Package ‘rlang’

April 11, 2025

Version 1.1.6

Title Functions for Base Types and Core R and 'Tidyverse' Features

Description A toolbox for working with base types, core R features like the condition system, and core 'Tidyverse' features like tidy evaluation.

License MIT + file LICENSE

ByteCompile true

Biarch true

Depends R (>= 3.5.0)

Imports utils

Suggests cli (>= 3.1.0), covr, crayon, desc, fs, glue, knitr, magrittr, methods, pillar, pkgload, rmarkdown, stats, testthat (>= 3.2.0), tibble, usethis, vctrs (>= 0.2.3), withr

Enhances winch

Encoding UTF-8

RoxygenNote 7.3.2

URL https://rlang.r-lib.org, https://github.com/r-lib/rlang

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

Config/build/compilation-database true

Config/testthat/edition 3

Config/Needs/website dplyr, tidyverse/tidytemplate

NeedsCompilation yes

Author Lionel Henry [aut, cre],
Hadley Wickham [aut],
mikefc [cph] (Hash implementation based on Mike's xxhashlite),
Yann Collet [cph] (Author of the embedded xxHash library),
Posit, PBC [cph, fnd]

Maintainer Lionel Henry <lionel@posit.co>

Repository CRAN

Date/Publication 2025-04-11 08:40:10 UTC

Contents

`abort` *Signal an error, warning, or message*

Description

These functions are equivalent to base functions `base::stop()`, `base::warning()`, and `base::message()`. They signal a condition (an error, warning, or message respectively) and make it easy to supply condition metadata:

- Supply `class` to create a classed condition that can be caught or handled selectively, allowing for finer-grained error handling.

- Supply metadata with named ... arguments. This data is stored in the condition object and can be examined by handlers.
- Supply call to inform users about which function the error occurred in.
- Supply another condition as parent to create a chained condition.

Certain components of condition messages are formatted with unicode symbols and terminal colours by default. These aspects can be customised, see Customising condition messages.

Usage

```
abort(
  message = NULL,
  class = NULL,
  ...,
  call,
  body = NULL,
  footer = NULL,
  trace = NULL,
  parent = NULL,
  use_cli_format = NULL,
  .inherit = TRUE,
  .internal = FALSE,
  .file = NULL,
  .frame = caller_env(),
  .trace_bottom = NULL,
  .subclass = deprecated()
)

warn(
  message = NULL,
  class = NULL,
  ...,
  body = NULL,
  footer = NULL,
  parent = NULL,
  use_cli_format = NULL,
  .inherit = NULL,
  .frequency = c("always", "regularly", "once"),
  .frequency_id = NULL,
  .subclass = deprecated()
)

inform(
  message = NULL,
  class = NULL,
  ...,
  body = NULL,
  footer = NULL,
  parent = NULL,
```

```
  use_cli_format = NULL,
  .inherit = NULL,
  .file = NULL,
  .frequency = c("always", "regularly", "once"),
  .frequency_id = NULL,
  .subclass = deprecated()
)

signal(message = "", class, ..., .subclass = deprecated())

reset_warning_verbosity(id)

reset_message_verbosity(id)
```

Arguments

`message` The message to display, formatted as a **bulleted list**. The first element is displayed as an *alert* bullet prefixed with ! by default. Elements named `"*"`, `"i"`, `"v"`, `"x"`, and `"!"` are formatted as regular, info, success, failure, and error bullets respectively. See Formatting messages with cli for more about bulleted messaging.

If a message is not supplied, it is expected that the message is generated **lazily** through `cnd_header()` and `cnd_body()` methods. In that case, `class` must be supplied. Only `inform()` allows empty messages as it is occasionally useful to build user output incrementally.

If a function, it is stored in the `header` field of the error condition. This acts as a `cnd_header()` method that is invoked lazily when the error message is displayed.

`class` Subclass of the condition.

`...` Additional data to be stored in the condition object. If you supply condition fields, you should usually provide a `class` argument. You may consider prefixing condition fields with the name of your package or organisation to prevent name collisions.

`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.

`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.

.inherit
: Whether the condition inherits from parent according to cnd_inherits() and try_fetch(). By default, parent conditions of higher severity are not inherited. For instance an error chained to a warning is not inherited to avoid unexpectedly catching an error downgraded to a warning.

.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.

.subclass
: **[Deprecated]** This argument was renamed to class in rlang 0.4.2 for consistency with our conventions for class constructors documented in https://adv-r.hadley.nz/s3.html#s3-subclassing.

.frequency
: How frequently should the warning or message be displayed? By default ("always") it is displayed at each time. If "regularly", it is displayed once every 8 hours. If "once", it is displayed once per session.

.frequency_id
: A unique identifier for the warning or message. This is used when .frequency is supplied to recognise recurring conditions. This argument must be supplied if .frequency is not set to "always".

id
: The identifying string of the condition that was supplied as .frequency_id to warn() or inform().

Details

- abort() throws subclassed errors, see "rlang_error".
- warn() temporarily set the warning.length global option to the maximum value (8170), unless that option has been changed from the default value. The default limit (1000 characters) is especially easy to hit when the message contains a lot of ANSI escapes, as created by the crayon or cli packages

Error prefix

As with base::stop(), errors thrown with abort() are prefixed with "Error: ". Calls and source references are included in the prefix, e.g. "Error in my_function() at myfile.R:1:2:". There are a few cosmetic differences:

- The call is stripped from its arguments to keep it simple. It is then formatted using the cli package if available.
- A line break between the prefix and the message when the former is too long. When a source location is included, a line break is always inserted.

If your throwing code is highly structured, you may have to explicitly inform abort() about the relevant user-facing call to include in the prefix. Internal helpers are rarely relevant to end users. See the call argument of abort().

Backtrace

abort() saves a backtrace in the trace component of the error condition. You can print a simplified backtrace of the last error by calling last_error() and a full backtrace with summary(last_error()). Learn how to control what is displayed when an error is thrown with rlang_backtrace_on_error.

Muffling and silencing conditions

Signalling a condition with inform() or warn() displays a message in the console. These messages can be muffled as usual with base::suppressMessages() or base::suppressWarnings().

inform() and warn() messages can also be silenced with the global options rlib_message_verbosity and rlib_warning_verbosity. These options take the values:

- "default": Verbose unless the .frequency argument is supplied.
- "verbose": Always verbose.
- "quiet": Always quiet.

When set to quiet, the message is not displayed and the condition is not signalled.

stdout and stderr

By default, abort() and inform() print to standard output in interactive sessions. This allows rlang to be in control of the appearance of messages in IDEs like RStudio.

There are two situations where messages are streamed to stderr:

- In non-interactive sessions, messages are streamed to standard error so that R scripts can easily filter them out from normal output by redirecting stderr.

- If a sink is active (either on output or on messages) messages are always streamd to `stderr`.

These exceptions ensure consistency of behaviour in interactive and non-interactive sessions, and when sinks are active.

See Also

- Including function calls in error messages
- Including contextual information with error chains

Examples

```
# These examples are guarded to avoid throwing errors
if (FALSE) {

# Signal an error with a message just like stop():
abort("The error message.")


# Unhandled errors are saved automatically by `abort()` and can be
# retrieved with `last_error()`. The error prints with a simplified
# backtrace:
f <- function() try(g())
g <- function() evalq(h())
h <- function() abort("Tilt.")
last_error()

# Use `summary()` to print the full backtrace and the condition fields:
summary(last_error())


# Give a class to the error:
abort("The error message", "mypkg_bad_error")

# This allows callers to handle the error selectively
tryCatch(
  mypkg_function(),
  mypkg_bad_error = function(err) {
    warn(conditionMessage(err)) # Demote the error to a warning
    NA                          # Return an alternative value
  }
)

# You can also specify metadata that will be stored in the condition:
abort("The error message.", "mypkg_bad_error", data = 1:10)

# This data can then be consulted by user handlers:
tryCatch(
  mypkg_function(),
  mypkg_bad_error = function(err) {
    # Compute an alternative return value with the data:
    recover_error(err$data)
```

```
  }
)


# If you call low-level APIs it may be a good idea to create a
# chained error with the low-level error wrapped in a more
# user-friendly error. Use `try_fetch()` to fetch errors of a given
# class and rethrow them with the `parent` argument of `abort()`:
file <- "http://foo.bar/baz"
try(
  try_fetch(
    download(file),
    error = function(err) {
      msg <- sprintf("Can't download `%s`", file)
      abort(msg, parent = err)
  })
)

# You can also hard-code the call when it's not easy to
# forward it from the caller
 f <- function() {
  abort("my message", call = call("my_function"))
}
g <- function() {
  f()
}
# Shows that the error occured in `my_function()`
try(g())

}
```

args_error_context *Documentation anchor for error arguments*

Description

Use `@inheritParams rlang::args_error_context` in your package to document `arg` and `call` arguments (or equivalently their prefixed versions `error_arg` and `error_call`).

- `arg` parameters should be formatted as argument (e.g. using cli's `.arg` specifier) and included in error messages. See also `caller_arg()`.
- `call` parameters should be included in error conditions in a field named `call`. An easy way to do this is by passing a `call` argument to `abort()`. See also `local_error_call()`.

Arguments

`arg`	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.
`error_arg`	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.

call	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.
error_call	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.

`arg_match` *Match an argument to a character vector*

Description

This is equivalent to `base::match.arg()` with a few differences:

- Partial matches trigger an error.
- Error messages are a bit more informative and obey the tidyverse standards.

`arg_match()` derives the possible values from the caller function.

`arg_match0()` is a bare-bones version if performance is at a premium. It requires a string as `arg` and explicit character `values`. For convenience, `arg` may also be a character vector containing every element of `values`, possibly permuted. In this case, the first element of `arg` is used.

Usage

```
arg_match(
  arg,
  values = NULL,
  ...,
  multiple = FALSE,
  error_arg = caller_arg(arg),
  error_call = caller_env()
)

arg_match0(arg, values, arg_nm = caller_arg(arg), error_call = caller_env())
```

Arguments

arg	A symbol referring to an argument accepting strings.
values	A character vector of possible values that `arg` can take.
...	These dots are for future extensions and must be empty.
multiple	Whether `arg` may contain zero or several values.
error_arg	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.
error_call	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.
arg_nm	Same as `error_arg`.

Value

The string supplied to arg.

See Also

check_required()

Examples

```
fn <- function(x = c("foo", "bar")) arg_match(x)
fn("bar")

# Throws an informative error for mismatches:
try(fn("b"))
try(fn("baz"))

# Use the bare-bones version with explicit values for speed:
arg_match0("bar", c("foo", "bar", "baz"))

# For convenience:
fn1 <- function(x = c("bar", "baz", "foo")) fn3(x)
fn2 <- function(x = c("baz", "bar", "foo")) fn3(x)
fn3 <- function(x) arg_match0(x, c("foo", "bar", "baz"))
fn1()
fn2("bar")
try(fn3("zoo"))
```

---

as_box *Convert object to a box*

Description

- as_box() boxes its input only if it is not already a box. The class is also checked if supplied.
- as_box_if() boxes its input only if it not already a box, or if the predicate .p returns TRUE.

