(df2, "some_attribute") <- "a value"
df1
df2
construct_diff(df1, df2)

# Those are made easy to compare
construct_diff(substr, substring)
construct_diff(month.abb, month.name)

# more examples borrowed from {waldo} package
construct_diff(c("a", "b", "c"), c("a", "B", "c"))
construct_diff(c("X", letters), c(letters, "X"))
construct_diff(list(factor("x")), list(1L))
construct_diff(df1, df2)
x <- list(a = list(b = list(c = list(structure(1, e = 1)))))
y <- list(a = list(b = list(c = list(structure(1, e = "a")))))
construct_diff(x, y)

## End(Not run)
```
construct_dump

Dump Constructed Code to a File

**Description**

An alternative to `base::dump()` using code built with constructive

**Usage**

```r
construct_dump(x, path, append = FALSE, ...)
```

**Arguments**

- `x`: A named list or an environment.
- `path`: File or connection to write to.
- `append`: If FALSE, will overwrite existing file. If TRUE, will append to existing file. In both cases, if the file does not exist a new file is created.
- `...`: Forwarded to `construct_multi()`

**Value**

Returns NULL invisibly, called for side effects.

---

construct_issues

Show constructive issues

**Description**

Show constructive issues

**Usage**

```r
construct_issues(x = NULL)
```

**Arguments**

- `x`: An object built by `construct()`, if NULL the latest encountered issues will be displayed

**Value**

A character vector with class "waldo_compare"
### construct_signature

*Construct a function’s signature*

**Description**

Construct a function’s signature

**Usage**

```r
construct_signature(x, name = NULL, one_liner = FALSE, style = TRUE)
```

**Arguments**

- **x**: A function
- **name**: The name of the function, by default we use the symbol provided to `x`
- **one_liner**: Boolean. Whether to collapse multi-line expressions on a single line using semicolons
- **style**: Boolean. Whether to use `styler::style_text()` on the output

**Value**

A character vector with the class "vertical" (for nice printing with styler)

**Examples**

```r
construct_signature(lm)
```

---

### custom-constructors

*Custom constructors*

**Description**

We export a collection of functions that can be used to design custom methods for `.cstr_construct()` or custom constructors for a given method.

**Details**

- `.cstr_construct`: Low level generic for object construction code generation
- `.cstr_repair_attributes`: Helper to repair attributes of objects
- `.cstr_options`: Define and check options to pass to custom constructors
- `.cstr_fetch_opts`
- `.cstr_apply`
- `.cstr_wrap`
- `.cstr_pipe`
- `.cstr_combine_errors`
deparse_call

Deparse a language object

Description

This is an alternative to `base::deparse()` and `rlang::expr_deparse()` that handles additional corner cases and fails when encountering tokens other than symbols and syntactic literals where cited alternatives would produce non syntactic code.

Moreover it will produce a one liner if `one_liner = TRUE` and will attempt to use the base pipe on nested calls if `pipe = TRUE`.

By default the output is styled with styler, if style is set to `FALSE` a regular string is returned, using "\n" for new lines.

Usage

deparse_call(call, one_liner = FALSE, pipe = FALSE, style = TRUE)

Arguments

call: A call

one_liner: Boolean. Whether to collapse multi-line expressions on a single line using semicolons

pipe: Boolean. Whether to use the base pipe to disentangle nested calls. This works best on simple calls.

style: Boolean. Whether to use `styler::style_text()` on the output

Value

a string or a styled character vector

Examples

```r
expr <- quote(foo(bar({this; that}, 1)))
deparse_call(expr)
deparse_call(expr, one_liner = TRUE)
deparse_call(expr, pipe = TRUE)
deparse_call(expr, style = FALSE)
# some corner cases are handled better than in base R
deparse(call("$", 1, 1)) # returns non syntactic output
deparse_call(call("$", 1, 1))
```
opts_array

Constructive options for arrays

Description

These options will be used on arrays. Note that arrays can be built on top of vectors, lists or expressions. Canonical arrays have an implicit class "array" shown by `class()` but "array" is not part of the class attribute.

Usage

```r
opts_array(constructor = c("array", "next"), ...)
```

Arguments

- `constructor` String. Name of the function used to construct the environment, see Details section.
- `...` Should not be used. Forces passing arguments by name.

Details

Depending on `constructor`, we construct the object as follows:

- "array" (default): Use the `array()` function
- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried.

Value

An object of class <constructive_options/constructive_options_array>

opts_AsIs

Constructive options for the class AsIs

Description

These options will be used on objects of class AsIs. AsIs objects are created with `I()` which only prepends "AsIs" to the class attribute.

Usage