Usage

```
as_box(x, class = NULL)

as_box_if(.x, .p, .class = NULL, ...)
```

Arguments

x, .x	An R object.
class, .class	A box class. If the input is already a box of that class, it is returned as is. If the input needs to be boxed, class is passed to new_box().
.p	A predicate function.
...	Arguments passed to .p.

`as_data_mask` *Create a data mask*

Description

A data mask is an environment (or possibly multiple environments forming an ancestry) containing user-supplied objects. Objects in the mask have precedence over objects in the environment (i.e. they mask those objects). Many R functions evaluate quoted expressions in a data mask so these expressions can refer to objects within the user data.

These functions let you construct a tidy eval data mask manually. They are meant for developers of tidy eval interfaces rather than for end users.

Usage

```
as_data_mask(data)

as_data_pronoun(data)

new_data_mask(bottom, top = bottom)
```

Arguments

`data`	A data frame or named vector of masking data.
`bottom`	The environment containing masking objects if the data mask is one environment deep. The bottom environment if the data mask comprises multiple environment. If you haven't supplied `top`, this **must** be an environment that you own, i.e. that you have created yourself.
`top`	The last environment of the data mask. If the data mask is only one environment deep, `top` should be the same as `bottom`. This **must** be an environment that you own, i.e. that you have created yourself. The parent of `top` will be changed by the tidy eval engine and should be considered undetermined. Never make assumption about the parent of `top`.

Value

A data mask that you can supply to `eval_tidy()`.

Why build a data mask?

Most of the time you can just call `eval_tidy()` with a list or a data frame and the data mask will be constructed automatically. There are three main use cases for manual creation of data masks:

- When `eval_tidy()` is called with the same data in a tight loop. Because there is some overhead to creating tidy eval data masks, constructing the mask once and reusing it for subsequent evaluations may improve performance.

- When several expressions should be evaluated in the exact same environment because a quoted expression might create new objects that can be referred in other quoted expressions evaluated at a later time. One example of this is `tibble::lst()` where new columns can refer to previous ones.
- When your data mask requires special features. For instance the data frame columns in dplyr data masks are implemented with active bindings.

Building your own data mask

Unlike `base::eval()` which takes any kind of environments as data mask, `eval_tidy()` has specific requirements in order to support quosures. For this reason you can't supply bare environments.

There are two ways of constructing an rlang data mask manually:

- `as_data_mask()` transforms a list or data frame to a data mask. It automatically installs the data pronoun `.data`.
- `new_data_mask()` is a bare bones data mask constructor for environments. You can supply a bottom and a top environment in case your data mask comprises multiple environments (see section below).

  Unlike `as_data_mask()` it does not install the `.data` pronoun so you need to provide one yourself. You can provide a pronoun constructed with `as_data_pronoun()` or your own pronoun class.

  `as_data_pronoun()` will create a pronoun from a list, an environment, or an rlang data mask. In the latter case, the whole ancestry is looked up from the bottom to the top of the mask. Functions stored in the mask are bypassed by the pronoun.

Once you have built a data mask, simply pass it to `eval_tidy()` as the `data` argument. You can repeat this as many times as needed. Note that any objects created there (perhaps because of a call to <-) will persist in subsequent evaluations.

Top and bottom of data mask

In some cases you'll need several levels in your data mask. One good reason is when you include functions in the mask. It's a good idea to keep data objects one level lower than function objects, so that the former cannot override the definitions of the latter (see examples).

In that case, set up all your environments and keep track of the bottom child and the top parent. You'll need to pass both to `new_data_mask()`.

Note that the parent of the top environment is completely undetermined, you shouldn't expect it to remain the same at all times. This parent is replaced during evaluation by `eval_tidy()` to one of the following environments:

- The default environment passed as the `env` argument of `eval_tidy()`.
- The environment of the current quosure being evaluated, if applicable.

Consequently, all masking data should be contained between the bottom and top environment of the data mask.

Examples

```
# Evaluating in a tidy evaluation environment enables all tidy
# features:
mask <- as_data_mask(mtcars)
eval_tidy(quo(letters), mask)

# You can install new pronouns in the mask:
mask$.pronoun <- as_data_pronoun(list(foo = "bar", baz = "bam"))
eval_tidy(quo(.pronoun$foo), mask)

# In some cases the data mask can leak to the user, for example if
# a function or formula is created in the data mask environment:
cyl <- "user variable from the context"
fn <- eval_tidy(quote(function() cyl), mask)
fn()

# If new objects are created in the mask, they persist in the
# subsequent calls:
eval_tidy(quote(new <- cyl + am), mask)
eval_tidy(quote(new * 2), mask)


# In some cases your data mask is a whole chain of environments
# rather than a single environment. You'll have to use
# `new_data_mask()` and let it know about the bottom of the mask
# (the last child of the environment chain) and the topmost parent.

# A common situation where you'll want a multiple-environment mask
# is when you include functions in your mask. In that case you'll
# put functions in the top environment and data in the bottom. This
# will prevent the data from overwriting the functions.
top <- new_environment(list(`+` = base::paste, c = base::paste))

# Let's add a middle environment just for sport:
middle <- env(top)

# And finally the bottom environment containing data:
bottom <- env(middle, a = "a", b = "b", c = "c")

# We can now create a mask by supplying the top and bottom
# environments:
mask <- new_data_mask(bottom, top = top)

# This data mask can be passed to eval_tidy() instead of a list or
# data frame:
eval_tidy(quote(a + b + c), data = mask)

# Note how the function `c()` and the object `c` are looked up
# properly because of the multi-level structure:
eval_tidy(quote(c(a, b, c)), data = mask)

# new_data_mask() does not create data pronouns, but
```

```
# data pronouns can be added manually:
mask$.fns <- as_data_pronoun(top)

# The `.data` pronoun should generally be created from the
# mask. This will ensure data is looked up throughout the whole
# ancestry. Only non-function objects are looked up from this
# pronoun:
mask$.data <- as_data_pronoun(mask)
mask$.data$c

# Now we can reference values with the pronouns:
eval_tidy(quote(c(.data$a, .data$b, .data$c)), data = mask)
```

---

as_environment *Coerce to an environment*

Description

`as_environment()` coerces named vectors (including lists) to an environment. The names must be unique. If supplied an unnamed string, it returns the corresponding package environment (see `pkg_env()`).

Usage

```
as_environment(x, parent = NULL)
```

Arguments

`x`	An object to coerce.
`parent`	A parent environment, `empty_env()` by default. This argument is only used when `x` is data actually coerced to an environment (as opposed to data representing an environment, like `NULL` representing the empty environment).

Details

If `x` is an environment and `parent` is not `NULL`, the environment is duplicated before being set a new parent. The return value is therefore a different environment than `x`.

Examples

```
# Coerce a named vector to an environment:
env <- as_environment(mtcars)

# By default it gets the empty environment as parent:
identical(env_parent(env), empty_env())


# With strings it is a handy shortcut for pkg_env():
as_environment("base")
```

```
as_environment("rlang")

# With NULL it returns the empty environment:
as_environment(NULL)
```

---

as_function *Convert to function*

Description

as_function() transforms a one-sided formula into a function. This powers the lambda syntax in packages like purrr.

Usage

```
as_function(
  x,
  env = global_env(),
  ...,
  arg = caller_arg(x),
  call = caller_env()
)

is_lambda(x)
```

Arguments

x A function or formula.

If a **function**, it is used as is.

If a **formula**, e.g. ~ .x + 2, it is converted to a function with up to two arguments: .x (single argument) or .x and .y (two arguments). The . placeholder can be used instead of .x. This allows you to create very compact anonymous functions (lambdas) with up to two inputs. Functions created from formulas have a special class. Use is_lambda() to test for it.

If a **string**, the function is looked up in env. Note that this interface is strictly for user convenience because of the scoping issues involved. Package developers should avoid supplying functions by name and instead supply them by value.

env Environment in which to fetch the function in case x is a string.

... These dots are for future extensions and must be empty.

arg An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.

call The execution environment of a currently running function, e.g. caller_env(). The function will be mentioned in error messages as the source of the error. See the call argument of abort() for more information.

Examples

```
f <- as_function(~ .x + 1)
f(10)

g <- as_function(~ -1 * .)
g(4)

h <- as_function(~ .x - .y)
h(6, 3)

# Functions created from a formula have a special class:
is_lambda(f)
is_lambda(as_function(function() "foo"))
```

---

as_label *Create a default name for an R object*

Description

as_label() transforms R objects into a short, human-readable description. You can use labels to:

- Display an object in a concise way, for example to labellise axes in a graphical plot.
- Give default names to columns in a data frame. In this case, labelling is the first step before name repair.

See also as_name() for transforming symbols back to a string. Unlike as_label(), as_name() is a well defined operation that guarantees the roundtrip symbol -> string -> symbol.

In general, if you don't know for sure what kind of object you're dealing with (a call, a symbol, an unquoted constant), use as_label() and make no assumption about the resulting string. If you know you have a symbol and need the name of the object it refers to, use as_name(). For instance, use as_label() with objects captured with enquo() and as_name() with symbols captured with ensym().

Usage

```
as_label(x)
```

Arguments

x	An object.

Transformation to string

- Quosures are squashed before being labelled.
- Symbols are transformed to string with as_string().
- Calls are abbreviated.
- Numbers are represented as such.
- Other constants are represented by their type, such as <dbl> or <data.frame>.

See Also

as_name() for transforming symbols back to a string deterministically.

Examples

```
# as_label() is useful with quoted expressions:
as_label(expr(foo(bar)))

as_label(expr(foobar))

# It works with any R object. This is also useful for quoted
# arguments because the user might unquote constant objects:
as_label(1:3)

as_label(base::list)
```

---

as_name *Extract names from symbols*

Description

as_name() converts symbols to character strings. The conversion is deterministic. That is, the roundtrip symbol -> name -> symbol always gives the same result.

- Use as_name() when you need to transform a symbol to a string to *refer* to an object by its name.
- Use as_label() when you need to transform any kind of object to a string to *represent* that object with a short description.

Usage

```
as_name(x)
```

Arguments

x	A string or symbol, possibly wrapped in a quosure. If a string, the attributes are removed, if any.

Details

rlang::as_name() is the *opposite* of base::as.name(). If you're writing base R code, we recommend using base::as.symbol() which is an alias of as.name() that follows a more modern terminology (R types instead of S modes).

Value

A character vector of length 1.

See Also

`as_label()` for converting any object to a single string suitable as a label. `as_string()` for a lower-level version that doesn't unwrap quosures.

Examples

```
# Let's create some symbols:
foo <- quote(foo)
bar <- sym("bar")

# as_name() converts symbols to strings:
foo
as_name(foo)

typeof(bar)
typeof(as_name(bar))

# as_name() unwraps quosured symbols automatically:
as_name(quo(foo))
```

---

as_string — *Cast symbol to string*

Description

`as_string()` converts symbols to character strings.

Usage

```
as_string(x)
```

Arguments

x	A string or symbol. If a string, the attributes are removed, if any.

Value

A character vector of length 1.

Unicode tags

Unlike `base::as.symbol()` and `base::as.name()`, `as_string()` automatically transforms unicode tags such as `"<U+5E78>"` to the proper UTF-8 character. This is important on Windows because:

- R on Windows has no UTF-8 support, and uses native encoding instead.
- The native encodings do not cover all Unicode characters. For example, Western encodings do not support CKJ characters.

- When a lossy UTF-8 -> native transformation occurs, uncovered characters are transformed to an ASCII unicode tag like `"<U+5E78>"`.
- Symbols are always encoded in native. This means that transforming the column names of a data frame to symbols might be a lossy operation.
- This operation is very common in the tidyverse because of data masking APIs like dplyr where data frames are transformed to environments. While the names of a data frame are stored as a character vector, the bindings of environments are stored as symbols.

Because it reencodes the ASCII unicode tags to their UTF-8 representation, the string -> symbol -> string roundtrip is more stable with `as_string()`.

See Also

`as_name()` for a higher-level variant of `as_string()` that automatically unwraps quosures.

Examples

```
# Let's create some symbols:
foo <- quote(foo)
bar <- sym("bar")

# as_string() converts symbols to strings:
foo
as_string(foo)

typeof(bar)
typeof(as_string(bar))
```

---

bare-type-predicates *Bare type predicates*

Description

These predicates check for a given type but only return `TRUE` for bare R objects. Bare objects have no class attributes. For example, a data frame is a list, but not a bare list.

Usage

```
is_bare_list(x, n = NULL)

is_bare_atomic(x, n = NULL)

is_bare_vector(x, n = NULL)

is_bare_double(x, n = NULL)

is_bare_complex(x, n = NULL)
```

```
is_bare_integer(x, n = NULL)

is_bare_numeric(x, n = NULL)

is_bare_character(x, n = NULL)

is_bare_logical(x, n = NULL)

is_bare_raw(x, n = NULL)

is_bare_string(x, n = NULL)

is_bare_bytes(x, n = NULL)
```

Arguments

x	Object to be tested.
n	Expected length of a vector.

Details

- The predicates for vectors include the n argument for pattern-matching on the vector length.
- Like is_atomic() and unlike base R is.atomic() for R < 4.4.0, is_bare_atomic() does not return TRUE for NULL. Starting in R 4.4.0, is.atomic(NULL) returns FALSE.
- Unlike base R is.numeric(), is_bare_double() only returns TRUE for floating point numbers.

See Also

type-predicates, scalar-type-predicates

---

box — *Box a value*

Description

new_box() is similar to base::I() but it protects a value by wrapping it in a scalar list rather than by adding an attribute. unbox() retrieves the boxed value. is_box() tests whether an object is boxed with optional class. as_box() ensures that a value is wrapped in a box. as_box_if() does the same but only if the value matches a predicate.

Usage

```
new_box(.x, class = NULL, ...)

is_box(x, class = NULL)

unbox(box)
```

Arguments

`class`	For `new_box()`, an additional class for the boxed value (in addition to `rlang_box`). For `is_box()`, a class or vector of classes passed to `inherits_all()`.
`...`	Additional attributes passed to `base::structure()`.
`x, .x`	An R object.
`box`	A boxed value to unbox.

Examples

```
boxed <- new_box(letters, "mybox")
is_box(boxed)
is_box(boxed, "mybox")
is_box(boxed, "otherbox")

unbox(boxed)

# as_box() avoids double-boxing:
boxed2 <- as_box(boxed, "mybox")
boxed2
unbox(boxed2)

# Compare to:
boxed_boxed <- new_box(boxed, "mybox")
boxed_boxed
unbox(unbox(boxed_boxed))

# Use `as_box_if()` with a predicate if you need to ensure a box
# only for a subset of values:
as_box_if(NULL, is_null, "null_box")
as_box_if("foo", is_null, "null_box")
```

`bytes-class` *Human readable memory sizes*

Description

Construct, manipulate and display vectors of byte sizes. These are numeric vectors, so you can compare them numerically, but they can also be compared to human readable values such as '10MB'.

- `parse_bytes()` takes a character vector of human-readable bytes and returns a structured bytes vector.
- `as_bytes()` is a generic conversion function for objects representing bytes.

Note: A `bytes()` constructor will be exported soon.

Usage

```
as_bytes(x)

parse_bytes(x)
```

Arguments

x	A numeric or character vector. Character representations can use shorthand sizes (see examples).

Details

These memory sizes are always assumed to be base 1000, rather than 1024.

Examples

```
parse_bytes("1")
parse_bytes("1K")
parse_bytes("1Kb")
parse_bytes("1KiB")
parse_bytes("1MB")

parse_bytes("1KB") < "1MB"

sum(parse_bytes(c("1MB", "5MB", "500KB")))
```

---

call2 *Create a call*

Description

Quoted function calls are one of the two types of symbolic objects in R. They represent the action of calling a function, possibly with arguments. There are two ways of creating a quoted call:

- By quoting it. Quoting prevents functions from being called. Instead, you get the description of the function call as an R object. That is, a quoted function call.
- By constructing it with `base::call()`, `base::as.call()`, or `call2()`. In this case, you pass the call elements (the function to call and the arguments to call it with) separately.

See section below for the difference between `call2()` and the base constructors.

Usage

```
call2(.fn, ..., .ns = NULL)
```

Arguments

.fn	Function to call. Must be a callable object: a string, symbol, call, or a function.
...	<dynamic> Arguments for the function call. Empty arguments are preserved.
.ns	Namespace with which to prefix `.fn`. Must be a string or symbol.

Difference with base constructors

`call2()` is more flexible than `base::call()`:

- The function to call can be a string or a callable object: a symbol, another call (e.g. a `$` or `[[` call), or a function to inline. `base::call()` only supports strings and you need to use `base::as.call()` to construct a call with a callable object.

  ```
  call2(list, 1, 2)

  as.call(list(list, 1, 2))
  ```

- The `.ns` argument is convenient for creating namespaced calls.

  ```
  call2("list", 1, 2, .ns = "base")

  # Equivalent to
  ns_call <- call("::", as.symbol("list"), as.symbol("base"))
  as.call(list(ns_call, 1, 2))
  ```

- `call2()` has dynamic dots support. You can splice lists of arguments with `!!!` or unquote an argument name with glue syntax.

  ```
  args <- list(na.rm = TRUE, trim = 0)

  call2("mean", 1:10, !!!args)

  # Equivalent to
  as.call(c(list(as.symbol("mean"), 1:10), args))
  ```

Caveats of inlining objects in calls

`call2()` makes it possible to inline objects in calls, both in function and argument positions. Inlining an object or a function has the advantage that the correct object is used in all environments. If all components of the code are inlined, you can even evaluate in the empty environment.

However inlining also has drawbacks. It can cause issues with NSE functions that expect symbolic arguments. The objects may also leak in representations of the call stack, such as `traceback()`.

See Also

`call_modify()`

Examples

```
# fn can either be a string, a symbol or a call
call2("f", a = 1)
call2(quote(f), a = 1)
call2(quote(f()), a = 1)

#' Can supply arguments individually or in a list
call2(quote(f), a = 1, b = 2)
call2(quote(f), !!!list(a = 1, b = 2))
```

```
# Creating namespaced calls is easy:
call2("fun", arg = quote(baz), .ns = "mypkg")

# Empty arguments are preserved:
call2("[", quote(x), , drop = )
```

---

caller_arg *Find the caller argument for error messages*

Description

`caller_arg()` is a variant of `substitute()` or `ensym()` for arguments that reference other arguments. Unlike `substitute()` which returns an expression, `caller_arg()` formats the expression as a single line string which can be included in error messages.

- When included in an error message, the resulting label should generally be formatted as argument, for instance using the `.arg` in the cli package.
- Use `@inheritParams rlang::args_error_context` to document an `arg` or `error_arg` argument that takes `error_arg()` as default.

Arguments

arg	An argument name in the current function.

Examples

```
arg_checker <- function(x, arg = caller_arg(x), call = caller_env()) {
  cli::cli_abort("{.arg {arg}} must be a thingy.", arg = arg, call = call)
}

my_function <- function(my_arg) {
  arg_checker(my_arg)
}

try(my_function(NULL))
```

---

call_args *Extract arguments from a call*

Description

Extract arguments from a call

Usage

```
call_args(call)

call_args_names(call)
```

Arguments

`call`	A defused call.

Value

A named list of arguments.

See Also

`fn_fmls()` and `fn_fmls_names()`

Examples

```
call <- quote(f(a, b))

# Subsetting a call returns the arguments converted to a language
# object:
call[-1]

# On the other hand, call_args() returns a regular list that is
# often easier to work with:
str(call_args(call))

# When the arguments are unnamed, a vector of empty strings is
# supplied (rather than NULL):
call_args_names(call)
```

`call_inspect` *Inspect a call*

Description

This function is a wrapper around `base::match.call()`. It returns its own function call.

Usage

```
call_inspect(...)
```

Arguments

`...`	Arguments to display in the returned call.

Examples

```
# When you call it directly, it simply returns what you typed
call_inspect(foo(bar), "" %>% identity())

# Pass `call_inspect` to functionals like `lapply()` or `map()` to
# inspect the calls they create around the supplied function
lapply(1:3, call_inspect)
```

---

call_match *Match supplied arguments to function definition*

Description

`call_match()` is like `match.call()` with these differences:

- It supports matching missing argument to their defaults in the function definition.
- It requires you to be a little more specific in some cases. Either all arguments are inferred from the call stack or none of them are (see the Inference section).

Usage

```
call_match(
  call = NULL,
  fn = NULL,
  ...,
  defaults = FALSE,
  dots_env = NULL,
  dots_expand = TRUE
)
```

Arguments

`call`	A call. The arguments will be matched to `fn`.
`fn`	A function definition to match arguments to.
`...`	These dots must be empty.
`defaults`	Whether to match missing arguments to their defaults.
`dots_env`	An execution environment where to find dots. If supplied and dots exist in this environment, and if `call` includes `...`, the forwarded dots are matched to numbered dots (e.g. `..1`, `..2`, etc). By default this is set to the empty environment which means that `...` expands to nothing.
`dots_expand`	If `FALSE`, arguments passed through `...` will not be spliced into `call`. Instead, they are gathered in a pairlist and assigned to an argument named `...`. Gathering dots arguments is useful if you need to separate them from the other named arguments. Note that the resulting call is not meant to be evaluated since R does not support passing dots through a named argument, even if named `"..."`.