```r
opts_AsIs(constructor = c("I", "next", "atomic"), ...)
```
opts_atomic

Arguments

- **constructor**: String. Name of the function used to construct the environment, see Details section.
- **...**: Should not be used. Forces passing arguments by name.

Details

Depending on **constructor**, we construct the object as follows:

- **"I"** (default): Use the **I()** function
- **"next"**: Use the constructor for the next supported class. Call .class2() on the object to see in which order the methods will be tried.
- **"atomic"**: We define as an atomic vector and repair attributes

Value

An object of class `<constructive_options/constructive_options_array>`

---

**opts_atomic**  
*Constructive options for atomic types*

---

Description

These options will be used on atomic types ("logical", "integer", "numeric", "complex", "character" and "raw")

Usage

```r
opts_atomic(
    ..., 
    trim = NULL,
    fill = c("default", "rlang", "+", ":", "none"),
    compress = TRUE,
    unicode_representation = c("ascii", "latin", "character", "unicode"),
    escape = FALSE
)
```

Arguments

- **...**: Should not be used. Forces passing arguments by name.
- **trim**: NULL or integerish. Maximum of elements showed before it’s trimmed. Note that it will necessarily produce code that doesn’t reproduce the input. This code will parse without failure but its evaluation might fail.
- **fill**: String. Method to use to represent the trimmed elements.
compress is a Boolean. It replaces `c()` with `seq()`, `rep()`, or atomic constructors `logical()`, `integer()`, `numeric()`, `complex()`, `raw()` when relevant to simplify the output.

### unicode_representation

By default "ascii", which means only ASCII characters (code point < 128) will be used to construct a string. This makes sure that homoglyphs (different spaces and other identically displayed unicode characters) are printed differently, and avoid possible unfortunate copy and paste auto conversion issues. "latin" is more lax and uses all latin characters (code point < 256). "character" shows all characters, but not emojis. Finally "unicode" displays all characters and emojis, which is what `dput()` does.

### escape

Whether to escape double quotes and backslashes. If `FALSE` we use single quotes to surround strings containing double quotes, and raw strings for strings that contain backslashes and/or a combination of single and double quotes. Depending on `unicode_representation` escape = `FALSE` cannot be applied on all strings.

### Details

If `trim` is provided, depending on `fill` we will present trimmed elements as followed:

- **"default"**: Use default atomic constructors, so for instance `c("a", "b", "c")` might become `c("a", character(2)).`
- **"rlang"**: Use rlang atomic constructors, so for instance `c("a", "b", "c")` might become `c("a", rlang::new_character(2)). These rlang constructors create vectors of NAs, so it’s different from the default option.
- **"+"**: Use unary +, so for instance `c("a", "b", "c")` might become `c("a", +2).
- **"..."**: Use ..., so for instance `c("a", "b", "c")` might become `c("a", ...)`
- **"none"**: Don’t represent trimmed elements.

Depending on the case some or all of the choices above might generate code that cannot be executed. The 2 former options above are the most likely to succeed and produce an output of the same type and dimensions recursively. This would at least be the case for data frame.

### Value

An object of class `<constructive_options/constructive_options_atomic>`

### Examples

```r
construct(iris, opts_atomic(trim = 2), check = FALSE) # fill = "default"
construct(iris, opts_atomic(trim = 2, fill = "rlang"), check = FALSE)
construct(iris, opts_atomic(trim = 2, fill = "+") , check = FALSE)
construct(iris, opts_atomic(trim = 2, fill = "..." ), check = FALSE)
construct(iris, opts_atomic(trim = 2, fill = "none"), check = FALSE)
x <- c("a a", "a\U000000A0a", "a\U00002002a", "\U430 \U430")
construct(x, opts_atomic(unicode_representation = "unicode"))
construct(x, opts_atomic(unicode_representation = "character"))
construct(x, opts_atomic(unicode_representation = "latin"))
construct(x, opts_atomic(unicode_representation = "ascii"))
```
**opts_classGeneratorFunction**

*Constructive options for class 'classGeneratorFunction'*

**Description**

These options will be used on objects of class 'classGeneratorFunction'.

**Usage**

```r
opts_classGeneratorFunction(constructor = c("setClass"), ...)
```

**Arguments**

- `constructor`  
  String. Name of the function used to construct the object.
- `...`  
  Should not be used. Forces passing arguments by name.

**Value**

An object of class `<constructive_options/constructive_options_classGeneratorFunction>`

---

**opts_classPrototypeDef**

*Constructive options for class 'classPrototypeDef'*

**Description**

These options will be used on objects of class 'classPrototypeDef'.

**Usage**

```r
opts_classPrototypeDef(constructor = c("prototype"), ...)
```

**Arguments**

- `constructor`  
  String. Name of the function used to construct the object, see Details section.
- `...`  
  Should not be used. Forces passing arguments by name.

**Value**

An object of class `<constructive_options/constructive_options_classPrototypeDef>`
**opts_classRepresentation**

*Constructive options for class 'classRepresentation'*

**Description**

These options will be used on objects of class 'classRepresentation'.

**Usage**

```r
opts_classRepresentation(constructor = c("getClassDef"), ...)
```

**Arguments**

- `constructor` String. Name of the function used to construct the object.
- `...` Should not be used. Forces passing arguments by name.

**Value**

An object of class `<constructive_options/constructive_options_classRepresentation>`

---

**opts_constructive_options**

*Constructive options for the class constructive_options*

**Description**

These options will be used on objects of class `constructive_options`.

**Usage**

```r
opts_constructive_options(constructor = c("opts", "next"), ...)
```

**Arguments**

- `constructor` String. Name of the function used to construct the environment, see Details section.
- `...` Should not be used. Forces passing arguments by name.

**Details**

Depending on `constructor`, we construct the object as follows:

- "opts": Use the relevant `constructive::opts_?()` function.
- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried.
Value

An object of class <constructive_options/constructive_options_array>

Description

These options will be used on objects of class 'data.frame'.

Usage

```r
opts_data.frame(
  constructor = c("data.frame", "read.table", "next", "list"),
  ...
)
```

Arguments

- **constructor**  String. Name of the function used to construct the environment, see Details section.
- **...**  Should not be used. Forces passing arguments by name.

Details

Depending on `constructor`, we construct the object as follows:

- "data.frame" (default): Wrap the column definitions in a `data.frame()` call. If some columns are lists or data frames, we wrap the column definitions in `tibble::tibble()`, then use `as.data.frame()`.
- "read.table": We build the object using `read.table()` if possible, or fall back to `data.frame()`.
- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried.
- "list": Use `list()` and treat the class as a regular attribute.

Value

An object of class <constructive_options/constructive_options_data.frame>
opts_data.table  Constructive options for class 'data.table'

Description

These options will be used on objects of class 'data.table'.

Usage

opts_data.table(
  constructor = c("data.table", "next", "list"),
  ..., selfref = FALSE
)

Arguments

constructor  String. Name of the function used to construct the environment, see Details section.
...
selfref  Boolean. Whether to include the .internal.selfref attribute. It's probably not useful, hence the default, waldo::compare() is used to assess the output fidelity and doesn’t check it, but if you really need to generate code that builds an object identical() to the input you’ll need to set this to TRUE.

Details

Depending on constructor, we construct the object as follows:

- "data.table" (default): Wrap the column definitions in a data.table() call.
- "next": Use the constructor for the next supported class. Call .class2() on the object to see in which order the methods will be tried.
- "list": Use list() and treat the class as a regular attribute.

Value

An object of class <constructive_options/constructive_options_data.table>
opts_Date

Description

These options will be used on objects of class 'date'.

Usage

```r
opts_Date(
  constructor = c("as.Date", "as_date", "date", "new_date", "as.Date.numeric", 
                 "as_date.numeric", "next", "atomic"),
  ..., 
  origin = "1970-01-01"
)
```

Arguments

- `constructor` String. Name of the function used to construct the environment.
- `...` Should not be used. Forces passing arguments by name.
- `origin` Origin to be used, ignored when irrelevant.

Details

Depending on `constructor`, we construct the environment as follows:

- "as.Date" (default): We wrap a character vector with `as.Date()`, if the date is infinite it cannot be converted to character and we wrap a numeric vector and provide an `origin` argument.
- "as_date": Similar as above but using `lubridate::as_date()`, the only difference is that we never need to supply `origin`.
- "date": Similar as above but using `lubridate::date()`, it doesn’t support infinite dates so we fall back on `lubridate::as_date()` when we encounter them.
- "new_date": We wrap a numeric vector with `vctrs::new_date()`.
- "as.Date.numeric": We wrap a numeric vector with `as.Date()` and use the provided `origin`.
- "as_date.numeric": Same as above but using `lubridate::as_date()` and use the provided `origin`.
- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried.
- "atomic": We define as an atomic vector and repair attributes.

Value

An object of class `<constructive_options/constructive_options_environment>`
opts_dm

Constructive options class 'dm'

Description
These options will be used on objects of class 'dm'.

Usage
opts_dm(constructor = c("dm", "next", "list"), ...)

Arguments
constructor  String. Name of the function used to construct the environment.
...          Should not be used. Forces passing arguments by name.

Details
Depending on constructor, we construct the environment as follows:

• "dm" (default): We use dm::dm() and other functions from dm to adjust the content.
• "next": Use the constructor for the next supported class. Call .class2() on the object to see in which order the methods will be tried.
• "list": Use list() and treat the class as a regular attribute.

Value
An object of class <constructive_options/constructive_options_environment>

opts_dots

Constructive options for type '...'

Description
These options will be used on objects of type '...'. These are rarely encountered in practice. By default this function is useless as nothing can be set, this is provided in case users want to extend the method with other constructors.

Usage
opts_dots(constructor = c("default"), ...)

Arguments
constructor  String. Name of the function used to construct the environment.
...          Should not be used. Forces passing arguments by name.
Details

Depending on constructor, we construct the environment as follows:

- "default": We use the construct (function(...) environment()$...)(a = x, y) which we evaluate in the correct environment.

Value

An object of class <constructive_options/constructive_options_environment>

---

`opts_environment`  **Constructive options for type 'environment'**

Description

Environments use reference semantics, they cannot be copied. An attempt to copy an environment would indeed yield a different environment and identical(env, copy) would be FALSE. Moreover most environments have a parent (exceptions are emptyenv() and some rare cases where the parent is NULL) and thus to copy the environment we’d have to have a way to point to the parent, or copy it too.

For this reason environments are constructive’s cryptonite. They make some objects impossible to reproduce exactly. And since every function or formula has one they’re hard to avoid.

Usage