Inference from the call stack

When call is not supplied, it is inferred from the call stack along with fn and dots_env.

- call and fn are inferred from the calling environment: sys.call(sys.parent()) and sys.function(sys.parent())
- dots_env is inferred from the caller of the calling environment: caller_env(2).

If call is supplied, then you must supply fn as well. Also consider supplying dots_env as it is set to the empty environment when not inferred.

Examples

```
# `call_match()` supports matching missing arguments to their
# defaults
fn <- function(x = "default") fn
call_match(quote(fn()), fn)
call_match(quote(fn()), fn, defaults = TRUE)
```

---

call_modify *Modify the arguments of a call*

Description

If you are working with a user-supplied call, make sure the arguments are standardised with call_match() before modifying the call.

Usage

```
call_modify(
  .call,
  ...,
  .homonyms = c("keep", "first", "last", "error"),
  .standardise = NULL,
  .env = caller_env()
)
```

Arguments

.call	Can be a call, a formula quoting a call in the right-hand side, or a frame object from which to extract the call expression.
...	<dynamic> Named or unnamed expressions (constants, names or calls) used to modify the call. Use zap() to remove arguments. Empty arguments are preserved.
.homonyms	How to treat arguments with the same name. The default, "keep", preserves these arguments. Set .homonyms to "first" to only keep the first occurrences, to "last" to keep the last occurrences, and to "error" to raise an informative error and indicate what arguments have duplicated names.
.standardise, .env	Deprecated as of rlang 0.3.0. Please call call_match() manually.

Value

A quosure if `.call` is a quosure, a call otherwise.

Examples

```
call <- quote(mean(x, na.rm = TRUE))

# Modify an existing argument
call_modify(call, na.rm = FALSE)
call_modify(call, x = quote(y))

# Remove an argument
call_modify(call, na.rm = zap())

# Add a new argument
call_modify(call, trim = 0.1)

# Add an explicit missing argument:
call_modify(call, na.rm = )

# Supply a list of new arguments with `!!!`
newargs <- list(na.rm = zap(), trim = 0.1)
call <- call_modify(call, !!!newargs)
call

# Remove multiple arguments by splicing zaps:
newargs <- rep_named(c("na.rm", "trim"), list(zap()))
call <- call_modify(call, !!!newargs)
call


# Modify the `...` arguments as if it were a named argument:
call <- call_modify(call, ... = )
call

call <- call_modify(call, ... = zap())
call


# When you're working with a user-supplied call, standardise it
# beforehand in case it includes unmatched arguments:
user_call <- quote(matrix(x, nc = 3))
call_modify(user_call, ncol = 1)

# `call_match()` applies R's argument matching rules. Matching
# ensures you're modifying the intended argument.
user_call <- call_match(user_call, matrix)
user_call
call_modify(user_call, ncol = 1)


# By default, arguments with the same name are kept. This has
```

```
# subtle implications, for instance you can move an argument to
# last position by removing it and remapping it:
call <- quote(foo(bar = , baz))
call_modify(call, bar = zap(), bar = missing_arg())

# You can also choose to keep only the first or last homonym
# arguments:
args <-  list(bar = zap(), bar = missing_arg())
call_modify(call, !!!args, .homonyms = "first")
call_modify(call, !!!args, .homonyms = "last")
```

---

call_name *Extract function name or namespace of a call*

Description

`call_name()` and `call_ns()` extract the function name or namespace of *simple* calls as a string. They return `NULL` for complex calls.

- Simple calls: `foo()`, `bar::foo()`.
- Complex calls: `foo()()`, `bar::foo`, `foo$bar()`, `(function() NULL)()`.

The `is_call_simple()` predicate helps you determine whether a call is simple. There are two invariants you can count on:

1. If `is_call_simple(x)` returns `TRUE`, `call_name(x)` returns a string. Otherwise it returns `NULL`.
2. If `is_call_simple(x, ns = TRUE)` returns `TRUE`, `call_ns()` returns a string. Otherwise it returns `NULL`.

Usage

```
call_name(call)

call_ns(call)

is_call_simple(x, ns = NULL)
```

Arguments

`call`	A defused call.
`x`	An object to test.
`ns`	Whether call is namespaced. If `NULL`, `is_call_simple()` is insensitive to namespaces. If `TRUE`, `is_call_simple()` detects namespaced calls. If `FALSE`, it detects unnamespaced calls.

Value

The function name or namespace as a string, or `NULL` if the call is not named or namespaced.

Examples

```
# Is the function named?
is_call_simple(quote(foo()))
is_call_simple(quote(foo[[1]]()))

# Is the function namespaced?
is_call_simple(quote(list()), ns = TRUE)
is_call_simple(quote(base::list()), ns = TRUE)

# Extract the function name from quoted calls:
call_name(quote(foo(bar)))
call_name(quo(foo(bar)))

# Namespaced calls are correctly handled:
call_name(quote(base::matrix(baz)))

# Anonymous and subsetted functions return NULL:
call_name(quote(foo$bar()))
call_name(quote(foo[[bar]]()))
call_name(quote(foo()()))

# Extract namespace of a call with call_ns():
call_ns(quote(base::bar()))

# If not namespaced, call_ns() returns NULL:
call_ns(quote(bar()))
```

---

catch_cnd *Catch a condition*

Description

This is a small wrapper around `tryCatch()` that captures any condition signalled while evaluating its argument. It is useful for situations where you expect a specific condition to be signalled, for debugging, and for unit testing.

Usage

```
catch_cnd(expr, classes = "condition")
```

Arguments

`expr`	Expression to be evaluated with a catching condition handler.
`classes`	A character vector of condition classes to catch. By default, catches all conditions.

Value

A condition if any was signalled, `NULL` otherwise.

Examples

```
catch_cnd(10)
catch_cnd(abort("an error"))
catch_cnd(signal("my_condition", message = "a condition"))
```

`check_dots_empty` *Check that dots are empty*

Description

`...` can be inserted in a function signature to force users to fully name the details arguments. In this case, supplying data in `...` is almost always a programming error. This function checks that `...` is empty and fails otherwise.

Usage

```
check_dots_empty(
  env = caller_env(),
  error = NULL,
  call = caller_env(),
  action = abort
)
```

Arguments

`env`	Environment in which to look for `...`.
`error`	An optional error handler passed to `try_fetch()`. Use this e.g. to demote an error into a warning.
`call`	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.
`action`	**[Deprecated]**

Details

In packages, document `...` with this standard tag:

```
@inheritParams rlang::args_dots_empty
```

See Also

Other dots checking functions: `check_dots_unnamed()`, `check_dots_used()`

Examples

```
f <- function(x, ..., foofy = 8) {
  check_dots_empty()
  x + foofy
}

# This fails because `foofy` can't be matched positionally
try(f(1, 4))

# This fails because `foofy` can't be matched partially by name
try(f(1, foof = 4))

# Thanks to `...`, it must be matched exactly
f(1, foofy = 4)
```

---

check_dots_unnamed *Check that all dots are unnamed*

Description

In functions like `paste()`, named arguments in `...` are often a sign of misspelled argument names. Call `check_dots_unnamed()` to fail with an error when named arguments are detected.

Usage

```
check_dots_unnamed(
  env = caller_env(),
  error = NULL,
  call = caller_env(),
  action = abort
)
```

Arguments

env	Environment in which to look for `...`.
error	An optional error handler passed to `try_fetch()`. Use this e.g. to demote an error into a warning.
call	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.
action	**[Deprecated]**

See Also

Other dots checking functions: `check_dots_empty()`, `check_dots_used()`

Examples

```
f <- function(..., foofy = 8) {
  check_dots_unnamed()
  c(...)
}

f(1, 2, 3, foofy = 4)

try(f(1, 2, 3, foof = 4))
```

---

check_dots_used *Check that all dots have been used*

Description

When ... arguments are passed to a method, the method should match and use these arguments. If this isn't the case, this often indicates a programming error. Call `check_dots_used()` to fail with an error when unused arguments are detected.

Usage

```
check_dots_used(
  env = caller_env(),
  call = caller_env(),
  error = NULL,
  action = deprecated()
)
```

Arguments

`env`	Environment in which to look for ... and to set up handler.
`call`	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.
`error`	An optional error handler passed to `try_fetch()`. Use this e.g. to demote an error into a warning.
`action`	**[Deprecated]**

Details

In packages, document ... with this standard tag:

```
@inheritParams rlang::args_dots_used
```

`check_dots_used()` implicitly calls `on.exit()` to check that all elements of ... have been used when the function exits. If you use `on.exit()` elsewhere in your function, make sure to use `add = TRUE` so that you don't override the handler set up by `check_dots_used()`.

See Also

Other dots checking functions: check_dots_empty(), check_dots_unnamed()

Examples

```
f <- function(...) {
  check_dots_used()
  g(...)
}

g <- function(x, y, ...) {
  x + y
}
f(x = 1, y = 2)

try(f(x = 1, y = 2, z = 3))

try(f(x = 1, y = 2, 3, 4, 5))

# Use an `error` handler to handle the error differently.
# For instance to demote the error to a warning:
fn <- function(...) {
  check_dots_empty(
    error = function(cnd) {
      warning(cnd)
    }
  )
  "out"
}
fn()
```

---

check_exclusive — *Check that arguments are mutually exclusive*

Description

check_exclusive() checks that only one argument is supplied out of a set of mutually exclusive arguments. An informative error is thrown if multiple arguments are supplied.

Usage

```
check_exclusive(..., .require = TRUE, .frame = caller_env(), .call = .frame)
```

Arguments

...	Function arguments.
.require	Whether at least one argument must be supplied.

`.frame`	Environment where the arguments in `...` are defined.
`.call`	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.

Value

The supplied argument name as a string. If `.require` is `FALSE` and no argument is supplied, the empty string `""` is returned.

Examples

```
f <- function(x, y) {
  switch(
    check_exclusive(x, y),
    x = message("`x` was supplied."),
    y = message("`y` was supplied.")
  )
}

# Supplying zero or multiple arguments is forbidden
try(f())
try(f(NULL, NULL))

# The user must supply one of the mutually exclusive arguments
f(NULL)
f(y = NULL)


# With `.require` you can allow zero arguments
f <- function(x, y) {
  switch(
    check_exclusive(x, y, .require = FALSE),
    x = message("`x` was supplied."),
    y = message("`y` was supplied."),
    message("No arguments were supplied")
  )
}
f()
```

---

check_required — *Check that argument is supplied*

Description

Throws an error if x is missing.

Usage

```
check_required(x, arg = caller_arg(x), call = caller_env())
```

Arguments

x	A function argument. Must be a symbol.
arg	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.
call	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.

See Also

`arg_match()`

Examples

```
f <- function(x) {
  check_required(x)
}

# Fails because `x` is not supplied
try(f())

# Succeeds
f(NULL)
```

---

cnd_inherits — *Does a condition or its ancestors inherit from a class?*

Description

Like any R objects, errors captured with catchers like `tryCatch()` have a `class()` which you can test with `inherits()`. However, with chained errors, the class of a captured error might be different than the error that was originally signalled. Use `cnd_inherits()` to detect whether an error or any of its *parent* inherits from a class.

Whereas `inherits()` tells you whether an object is a particular kind of error, `cnd_inherits()` answers the question whether an object is a particular kind of error or has been caused by such an error.

Some chained conditions carry parents that are not inherited. See the `.inherit` argument of `abort()`, `warn()`, and `inform()`.

Usage

```
cnd_inherits(cnd, class)
```

Arguments

cnd	A condition to test.
class	A class passed to `inherits()`.

Capture an error with `cnd_inherits()`

Error catchers like `tryCatch()` and `try_fetch()` can only match the class of a condition, not the class of its parents. To match a class across the ancestry of an error, you'll need a bit of craftiness.

Ancestry matching can't be done with `tryCatch()` at all so you'll need to switch to `withCallingHandlers()`. Alternatively, you can use the experimental rlang function `try_fetch()` which is able to perform the roles of both `tryCatch()` and `withCallingHandlers()`.

`withCallingHandlers()`:

Unlike `tryCatch()`, `withCallingHandlers()` does not capture an error. If you don't explicitly jump with an *error* or a *value* throw, nothing happens.

Since we don't want to throw an error, we'll throw a value using `callCC()`:

```
f <- function() {
  parent <- error_cnd("bar", message = "Bar")
  abort("Foo", parent = parent)
}

cnd <- callCC(function(throw) {
  withCallingHandlers(
    f(),
    error = function(x) if (cnd_inherits(x, "bar")) throw(x)
  )
})

class(cnd)
#> [1] "rlang_error" "error"       "condition"
class(cnd$parent)
#> [1] "bar"         "rlang_error" "error"       "condition"
```

`try_fetch()`:

This pattern is easier with `try_fetch()`. Like `withCallingHandlers()`, it doesn't capture a matching error right away. Instead, it captures it only if the handler doesn't return a `zap()` value.

```
cnd <- try_fetch(
  f(),
  error = function(x) if (cnd_inherits(x, "bar")) x else zap()
)

class(cnd)
#> [1] "rlang_error" "error"       "condition"
class(cnd$parent)
#> [1] "bar"         "rlang_error" "error"       "condition"
```

Note that `try_fetch()` uses `cnd_inherits()` internally. This makes it very easy to match a parent condition:

```
cnd <- try_fetch(
  f(),
  bar = function(x) x
```

```
)

# This is the parent
class(cnd)
#> [1] "bar"         "rlang_error" "error"       "condition"
```

`cnd_message` *Build an error message from parts*

Description

`cnd_message()` assembles an error message from three generics:

- `cnd_header()`
- `cnd_body()`
- `cnd_footer()`

Methods for these generics must return a character vector. The elements are combined into a single string with a newline separator. Bullets syntax is supported, either through rlang (see `format_error_bullets()`), or through cli if the condition has `use_cli_format` set to `TRUE`.

The default method for the error header returns the `message` field of the condition object. The default methods for the body and footer return the the `body` and `footer` fields if any, or empty character vectors otherwise.

`cnd_message()` is automatically called by the `conditionMessage()` for rlang errors, warnings, and messages. Error classes created with `abort()` only need to implement header, body or footer methods. This provides a lot of flexibility for hierarchies of error classes, for instance you could inherit the body of an error message from a parent class while overriding the header and footer.

Usage

```
cnd_message(cnd, ..., inherit = TRUE, prefix = FALSE)

cnd_header(cnd, ...)

cnd_body(cnd, ...)

cnd_footer(cnd, ...)
```

Arguments

`cnd`	A condition object.
`...`	Arguments passed to methods.
`inherit`	Wether to include parent messages. Parent messages are printed with a "Caused by error:" prefix, even if `prefix` is `FALSE`.

`prefix`	Whether to print the full message, including the condition prefix (`Error:`, `Warning:`, `Message:`, or `Condition:`). The prefix mentions the `call` field if present, and the `srcref` info if present. If `cnd` has a `parent` field (i.e. the condition is chained), the parent messages are included in the message with a `Caused by` prefix.

Overriding header, body, and footer methods

Sometimes the contents of an error message depends on the state of your checking routine. In that case, it can be tricky to lazily generate error messages with `cnd_header()`, `cnd_body()`, and `cnd_footer()`: you have the choice between overspecifying your error class hierarchies with one class per state, or replicating the type-checking control flow within the `cnd_body()` method. None of these options are ideal.

A better option is to define `header`, `body`, or `footer` fields in your condition object. These can be a static string, a lambda-formula, or a function with the same signature as `cnd_header()`, `cnd_body()`, or `cnd_footer()`. These fields override the message generics and make it easy to generate an error message tailored to the state in which the error was constructed.

`cnd_signal` *Signal a condition object*

Description

`cnd_signal()` takes a condition as argument and emits the corresponding signal. The type of signal depends on the class of the condition:

- A message is signalled if the condition inherits from `"message"`. This is equivalent to signalling with `inform()` or `base::message()`.
- A warning is signalled if the condition inherits from `"warning"`. This is equivalent to signalling with `warn()` or `base::warning()`.
- An error is signalled if the condition inherits from `"error"`. This is equivalent to signalling with `abort()` or `base::stop()`.
- An interrupt is signalled if the condition inherits from `"interrupt"`. This is equivalent to signalling with `interrupt()`.

Usage

```
cnd_signal(cnd, ...)
```

Arguments

`cnd`	A condition object (see `cnd()`). If `NULL`, `cnd_signal()` returns without signalling a condition.
`...`	These dots are for future extensions and must be empty.

See Also

- `cnd_type()` to determine the type of a condition.
- `abort()`, `warn()` and `inform()` for creating and signalling structured R conditions in one go.
- `try_fetch()` for establishing condition handlers for particular condition classes.

Examples

```
# The type of signal depends on the class. If the condition
# inherits from "warning", a warning is issued:
cnd <- warning_cnd("my_warning_class", message = "This is a warning")
cnd_signal(cnd)

# If it inherits from "error", an error is raised:
cnd <- error_cnd("my_error_class", message = "This is an error")
try(cnd_signal(cnd))
```

---

done — *Box a final value for early termination*

Description

A value boxed with `done()` signals to its caller that it should stop iterating. Use it to shortcircuit a loop.

Usage

```
done(x)

is_done_box(x, empty = NULL)
```

Arguments

`x`	For `done()`, a value to box. For `is_done_box()`, a value to test.
`empty`	Whether the box is empty. If `NULL`, `is_done_box()` returns `TRUE` for all done boxes. If `TRUE`, it returns `TRUE` only for empty boxes. Otherwise it returns `TRUE` only for non-empty boxes.

Value

A boxed value.

Examples

```
done(3)

x <- done(3)
is_done_box(x)
```

dot-data   `.data` *and* `.env` *pronouns*

Description

The `.data` and `.env` pronouns make it explicit where to find objects when programming with data-masked functions.

```
m <- 10
mtcars %>% mutate(disp = .data$disp * .env$m)
```

- `.data` retrieves data-variables from the data frame.
- `.env` retrieves env-variables from the environment.

Because the lookup is explicit, there is no ambiguity between both kinds of variables. Compare:

```
disp <- 10
mtcars %>% mutate(disp = .data$disp * .env$disp)
mtcars %>% mutate(disp = disp * disp)
```

Note that `.data` is only a pronoun, it is not a real data frame. This means that you can't take its names or map a function over the contents of `.data`. Similarly, `.env` is not an actual R environment. For instance, it doesn't have a parent and the subsetting operators behave differently.

`.data` versus the magrittr pronoun `.`

In a magrittr pipeline, `.data` is not necessarily interchangeable with the magrittr pronoun `.`. With grouped data frames in particular, `.data` represents the current group slice whereas the pronoun `.` represents the whole data frame. Always prefer using `.data` in data-masked context.

Where does `.data` live?

The `.data` pronoun is automatically created for you by data-masking functions using the tidy eval framework. You don't need to import `rlang::.data` or use `library(rlang)` to work with this pronoun.

However, the `.data` object exported from rlang is useful to import in your package namespace to avoid a `R CMD check` note when referring to objects from the data mask. R does not have any way of knowing about the presence or absence of `.data` in a particular scope so you need to import it explicitly or equivalently declare it with `utils::globalVariables(".data")`.

Note that `rlang::.data` is a "fake" pronoun. Do not refer to `rlang::.data` with the `rlang::` qualifier in data masking code. Use the unqualified `.data` symbol that is automatically put in scope by data-masking functions.

dyn-dots *Dynamic dots features*

Description

The base `...` syntax supports:

- **Forwarding** arguments from function to function, matching them along the way to arguments.
- **Collecting** arguments inside data structures, e.g. with `c()` or `list()`.

Dynamic dots offer a few additional features, injection in particular:

1. You can **splice** arguments saved in a list with the splice operator `!!!`.
2. You can **inject** names with glue syntax on the left-hand side of `:=`.
3. Trailing commas are ignored, making it easier to copy and paste lines of arguments.

Add dynamic dots support in your functions

If your function takes dots, adding support for dynamic features is as easy as collecting the dots with `list2()` instead of `list()`. See also `dots_list()`, which offers more control over the collection.

In general, passing `...` to a function that supports dynamic dots causes your function to inherit the dynamic behaviour.

In packages, document dynamic dots with this standard tag:

```
@param ... <[`dynamic-dots`][rlang::dyn-dots]> What these dots do.
```

Examples

```
f <- function(...) {
  out <- list2(...)
  rev(out)
}

# Trailing commas are ignored
f(this = "that", )

# Splice lists of arguments with `!!!`
x <- list(alpha = "first", omega = "last")
f(!!!x)

# Inject a name using glue syntax
if (is_installed("glue")) {
  nm <- "key"
  f("{nm}" := "value")
  f("prefix_{nm}" := "value")
}
```

embrace-operator — *Embrace operator* {{

Description

The embrace operator {{ is used to create functions that call other data-masking functions. It transports a data-masked argument (an argument that can refer to columns of a data frame) from one function to another.

```
my_mean <- function(data, var) {
  dplyr::summarise(data, mean = mean({{ var }}))
}
```

Under the hood

{{ combines `enquo()` and `!!` in one step. The snippet above is equivalent to:

```
my_mean <- function(data, var) {
  var <- enquo(var)
  dplyr::summarise(data, mean = mean(!!var))
}
```

See Also

- What is data-masking and why do I need curly-curly?
- Data mask programming patterns

empty_env — *Get the empty environment*

Description

The empty environment is the only one that does not have a parent. It is always used as the tail of an environment chain such as the search path (see `search_envs()`).