```r
opts_environment(
  constructor = c(".env", "list2env", "as.environment", "new.env", "topenv", "new_environment"),
  ..., recurse = FALSE,
  predefine = FALSE
)
```

Arguments

- `constructor`  String. Name of the function used to construct the environment, see **Constructors** section.
- `...`  Should not be used. Forces passing arguments by name.
- `recurse`  Boolean. Only considered if constructor is "list2env" or "new_environment". Whether to attempt to recreate all parent environments until a known environment is found, if FALSE (the default) we will use topenv() to find a known ancestor to set as the parent.
- `predefine`  Boolean. Whether to define environments first. If TRUE constructor and recurse are ignored. It circumvents the circularity, recursivity and redundancy issues of other constructors. The caveat is that the created code won’t be a single call and will create objects in the workspace.
Details

In some case we can build code that points to a specific environment, namely:

- `.GlobalEnv`, `.BaseNamespaceEnv`, `baseenv()` and `emptyenv()` are used to construct the global environment, the base namespace, the base package environment and the empty environment
- Namespaces are constructed using `asNamespace("pkg")`
- Package environments are constructed using `as.environment("package:pkg")`

By default for other environments we use constructive’s function `constructive::.env()`, it fetches the environment from its memory address and provides as additional information the sequence of parents until we reach a special environment (those enumerated above). The advantage of this approach is that it’s readable and that the object is accurately reproduced. The inconvenient is that it’s not stable between sessions. If an environment has a NULL parent it’s always constructed with `constructive::.env()`, whatever the choice of the constructor.

Often however we wish to be able to reproduce from scratch a similar environment, so that we might run the constructed code later in a new session. We offer different different options to do this, with different trade-offs regarding accuracy and verbosity.

`{constructive}` will not signal any difference if it can reproduce an equivalent environment, defined as containing the same values and having a same or equivalent parent.

See also the `ignore_function_env` argument in `?compare_options`, which disables the check of environments of function.

Value

An object of class `<constructive_options/constructive_options_environment>`

Constructors

We might set the constructor argument to:

- ".env" (default): use `constructive::.env()` to construct the environment from its memory address.
- "list2env": We construct the environment as a list then use `base::list2env()` to convert it to an environment and assign it a parent. By default we will use `base::topenv()` to construct a parent. If `recurse` is `TRUE` the parent will be built recursively so all ancestors will be created until we meet a known environment, this might be verbose and will fail if environments are nested too deep or have a circular relationship. If the environment is empty we use `new.env(parent=)` for a more economic syntax.
- "new_environment": Similar to the above, but using `rlang::new_environment()`.
- "new.env": All environments will be recreated with the code "base::new.env()", without argument, effectively creating an empty environment child of the local (often global) environment. This is enough in cases where the environment doesn’t matter (or matters as long as it inherits from the local environment), as is often the case with formulas. `recurse` is ignored.
• "as.environment": we attempt to construct the environment as a list and use `base::as.environment()` on top of it, as in `as.environment(list(a=1, b=2))`, it will contain the same variables as the original environment but the parent will be the `emptyenv()`. `recurse` is ignored.

• "topenv": we construct `base::topenv(x)`, see `?topenv`. `recurse` is ignored. This is the most accurate we can be when constructing only special environments.

Predefine

Building environments from scratch using the above methods can be verbose and sometimes redundant if an environment is used several times. One last option is to define the environments and their content above the object returning call, using placeholder names `.env.1`, `.env.2`, etc. This is done by setting `predefine` to `TRUE`. `constructor` and `recurse` are ignored in that case.

---

**opts_externalptr**

Constructive options for type 'externalptr'

---

**Description**

These options will be used on objects of type `externalptr`. By default this function is useless as nothing can be set, this is provided in case users want to extend the method with other constructors.

**Usage**

```r
opts_externalptr(constructor = c("default"), ...)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td>String. Name of the function used to construct the environment.</td>
</tr>
<tr>
<td>...</td>
<td>Should not be used. Forces passing arguments by name.</td>
</tr>
</tbody>
</table>

**Details**

Depending on `constructor`, we construct the environment as follows:

• "default": We use a special function from the constructive

**Value**

An object of class `<constructive_options/constructive_options_environment>`
opts_factor

Constructive options for class 'factor'

Description

These options will be used on objects of class 'factor'.

Usage

opts_factor(
  constructor = c("factor", "as_factor", "new_factor", "next", "atomic"),
  ...
)

Arguments

constructor String. Name of the function used to construct the environment, see Details section.
... Should not be used. Forces passing arguments by name.

Details

Depending on constructor, we construct the environment as follows:

- "factor" (default): Build the object using a factor() call, levels won’t be defined explicitly if they are in alphabetical order (locale dependent!)
- "as_factor": Build the object using a forcats::as_factor() call whenever possible, i.e. when levels are defined in order of appearance in the vector. Otherwise falls back to "factor" constructor.
- "new_factor": Build the object using a vctrs::new_factor() call. Levels are always defined explicitly.
- "next": Use the constructor for the next supported class. Call .class2() on the object to see in which order the methods will be tried.
- "atomic": We define as an atomic vector and repair attributes.

Value

An object of class <constructive_options/constructive_options_factor>
**Description**

These options will be used on formulas, defined as calls to ~, regardless of their "class" attribute.

**Usage**

```r
opts_formula(
  constructor = c("~", "formula", "as.formula", "new_formula"),
  ..., 
  environment = TRUE
)
```

**Arguments**

- **constructor** String. Name of the function used to construct the environment, see Details section.
- **...** Should not be used. Forces passing arguments by name.
- **environment** Boolean. Whether to attempt to construct the environment, if it makes a difference to construct it.

Depending on `constructor`, we construct the formula as follows:

- "~" (default): We construct the formula in the most common way using the ~ operator.
- "formula": deparse the formula as a string and use `base::formula()` on top of it.
- "as.formula": Same as above, but using `base::as.formula()`.
- "new_formula": extract both sides of the formula as separate language objects and feed them to `rlang::new_formula()`, along with the reconstructed environment if relevant.

**Value**

An object of class `<constructive_options/constructive_options_environment>`
Constructive options for functions

Description

These options will be used on functions, i.e. objects of type "closure", "special" and "builtin".

Usage

```r
opts_function(
  constructor = c("function", "as.function", "new_function"),
  ..., 
  environment = TRUE,
  srcref = FALSE,
  trim = NULL
)
```

Arguments

- `constructor`: String. Name of the function used to construct the environment, see Details section.
- `...`: Should not be used. Forces passing arguments by name.
- `environment`: Boolean. Whether to reconstruct the function’s environment.
- `srcref`: Boolean. Whether to attempt to reconstruct the function’s srcref.
- `trim`: NULL or integerish. Maximum of lines showed in the body before it’s trimmed, replacing code with .... Note that it will necessarily produce code that doesn’t reproduce the input, but it will parse and evaluate without failure.

Details

Depending on constructor, we construct the environment as follows:

- "function" (default): Build the object using a standard function() {} definition. This won’t set the environment by default, unless environment is set to TRUE. If a srcref is available, if this srcref matches the function’s definition, and if trim is left NULL, the code is returned from using the srcref, so comments will be shown in the output of construct().
- "as.function": Build the object using a as.function() call. back to data.frame().
- "new_function": Build the object using a rlang::new_function() call.

Value

An object of class <constructive_options/constructive_options_function>
opts_grouped_df

Constructive options for class 'grouped_df'

Description

These options will be used on objects of class 'grouped_df'.

Usage

```r
opts_grouped_df(constructor = c("default", "next", "list"), ...)
```

Arguments

- `constructor` : String. Name of the function used to construct the environment, see Details section.
- `...` : Should not be used. Forces passing arguments by name.

Details

Depending on `constructor`, we construct the environment as follows:

- "next" : Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried.
- "list" : We define as an list object and repair attributes.

Value

An object of class `<constructive_options/constructive_options_factor>`

opts_language

Constructive options for type 'language'

Description

These options will be used on objects of type 'language'. By default this function is useless as nothing can be set, this is provided in case users want to extend the method with other constructors.

Usage

```r
opts_language(constructor = c("default"), ...)
```

Arguments

- `constructor` : String. Name of the function used to construct the environment.
- `...` : Should not be used. Forces passing arguments by name.
Details

Depending on constructor, we construct the environment as follows:

- "default": We use constructive's deparsing algorithm on attributeless calls, and use as.call() on other language elements when attributes need to be constructed.

Value

An object of class <constructive_options/constructive_options_environment>

Description

These options will be used on objects of class 'Layer'.

Usage

opts_Layer(constructor = c("default", "layer", "environment"), ...)

Arguments

- constructor String. Name of the function used to construct the environment, see Details section.
- ... Should not be used. Forces passing arguments by name.