Usage

```
empty_env()
```

Examples

```
# Create environments with nothing in scope:
child_env(empty_env())
```

englue — *Defuse function arguments with glue*

Description

`englue()` creates a string with the glue operators `{` and `{{`. These operators are normally used to inject names within dynamic dots. `englue()` makes them available anywhere within a function.

`englue()` must be used inside a function. `englue("{{ var }}")` defuses the argument `var` and transforms it to a string using the default name operation.

Usage

```
englue(x, env = caller_env(), error_call = current_env(), error_arg = "x")
```

Arguments

`x`	A string to interpolate with glue operators.
`env`	User environment where the interpolation data lives in case you're wrapping `englue()` in another function.
`error_call`	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.
`error_arg`	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.

Details

`englue("{{ var }}")` is equivalent to `as_label(enquo(var))`. It defuses `arg` and transforms the expression to a string with `as_label()`.

In dynamic dots, using only `{` is allowed. In `englue()` you must use `{{` at least once. Use `glue::glue()` for simple interpolation.

Before using `englue()` in a package, first ensure that glue is installed by adding it to your `Imports:` section.

```
usethis::use_package("glue", "Imports")
```

Wrapping `englue()`

You can provide englue semantics to a user provided string by supplying `env`. In this example we create a variant of `englue()` that supports a special `.qux` pronoun by:

- Creating an environment `masked_env` that inherits from the user env, the one where their data lives.

- Overriding the `error_arg` and `error_call` arguments to point to our own argument name and call environment. This pattern is slightly different from usual error context passing because `englue()` is a backend function that uses its own error context by default (and not a checking function that uses *your* error context by default).

```
my_englue <- function(text) {
  masked_env <- env(caller_env(), .qux = "QUX")

  englue(
    text,
    env = masked_env,
    error_arg = "text",
    error_call = current_env()
  )
}

# Users can then use your wrapper as they would use `englue()`:
fn <- function(x) {
  foo <- "FOO"
  my_englue("{{ x }}_{.qux}_{foo}")
}

fn(bar)
#> [1] "bar_QUX_FOO"
```

If you are creating a low level package on top of englue(), you should also consider exposing `env`, `error_arg` and `error_call` in your `englue()` wrapper so users can wrap your wrapper.

See Also

- Injecting with !!, !!!, and glue syntax

Examples

```
g <- function(var) englue("{{ var }}")
g(cyl)
g(1 + 1)
g(!!letters)

# These are equivalent to
as_label(quote(cyl))
as_label(quote(1 + 1))
as_label(letters)
```

enquo *Defuse function arguments*

Description

enquo() and enquos() defuse function arguments. A defused expression can be examined, modified, and injected into other expressions.

Defusing function arguments is useful for:

- Creating data-masking functions.
- Interfacing with another data-masking function using the defuse-and-inject pattern.

These are advanced tools. Make sure to first learn about the embrace operator {{ in Data mask programming patterns. {{ is easier to work with less theory, and it is sufficient in most applications.

Usage

```
enquo(arg)

enquos(
  ...,
  .named = FALSE,
  .ignore_empty = c("trailing", "none", "all"),
  .ignore_null = c("none", "all"),
  .unquote_names = TRUE,
  .homonyms = c("keep", "first", "last", "error"),
  .check_assign = FALSE
)
```

Arguments

arg	An unquoted argument name. The expression supplied to that argument is defused and returned.
...	Names of arguments to defuse.
.named	If TRUE, unnamed inputs are automatically named with as_label(). This is equivalent to applying exprs_auto_name() on the result. If FALSE, unnamed elements are left as is and, if fully unnamed, the list is given minimal names (a vector of ""). If NULL, fully unnamed results are left with NULL names.
.ignore_empty	Whether to ignore empty arguments. Can be one of "trailing", "none", "all". If "trailing", only the last argument is ignored if it is empty. Named arguments are not considered empty.
.ignore_null	Whether to ignore unnamed null arguments. Can be "none" or "all".
.unquote_names	Whether to treat := as =. Unlike =, the := syntax supports names injection.

`.homonyms` How to treat arguments with the same name. The default, `"keep"`, preserves these arguments. Set `.homonyms` to `"first"` to only keep the first occurrences, to `"last"` to keep the last occurrences, and to `"error"` to raise an informative error and indicate what arguments have duplicated names.

`.check_assign` Whether to check for `<-` calls. When `TRUE` a warning recommends users to use `=` if they meant to match a function parameter or wrap the `<-` call in curly braces otherwise. This ensures assignments are explicit.

Value

`enquo()` returns a quosure and `enquos()` returns a list of quosures.

Implicit injection

Arguments defused with `enquo()` and `enquos()` automatically gain injection support.

```
my_mean <- function(data, var) {
  var <- enquo(var)
  dplyr::summarise(data, mean(!!var))
}

# Can now use `!!` and `{{`
my_mean(mtcars, !!sym("cyl"))
```

See `enquo0()` and `enquos0()` for variants that don't enable injection.

See Also

- Defusing R expressions for an overview.
- `expr()` to defuse your own local expressions.
- Advanced defusal operators.
- `base::eval()` and `eval_bare()` for resuming evaluation of a defused expression.

Examples

```
# `enquo()` defuses the expression supplied by your user
f <- function(arg) {
  enquo(arg)
}

f(1 + 1)

# `enquos()` works with arguments and dots. It returns a list of
# expressions
f <- function(...) {
  enquos(...)
}

f(1 + 1, 2 * 10)
```

```
# `enquo()` and `enquos()` enable _injection_ and _embracing_ for
# your users
g <- function(arg) {
  f({{ arg }} * 2)
}
g(100)

column <- sym("cyl")
g(!!column)
```

---

env *Create a new environment*

Description

These functions create new environments.

- `env()` creates a child of the current environment by default and takes a variable number of named objects to populate it.
- `new_environment()` creates a child of the empty environment by default and takes a named list of objects to populate it.

Usage

```
env(...)

new_environment(data = list(), parent = empty_env())
```

Arguments

`..., data`	<dynamic> Named values. You can supply one unnamed to specify a custom parent, otherwise it defaults to the current environment.
`parent`	A parent environment.

Environments as objects

Environments are containers of uniquely named objects. Their most common use is to provide a scope for the evaluation of R expressions. Not all languages have first class environments, i.e. can manipulate scope as regular objects. Reification of scope is one of the most powerful features of R as it allows you to change what objects a function or expression sees when it is evaluated.

Environments also constitute a data structure in their own right. They are a collection of uniquely named objects, subsettable by name and modifiable by reference. This latter property (see section on reference semantics) is especially useful for creating mutable OO systems (cf the R6 package and the ggproto system for extending ggplot2).

Inheritance

All R environments (except the empty environment) are defined with a parent environment. An environment and its grandparents thus form a linear hierarchy that is the basis for lexical scoping in R. When R evaluates an expression, it looks up symbols in a given environment. If it cannot find these symbols there, it keeps looking them up in parent environments. This way, objects defined in child environments have precedence over objects defined in parent environments.

The ability of overriding specific definitions is used in the tidyeval framework to create powerful domain-specific grammars. A common use of masking is to put data frame columns in scope. See for example `as_data_mask()`.

Reference semantics

Unlike regular objects such as vectors, environments are an uncopyable object type. This means that if you have multiple references to a given environment (by assigning the environment to another symbol with `<-` or passing the environment as argument to a function), modifying the bindings of one of those references changes all other references as well.

See Also

`env_has()`, `env_bind()`.

Examples

```
# env() creates a new environment that inherits from the current
# environment by default
env <- env(a = 1, b = "foo")
env$b
identical(env_parent(env), current_env())

# Supply one unnamed argument to inherit from another environment:
env <- env(base_env(), a = 1, b = "foo")
identical(env_parent(env), base_env())


# Both env() and child_env() support tidy dots features:
objs <- list(b = "foo", c = "bar")
env <- env(a = 1, !!! objs)
env$c

# You can also unquote names with the definition operator `:=`
var <- "a"
env <- env(!!var := "A")
env$a


# Use new_environment() to create containers with the empty
# environment as parent:
env <- new_environment()
env_parent(env)

# Like other new_ constructors, it takes an object rather than dots:
```

```
new_environment(list(a = "foo", b = "bar"))
```

---

env_bind *Bind symbols to objects in an environment*

Description

These functions create bindings in an environment. The bindings are supplied through `...` as pairs of names and values or expressions. `env_bind()` is equivalent to evaluating a `<-` expression within the given environment. This function should take care of the majority of use cases but the other variants can be useful for specific problems.

- `env_bind()` takes named *values* which are bound in `.env`. `env_bind()` is equivalent to `base::assign()`.
- `env_bind_active()` takes named *functions* and creates active bindings in `.env`. This is equivalent to `base::makeActiveBinding()`. An active binding executes a function each time it is evaluated. The arguments are passed to `as_function()` so you can supply formulas instead of functions.

  Remember that functions are scoped in their own environment. These functions can thus refer to symbols from this enclosure that are not actually in scope in the dynamic environment where the active bindings are invoked. This allows creative solutions to difficult problems (see the implementations of `dplyr::do()` methods for an example).
- `env_bind_lazy()` takes named *expressions*. This is equivalent to `base::delayedAssign()`. The arguments are captured with `exprs()` (and thus support call-splicing and unquoting) and assigned to symbols in `.env`. These expressions are not evaluated immediately but lazily. Once a symbol is evaluated, the corresponding expression is evaluated in turn and its value is bound to the symbol (the expressions are thus evaluated only once, if at all).
- `%<~%` is a shortcut for `env_bind_lazy()`. It works like `<-` but the RHS is evaluated lazily.

Usage

```
env_bind(.env, ...)

env_bind_lazy(.env, ..., .eval_env = caller_env())

env_bind_active(.env, ...)

lhs %<~% rhs
```

Arguments

`.env`	An environment.
`...`	<dynamic> Named objects (`env_bind()`), expressions `env_bind_lazy()`, or functions (`env_bind_active()`). Use `zap()` to remove bindings.
`.eval_env`	The environment where the expressions will be evaluated when the symbols are forced.
`lhs`	The variable name to which `rhs` will be lazily assigned.
`rhs`	An expression lazily evaluated and assigned to `lhs`.

Value

The input object .env, with its associated environment modified in place, invisibly.

Side effects

Since environments have reference semantics (see relevant section in env() documentation), modifying the bindings of an environment produces effects in all other references to that environment. In other words, env_bind() and its variants have side effects.

Like other side-effecty functions like par() and options(), env_bind() and variants return the old values invisibly.

See Also

env_poke() for binding a single element.

Examples

```
# env_bind() is a programmatic way of assigning values to symbols
# with `<-`. We can add bindings in the current environment:
env_bind(current_env(), foo = "bar")
foo

# Or modify those bindings:
bar <- "bar"
env_bind(current_env(), bar = "BAR")
bar

# You can remove bindings by supplying zap sentinels:
env_bind(current_env(), foo = zap())
try(foo)

# Unquote-splice a named list of zaps
zaps <- rep_named(c("foo", "bar"), list(zap()))
env_bind(current_env(), !!!zaps)
try(bar)

# It is most useful to change other environments:
my_env <- env()
env_bind(my_env, foo = "foo")
my_env$foo

# A useful feature is to splice lists of named values:
vals <- list(a = 10, b = 20)
env_bind(my_env, !!!vals, c = 30)
my_env$b
my_env$c

# You can also unquote a variable referring to a symbol or a string
# as binding name:
var <- "baz"
env_bind(my_env, !!var := "BAZ")
```

```
my_env$baz


# The old values of the bindings are returned invisibly:
old <- env_bind(my_env, a = 1, b = 2, baz = "baz")
old

# You can restore the original environment state by supplying the
# old values back:
env_bind(my_env, !!!old)

# env_bind_lazy() assigns expressions lazily:
env <- env()
env_bind_lazy(env, name = { cat("forced!\n"); "value" })

# Referring to the binding will cause evaluation:
env$name

# But only once, subsequent references yield the final value:
env$name

# You can unquote expressions:
expr <- quote(message("forced!"))
env_bind_lazy(env, name = !!expr)
env$name


# By default the expressions are evaluated in the current
# environment. For instance we can create a local binding and refer
# to it, even though the variable is bound in a different
# environment:
who <- "mickey"
env_bind_lazy(env, name = paste(who, "mouse"))
env$name

# You can specify another evaluation environment with `.eval_env`:
eval_env <- env(who = "minnie")
env_bind_lazy(env, name = paste(who, "mouse"), .eval_env = eval_env)
env$name

# Or by unquoting a quosure:
quo <- local({
  who <- "fievel"
  quo(paste(who, "mouse"))
})
env_bind_lazy(env, name = !!quo)
env$name

# You can create active bindings with env_bind_active(). Active
# bindings execute a function each time they are evaluated:
fn <- function() {
  cat("I have been called\n")
  rnorm(1)
```

```
}

env <- env()
env_bind_active(env, symbol = fn)

# `fn` is executed each time `symbol` is evaluated or retrieved:
env$symbol
env$symbol
eval_bare(quote(symbol), env)
eval_bare(quote(symbol), env)

# All arguments are passed to as_function() so you can use the
# formula shortcut:
env_bind_active(env, foo = ~ runif(1))
env$foo
env$foo
```

---

env_browse *Browse environments*

Description

[Defunct]

- env_browse(env) is equivalent to evaluating browser() in env. It persistently sets the environment for step-debugging. Supply value = FALSE to disable browsing.
- env_is_browsed() is a predicate that inspects whether an environment is being browsed.

Usage

```
env_browse(env, value = TRUE)

env_is_browsed(env)
```

Arguments

env	An environment.
value	Whether to browse env.

Value

env_browse() returns the previous value of env_is_browsed() (a logical), invisibly.

env_cache *Cache a value in an environment*

Description

env_cache() is a wrapper around env_get() and env_poke() designed to retrieve a cached value from env.

- If the nm binding exists, it returns its value.
- Otherwise, it stores the default value in env and returns that.

Usage

```
env_cache(env, nm, default)
```

Arguments

env	An environment.
nm	Name of binding, a string.
default	The default value to store in env if nm does not exist yet.

Value

Either the value of nm or default if it did not exist yet.

Examples

```
e <- env(a = "foo")

# Returns existing binding
env_cache(e, "a", "default")

# Creates a `b` binding and returns its default value
env_cache(e, "b", "default")

# Now `b` is defined
e$b
```

env_clone *Clone or coalesce an environment*

Description

- env_clone() creates a new environment containing exactly the same bindings as the input, optionally with a new parent.
- env_coalesce() copies binding from the RHS environment into the LHS. If the RHS already contains bindings with the same name as in the LHS, those are kept as is.

Both these functions preserve active bindings and promises (the latter are only preserved on R >= 4.0.0).

Usage

```
env_clone(env, parent = env_parent(env))

env_coalesce(env, from)
```

Arguments

env	An environment.
parent	The parent of the cloned environment.
from	Environment to copy bindings from.

Examples

```
# A clone initially contains the same bindings as the original
# environment
env <- env(a = 1, b = 2)
clone <- env_clone(env)

env_print(clone)
env_print(env)

# But it can acquire new bindings or change existing ones without
# impacting the original environment
env_bind(clone, a = "foo", c = 3)

env_print(clone)
env_print(env)


# `env_coalesce()` copies bindings from one environment to another
lhs <- env(a = 1)
rhs <- env(a = "a", b = "b", c = "c")
env_coalesce(lhs, rhs)
env_print(lhs)
```

```
# To copy all the bindings from `rhs` into `lhs`, first delete the
# conflicting bindings from `rhs`
env_unbind(lhs, env_names(rhs))
env_coalesce(lhs, rhs)
env_print(lhs)
```

---

env_depth *Depth of an environment chain*

Description

This function returns the number of environments between env and the empty environment, including env. The depth of env is also the number of parents of env (since the empty environment counts as a parent).

Usage

```
env_depth(env)
```

Arguments

env	An environment.

Value

An integer.

See Also

The section on inheritance in env() documentation.

Examples

```
env_depth(empty_env())
env_depth(pkg_env("rlang"))
```

env_get *Get an object in an environment*

Description

env_get() extracts an object from an enviroment env. By default, it does not look in the parent environments. env_get_list() extracts multiple objects from an environment into a named list.

Usage

```
env_get(env = caller_env(), nm, default, inherit = FALSE, last = empty_env())

env_get_list(
  env = caller_env(),
  nms,
  default,
  inherit = FALSE,
  last = empty_env()
)
```

Arguments

env	An environment.
nm	Name of binding, a string.
default	A default value in case there is no binding for nm in env.
inherit	Whether to look for bindings in the parent environments.
last	Last environment inspected when inherit is TRUE. Can be useful in conjunction with base::topenv().
nms	Names of bindings, a character vector.

Value

An object if it exists. Otherwise, throws an error.

See Also

env_cache() for a variant of env_get() designed to cache a value in an environment.

Examples

```
parent <- child_env(NULL, foo = "foo")
env <- child_env(parent, bar = "bar")

# This throws an error because `foo` is not directly defined in env:
# env_get(env, "foo")

# However `foo` can be fetched in the parent environment:
```

```
env_get(env, "foo", inherit = TRUE)

# You can also avoid an error by supplying a default value:
env_get(env, "foo", default = "FOO")
```

---

env_has *Does an environment have or see bindings?*

Description

`env_has()` is a vectorised predicate that queries whether an environment owns bindings personally (with `inherit` set to `FALSE`, the default), or sees them in its own environment or in any of its parents (with `inherit = TRUE`).

Usage

```
env_has(env = caller_env(), nms, inherit = FALSE)
```

Arguments

env	An environment.
nms	A character vector of binding names for which to check existence.
inherit	Whether to look for bindings in the parent environments.

Value

A named logical vector as long as `nms`.

Examples

```
parent <- child_env(NULL, foo = "foo")
env <- child_env(parent, bar = "bar")

# env does not own `foo` but sees it in its parent environment:
env_has(env, "foo")
env_has(env, "foo", inherit = TRUE)
```

env_inherits — *Does environment inherit from another environment?*

Description

This returns `TRUE` if `x` has `ancestor` among its parents.

Usage

```
env_inherits(env, ancestor)
```

Arguments

`env`	An environment.
`ancestor`	Another environment from which `x` might inherit.

env_is_user_facing — *Is frame environment user facing?*

Description

Detects if `env` is user-facing, that is, whether it's an environment that inherits from:

- The global environment, as would happen when called interactively
- A package that is currently being tested

If either is true, we consider `env` to belong to an evaluation frame that was called *directly* by the end user. This is by contrast to *indirect* calls by third party functions which are not user facing.

For instance the lifecycle package uses `env_is_user_facing()` to figure out whether a deprecated function was called directly or indirectly, and select an appropriate verbosity level as a function of that.

Usage

```
env_is_user_facing(env)
```

Arguments

`env`	An environment.

Escape hatch

You can override the return value of `env_is_user_facing()` by setting the global option `"rlang_user_facing"` to:

- `TRUE` or `FALSE`.
- A package name as a string. Then `env_is_user_facing(x)` returns `TRUE` if `x` inherits from the namespace corresponding to that package name.

Examples

```
fn <- function() {
  env_is_user_facing(caller_env())
}

# Direct call of `fn()` from the global env
with(global_env(), fn())

# Indirect call of `fn()` from a package
with(ns_env("utils"), fn())
```

env_name — *Label of an environment*

Description

Special environments like the global environment have their own names. `env_name()` returns:

- "global" for the global environment.
- "empty" for the empty environment.
- "base" for the base package environment (the last environment on the search path).
- "namespace:pkg" if `env` is the namespace of the package "pkg".
- The `name` attribute of `env` if it exists. This is how the package environments and the imports environments store their names. The name of package environments is typically "package:pkg".
- The empty string `""` otherwise.

`env_label()` is exactly like `env_name()` but returns the memory address of anonymous environments as fallback.

Usage

```
env_name(env)

env_label(env)
```

Arguments

`env`	An environment.

Examples

```
# Some environments have specific names:
env_name(global_env())
env_name(ns_env("rlang"))

# Anonymous environments don't have names but are labelled by their
# address in memory:
env_name(env())
env_label(env())
```

env_names *Names and numbers of symbols bound in an environment*

Description

env_names() returns object names from an enviroment env as a character vector. All names are returned, even those starting with a dot. env_length() returns the number of bindings.

Usage

```
env_names(env)

env_length(env)
```

Arguments

env	An environment.

Value

A character vector of object names.

Names of symbols and objects

Technically, objects are bound to symbols rather than strings, since the R interpreter evaluates symbols (see is_expression() for a discussion of symbolic objects versus literal objects). However it is often more convenient to work with strings. In rlang terminology, the string corresponding to a symbol is called the *name* of the symbol (or by extension the name of an object bound to a symbol).

Encoding

There are deep encoding issues when you convert a string to symbol and vice versa. Symbols are *always* in the native encoding. If that encoding (let's say latin1) cannot support some characters, these characters are serialised to ASCII. That's why you sometimes see strings looking like <U+1234>, especially if you're running Windows (as R doesn't support UTF-8 as native encoding on that platform).

To alleviate some of the encoding pain, env_names() always returns a UTF-8 character vector (which is fine even on Windows) with ASCII unicode points translated back to UTF-8.

Examples

```
env <- env(a = 1, b = 2)
env_names(env)
```

env_parent *Get parent environments*

Description

- env_parent() returns the parent environment of env if called with n = 1, the grandparent with n = 2, etc.
- env_tail() searches through the parents and returns the one which has empty_env() as parent.
- env_parents() returns the list of all parents, including the empty environment. This list is named using env_name().