Details

Depending on constructor, we construct the object as follows:

- "default": We attempt to use the function originally used to create the plot.
- "layer": We use the ggplot2::layer() function
- "environment": Reconstruct the object using the general environment method (which can be itself tweaked using opts_environment())

The latter constructor is the only one that reproduces the object exactly since Layers are environments and environments can't be exactly copied (see ?opts_environment)

Value

An object of class <constructive_options/constructive_options_Layer>
**Description**

These options will be used on objects of type 'list'.

**Usage**

```r
opts_list(
  constructor = c("list", "list2"),
  ..., 
  trim = NULL,
  fill = c("vector", "new_list", "+", ",...", "none")
)
```

**Arguments**

- **constructor**: String. Name of the function used to construct the environment, see Details section.
- **...**: Should not be used. Forces passing arguments by name.
- **trim**: NULL or integerish. Maximum of elements showed before it’s trimmed. Note that it will necessarily produce code that doesn’t reproduce the input. This code will parse without failure but its evaluation might fail.
- **fill**: String. Method to use to represent the trimmed elements.

**Details**

Depending on constructor, we construct the environment as follows:

- "list" (default): Build the object by calling `list()`.
- "list2": Build the object by calling `rlang::list2()`, the only difference with the above is that we keep a trailing comma when the list is not trimmed and the call spans several lines.

If trim is provided, depending on fill we will present trimmed elements as followed:

- "vector" (default): Use `vector()`, so for instance `list("a", "b", "c")` might become `c(list("a"), vector("list", 2))`.
- "new_list": Use `rlang::new_list()`, so for instance `list("a", "b", "c")` might become `c(list("a"), rlang::new_list(2))`.
- "+": Use unary +, so for instance `list("a", "b", "c")` might become `list("a", +2)`.
- "...": Use ..., so for instance `list("a", "b", "c")` might become `list("a", ...)`
- "none": Don’t represent trimmed elements.

When trim is used the output is parsable but might not be possible to evaluate, especially with fill = "...". In that case you might want to set check = FALSE.
Value

An object of class `<constructive_options/constructive_options_list>`

---

opts_matrix  Constructive options for matrices

Description

Matrices are atomic vectors, lists, or objects of type "expression" with a "dim" attributes of length 2.

Usage

`opts_matrix(constructor = c("matrix", "array", "next", "atomic"), ...)`

Arguments

- **constructor**  String. Name of the function used to construct the environment.
- **...**  Should not be used. Forces passing arguments by name.

Details

Depending on `constructor`, we construct the environment as follows:

- "matrix": We use `matrix()`
- "array": We use `array()`
- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried. This will usually be equivalent to "array"
- "atomic": We define as an atomic vector and repair attributes

Value

An object of class `<constructive_options/constructive_options_environment>`
## opts_mts

**Constructive options for time-series objects**

### Description

Depending on `constructor`, we construct the environment as follows:

- "ts": We use `ts()`
- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried. This will usually be equivalent to "atomic"
- "atomic": We define as an atomic vector and repair attributes

### Usage

```r
opts_mts(constructor = c("ts", "next", "atomic"), ...)
```

### Arguments

- `constructor` String. Name of the function used to construct the environment.
- `...` Should not be used. Forces passing arguments by name.

### Value

An object of class `<constructive_options/constructive_options_environment>`

---

## opts_numeric_version

**Constructive options for numeric_version**

### Description

Depending on `constructor`, we construct the environment as follows:

- "numeric_version": We use `numeric_version()`
- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried. This will usually be equivalent to "array"
- "atomic": We define as an atomic vector and repair attributes

### Usage

```r
opts_numeric_version(constructor = c("numeric_version", "next", "atomic"), ...)
```

### Arguments

- `constructor` String. Name of the function used to construct the environment.
- `...` Should not be used. Forces passing arguments by name.
opts_ordered

Constructive options for class 'ordered'

Description

These options will be used on objects of class 'ordered'.

Usage

opts_ordered(
  constructor = c("ordered", "factor", "new_ordered", "next", "atomic"),
  ...
)

Arguments

constructor String. Name of the function used to construct the environment, see Details section.

... Should not be used. Forces passing arguments by name.

Details

Depending on constructor, we construct the environment as follows:

- "ordered" (default): Build the object using a ordered() call, levels won’t be defined explicitly if they are in alphabetical order (locale dependent!)
- "factor": Same as above but build the object using a factor() call and ordered = TRUE.
- "new_ordered": Build the object using a vctrs::new_ordered(). Levels are always defined explicitly.
- "next": Use the constructor for the next supported class. Call .class2() on the object to see in which order the methods will be tried.
- "atomic": We define as an atomic vector and repair attributes

Value

An object of class <constructive_options/construcive_options_factor>
opts_package_version  Constructive options for package_version

Description

Depending on constructor, we construct the environment as follows:

- "package_version" : We use package_version()
- "next" : Use the constructor for the next supported class. Call .class2() on the object to see in which order the methods will be tried. This will usually be equivalent to "array"
- "atomic" : We define as an atomic vector and repair attributes

Usage

opts_package_version(constructor = c("package_version", "next", "atomic"), ...)

Arguments

constructor  String. Name of the function used to construct the environment.
...  Should not be used. Forces passing arguments by name.

Value

An object of class <constructive_options/constructive_options_environment>

opts_pairlist  Constructive options for pairlists

Description

Depending on constructor, we construct the environment as follows:

- "pairlist" (default): Build the object using a pairlist() call.
- "pairlist2" : Build the object using a rlang::pairlist2() call.

Usage

opts_pairlist(constructor = c("pairlist", "pairlist2"), ...)

Arguments

constructor  String. Name of the function used to construct the environment, see Details section.
...  Should not be used. Forces passing arguments by name.

Value

An object of class <constructive_options/constructive_options_factor>
**opts_POSIXct**

**Constructive options for class 'POSIXct'**

**Description**

These options will be used on objects of class 'POSIXct'.

**Usage**

```r
opts_POSIXct(
  constructor = c("as.POSIXct", ".POSIXct", "as_datetime", "as.POSIXct.numeric", 
                  "as_datetime.numeric", "next", "atomic"),
  ...,
  origin = "1970-01-01"
)
```

**Arguments**

- `constructor` String. Name of the function used to construct the environment, see Details section.
- `...` Should not be used. Forces passing arguments by name.
- `origin` Origin to be used, ignored when irrelevant.

**Details**

Depending on `constructor`, we construct the environment as follows:

- "as.POSIXct" (default): Build the object using a `as.POSIXct()` call on a character vector.
- ".POSIXct": Build the object using a `POSIXct()` call on a numeric vector.
- "as_datetime": Build the object using a `lubridate::as_datetime()` call on a character vector.
- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried.
- "atomic": We define as an atomic vector and repair attributes.

**Value**

An object of class `<constructive_options/constructive_options_factor>`
Description

These options will be used on objects of class ‘POSIXlt’.

Usage

```r
opts_POSIXlt(constructor = c("as.POSIXlt", "next", "list"), ...)
```

Arguments

- `constructor` String. Name of the function used to construct the environment, see Details section.
- `...` Should not be used. Forces passing arguments by name.

Details

Depending on `constructor`, we construct the environment as follows:

- "as.POSIXlt" (default): Build the object using a `as.POSIXlt()` call on a character vector.
- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried.
- "list": We define as a list and repair attributes.

Value

An object of class `<constructive_options/constructive_options_factor>`

---

Description

These options will be used on objects of class ‘quosure’.

Usage

```r
opts_quosure(constructor = c("new_quosure", "next", "language"), ...)
```

Arguments

- `constructor` String. Name of the function used to construct the environment, see Details section.
- `...` Should not be used. Forces passing arguments by name.
Details

Depending on constructor, we construct the environment as follows:

- "new_quosure" (default): Build the object using a `new_quosure()` call on a character vector.
- "next" : Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried.
- "language" : We define as an language object and repair attributes.

Value

An object of class `<constructive_options/constructive_options_factor>`
**opts_rowwise_df**  
*Constructive options for class 'rowwise_df'*

**Description**

These options will be used on objects of class 'rowwise_df'.

**Usage**

```r
opts_rowwise_df(constructor = c("default", "next", "list"), ...)```

**Arguments**

- **constructor**  
  String. Name of the function used to construct the environment, see Details section.
- **...**  
  Should not be used. Forces passing arguments by name.

**Details**

Depending on constructor, we construct the environment as follows:

- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried.
- "list": We define as an list object and repair attributes.

**Value**

An object of class `<constructive_options/constructive_options_factor>`

---

**opts_R_system_version**  
*Constructive options for R_system_version*

**Description**

Depending on constructor, we construct the environment as follows:

- "R_system_version": We use `R_system_version()`
- "next": Use the constructor for the next supported class. Call `.class2()` on the object to see in which order the methods will be tried. This will usually be equivalent to "array"
- "atomic": We define as an atomic vector and repair attributes

**Usage**

```r
opts_R_system_version(  
    constructor = c("R_system_version", "next", "atomic"),  
    ...  
  )
)```
opts_tbl_df

Arguments

  constructor  String. Name of the function used to construct the environment.
  ...          Should not be used. Forces passing arguments by name.

Value

An object of class <constructive_options/constructive_options_environment>

---

opts_S4  

Constructive options for class 'S4'

Description

These options will be used on objects of class 'S4'. Note that the support for S4 is very experimental so might easily break. Please report issues if it does.

Usage

opts_S4(constructor = c("new"), ...)

Arguments

  constructor  String. Name of the function used to construct the environment, see Details section.
  ...          Should not be used. Forces passing arguments by name.

Value

An object of class <constructive_options/constructive_options_S4>

---

opts_tbl_df  

Constructive options for tibbles

Description

These options will be used on objects of class 'tbl_df', also known as tibbles.

Usage

opts_tbl_df(
  constructor = c("tibble", "tribble", "next", "list"),
  ..., trailing_comma = TRUE
)
opts_ts

Arguments

constructor String. Name of the function used to construct the environment, see Details section.

... Should not be used. Forces passing arguments by name.

trailing_comma Boolean, whether to leave a trailing comma at the end of the constructor call

details

Depending on constructor, we construct the object as follows:

- "tibble" (default): Wrap the column definitions in a tibble::tibble() call.
- "tribble": We build the object using tibble::tribble() if possible, and fall back to tibble::tibble().
- "next": Use the constructor for the next supported class. Call .class2() on the object to see in which order the methods will be tried.
- "list": Use list() and treat the class as a regular attribute.

Value

An object of class <constructive_options/constructive_options_tbl_df>

opts_ts Constructive options for time-series objets

Description

Depending on constructor, we construct the environment as follows:

- "ts": We use ts()
- "next": Use the constructor for the next supported class. Call .class2() on the object to see in which order the methods will be tried. This will usually be equivalent to "atomic"
- "atomic": We define as an atomic vector and repair attributes

Usage

opts_ts(constructor = c("ts", "next", "atomic"), ...)

Arguments

constructor String. Name of the function used to construct the environment.

... Should not be used. Forces passing arguments by name.

Value

An object of class <constructive_options/constructive_options_options_environment>
opts_vctrs_list_of  Constructive options for class 'data.table'

Description
These options will be used on objects of class 'data.table'.

Usage
opts_vctrs_list_of(constructor = c("list_of", "list"), ...)

Arguments
constructor  String. Name of the function used to construct the environment, see Details section.
...  Should not be used. Forces passing arguments by name.

Details
Depending on constructor, we construct the object as follows:
- "list_of" (default): Wrap the column definitions in a list_of() call.
- "list": Use list() and treat the class as a regular attribute.

Value
An object of class <constructive_options/constructive_options_data.table>

opts_weakref  Constructive options for the class weakref

Description
These options will be used on objects of type weakref. weakref objects are rarely encountered and there is no base R function to create them. However rlang has a new_weakref function that we can use.

Usage
opts_weakref(constructor = c("new_weakref"), ...)

Arguments
constructor  String. Name of the constructor.
...  Should not be used. Forces passing arguments by name.
Value

An object of class `<constructive_options/constructive_options_array>`
Index

* datasets
  constructors, 15
  .cstr_apply, 3
  .cstr_combine_errors, 4
  .cstr_construct, 6
  .cstr_fetch_opts, 6
  .cstr_match_constructor, 7
  .cstr_options, 7
  .cstr_pipe, 8
  .cstr_register_constructors, 8
  .cstr_repair_attributes, 9
  .cstr_wrap, 10
  .env, 10
  .xptr, 11

compare_options, 11
construct, 12
construct_diff, 15
construct_dump, 17
construct_issues, 17
construct_multi (construct), 12
construct_signature, 18
constructors, 15
custom-constructors, 18
defused function call, 5
deparse_call, 19

Formatting messages with cli, 5
Including contextual information with error chains, 5
Including function calls in error messages, 5

local_use_cli(), 5
opts_array, 13, 20
opts_AsIs, 13, 20
opts_atomic, 13, 21
opts_classGeneratorFunction, 13, 23

opts_classPrototypeDef, 13, 23
opts_classRepresentation, 13, 24
opts_constructive_options, 13, 24
 opts_data.frame, 14, 25
 opts_data.table, 14, 26
 opts_Date, 13, 27
 opts_dm, 14, 28
 opts_dots, 14, 28
 opts_environment, 14, 29
 opts_externalptr, 14, 31
 opts_factor, 14, 32
 opts_formula, 14, 33
 opts_function, 14, 34
 opts_grouped_df, 14, 35
 opts_language, 14, 35
 opts_Layer, 13, 36
 opts_list, 14, 37
 opts_matrix, 14, 38
 opts_mts, 14, 39
 opts_numeric_version, 14, 39
 opts_ordered, 14, 40
 opts_package_version, 14, 41
 opts_pairlist, 14, 41
 opts_POSIXct, 13, 42
 opts_POSIXlt, 13, 43
 opts_quosure, 14, 43
 opts_quosures, 14, 44
 opts_R_system_version, 13, 45
 opts_rowwise_df, 14, 45
 opts_S4, 13, 46
 opts_tbl_df, 14, 46
 opts_ts, 14, 47
 opts_vctrs_list_of, 14, 48
 opts_weakref, 14, 48

trace_back(), 5
try_fetch(), 5
tryCatch(), 5