See the section on *inheritance* in env()'s documentation.

Usage

```
env_parent(env = caller_env(), n = 1)

env_tail(env = caller_env(), last = global_env())

env_parents(env = caller_env(), last = global_env())
```

Arguments

env	An environment.
n	The number of generations to go up.
last	The environment at which to stop. Defaults to the global environment. The empty environment is always a stopping condition so it is safe to leave the default even when taking the tail or the parents of an environment on the search path. env_tail() returns the environment which has last as parent and env_parents() returns the list of environments up to last.

Value

An environment for env_parent() and env_tail(), a list of environments for env_parents().

Examples

```
# Get the parent environment with env_parent():
env_parent(global_env())

# Or the tail environment with env_tail():
env_tail(global_env())

# By default, env_parent() returns the parent environment of the
# current evaluation frame. If called at top-level (the global
```

```
# frame), the following two expressions are equivalent:
env_parent()
env_parent(base_env())

# This default is more handy when called within a function. In this
# case, the enclosure environment of the function is returned
# (since it is the parent of the evaluation frame):
enclos_env <- env()
fn <- set_env(function() env_parent(), enclos_env)
identical(enclos_env, fn())
```

---

env_poke *Poke an object in an environment*

Description

env_poke() will assign or reassign a binding in env if create is TRUE. If create is FALSE and a binding does not already exists, an error is issued.

Usage

```
env_poke(env = caller_env(), nm, value, inherit = FALSE, create = !inherit)
```

Arguments

env	An environment.
nm	Name of binding, a string.
value	The value for a new binding.
inherit	Whether to look for bindings in the parent environments.
create	Whether to create a binding if it does not already exist in the environment.

Details

If inherit is TRUE, the parents environments are checked for an existing binding to reassign. If not found and create is TRUE, a new binding is created in env. The default value for create is a function of inherit: FALSE when inheriting, TRUE otherwise.

This default makes sense because the inheriting case is mostly for overriding an existing binding. If not found, something probably went wrong and it is safer to issue an error. Note that this is different to the base R operator <<- which will create a binding in the global environment instead of the current environment when no existing binding is found in the parents.

Value

The old value of nm or a zap sentinel if the binding did not exist yet.

See Also

env_bind() for binding multiple elements. env_cache() for a variant of env_poke() designed to cache values.

`env_print` *Pretty-print an environment*

Description

This prints:

- The label and the parent label.
- Whether the environment is locked.
- The bindings in the environment (up to 20 bindings). They are printed succintly using `pillar::type_sum()` (if available, otherwise uses an internal version of that generic). In addition fancy bindings (actives and promises) are indicated as such.
- Locked bindings get a `[L]` tag

Note that printing a package namespace (see `ns_env()`) with `env_print()` will typically tag function bindings as `<lazy>` until they are evaluated the first time. This is because package functions are lazily-loaded from disk to improve performance when loading a package.

Usage

```
env_print(env = caller_env())
```

Arguments

`env`	An environment, or object that can be converted to an environment by `get_env()`.

`env_unbind` *Remove bindings from an environment*

Description

`env_unbind()` is the complement of `env_bind()`. Like `env_has()`, it ignores the parent environments of `env` by default. Set `inherit` to `TRUE` to track down bindings in parent environments.

Usage

```
env_unbind(env = caller_env(), nms, inherit = FALSE)
```

Arguments

`env`	An environment.
`nms`	A character vector of binding names to remove.
`inherit`	Whether to look for bindings in the parent environments.

Value

The input object env with its associated environment modified in place, invisibly.

Examples

```
env <- env(foo = 1, bar = 2)
env_has(env, c("foo", "bar"))

# Remove bindings with `env_unbind()`
env_unbind(env, c("foo", "bar"))
env_has(env, c("foo", "bar"))

# With inherit = TRUE, it removes bindings in parent environments
# as well:
parent <- env(empty_env(), foo = 1, bar = 2)
env <- env(parent, foo = "b")

env_unbind(env, "foo", inherit = TRUE)
env_has(env, c("foo", "bar"))
env_has(env, c("foo", "bar"), inherit = TRUE)
```

---

eval_bare — *Evaluate an expression in an environment*

Description

eval_bare() is a lower-level version of function base::eval(). Technically, it is a simple wrapper around the C function Rf_eval(). You generally don't need to use eval_bare() instead of eval(). Its main advantage is that it handles stack-sensitive calls (such as return(), on.exit() or parent.frame()) more consistently when you pass an enviroment of a frame on the call stack.

Usage

```
eval_bare(expr, env = parent.frame())
```

Arguments

expr	An expression to evaluate.
env	The environment in which to evaluate the expression.

Details

These semantics are possible because eval_bare() creates only one frame on the call stack whereas eval() creates two frames, the second of which has the user-supplied environment as frame environment. When you supply an existing frame environment to base::eval() there will be two frames on the stack with the same frame environment. Stack-sensitive functions only detect the topmost of these frames. We call these evaluation semantics "stack inconsistent".

Evaluating expressions in the actual frame environment has useful practical implications for eval_bare():

- `return()` calls are evaluated in frame environments that might be burried deep in the call stack. This causes a long return that unwinds multiple frames (triggering the `on.exit()` event for each frame). By contrast `eval()` only returns from the `eval()` call, one level up.
- `on.exit()`, `parent.frame()`, `sys.call()`, and generally all the stack inspection functions `sys.xxx()` are evaluated in the correct frame environment. This is similar to how this type of calls can be evaluated deep in the call stack because of lazy evaluation, when you force an argument that has been passed around several times.

The flip side of the semantics of `eval_bare()` is that it can't evaluate `break` or `next` expressions even if called within a loop.

See Also

`eval_tidy()` for evaluation with data mask and quosure support.

Examples

```
# eval_bare() works just like base::eval() but you have to create
# the evaluation environment yourself:
eval_bare(quote(foo), env(foo = "bar"))

# eval() has different evaluation semantics than eval_bare(). It
# can return from the supplied environment even if its an
# environment that is not on the call stack (i.e. because you've
# created it yourself). The following would trigger an error with
# eval_bare():
ret <- quote(return("foo"))
eval(ret, env())
# eval_bare(ret, env())  # "no function to return from" error

# Another feature of eval() is that you can control surround loops:
bail <- quote(break)
while (TRUE) {
  eval(bail)
  # eval_bare(bail)  # "no loop for break/next" error
}

# To explore the consequences of stack inconsistent semantics, let's
# create a function that evaluates `parent.frame()` deep in the call
# stack, in an environment corresponding to a frame in the middle of
# the stack. For consistency with R's lazy evaluation semantics, we'd
# expect to get the caller of that frame as result:
fn <- function(eval_fn) {
  list(
    returned_env = middle(eval_fn),
    actual_env = current_env()
  )
}
middle <- function(eval_fn) {
  deep(eval_fn, current_env())
}
deep <- function(eval_fn, eval_env) {
```

```
  expr <- quote(parent.frame())
  eval_fn(expr, eval_env)
}

# With eval_bare(), we do get the expected environment:
fn(rlang::eval_bare)

# But that's not the case with base::eval():
fn(base::eval)
```

---

eval_tidy *Evaluate an expression with quosures and pronoun support*

Description

eval_tidy() is a variant of base::eval() that powers the tidy evaluation framework. Like eval() it accepts user data as argument. Whereas eval() simply transforms the data to an environment, eval_tidy() transforms it to a data mask with as_data_mask(). Evaluating in a data mask enables the following features:

- Quosures. Quosures are expressions bundled with an environment. If data is supplied, objects in the data mask always have precedence over the quosure environment, i.e. the data masks the environment.
- Pronouns. If data is supplied, the .env and .data pronouns are installed in the data mask. .env is a reference to the calling environment and .data refers to the data argument. These pronouns are an escape hatch for the data mask ambiguity problem.

Usage

```
eval_tidy(expr, data = NULL, env = caller_env())
```

Arguments

expr	An expression or quosure to evaluate.
data	A data frame, or named list or vector. Alternatively, a data mask created with as_data_mask() or new_data_mask(). Objects in data have priority over those in env. See the section about data masking.
env	The environment in which to evaluate expr. This environment is not applicable for quosures because they have their own environments.

When should eval_tidy() be used instead of eval()?

base::eval() is sufficient for simple evaluation. Use eval_tidy() when you'd like to support expressions referring to the .data pronoun, or when you need to support quosures.

If you're evaluating an expression captured with injection support, it is recommended to use eval_tidy() because users may inject quosures.

Note that unwrapping a quosure with quo_get_expr() does not guarantee that there is no quosures inside the expression. Quosures might be unquoted anywhere in the expression tree. For instance, the following does not work reliably in the presence of nested quosures:

```
my_quoting_fn <- function(x) {
  x <- enquo(x)
  expr <- quo_get_expr(x)
  env <- quo_get_env(x)
  eval(expr, env)
}

# Works:
my_quoting_fn(toupper(letters))

# Fails because of a nested quosure:
my_quoting_fn(toupper(!!quo(letters)))
```

Stack semantics of `eval_tidy()`

`eval_tidy()` always evaluates in a data mask, even when `data` is `NULL`. Because of this, it has different stack semantics than `base::eval()`:

- Lexical side effects, such as assignment with `<-`, occur in the mask rather than `env`.
- Functions that require the evaluation environment to correspond to a frame on the call stack do not work. This is why `return()` called from a quosure does not work.
- The mask environment creates a new branch in the tree representation of backtraces (which you can visualise in a `browser()` session with `lobstr::cst()`).

See also `eval_bare()` for more information about these differences.

See Also

- What is data-masking and why do I need curly-curly?.
- What are quosures and when are they needed?.
- Defusing R expressions.
- `new_data_mask()` and `as_data_mask()` for manually creating data masks.

Examples

```
# With simple defused expressions eval_tidy() works the same way as
# eval():
fruit <- "apple"
vegetable <- "potato"
expr <- quote(paste(fruit, vegetable, sep = " or "))
expr

eval(expr)
eval_tidy(expr)
```

```
# Both accept a data mask as argument:
data <- list(fruit = "banana", vegetable = "carrot")
eval(expr, data)
eval_tidy(expr, data)

# The main difference is that eval_tidy() supports quosures:
with_data <- function(data, expr) {
  quo <- enquo(expr)
  eval_tidy(quo, data)
}
with_data(NULL, fruit)
with_data(data, fruit)

# eval_tidy() installs the `.data` and `.env` pronouns to allow
# users to be explicit about variable references:
with_data(data, .data$fruit)
with_data(data, .env$fruit)
```

---

exec *Execute a function*

Description

This function constructs and evaluates a call to `.fn`. It has two primary uses:

- To call a function with arguments stored in a list (if the function doesn't support dynamic dots). Splice the list of arguments with `!!!`.
- To call every function stored in a list (in conjunction with `map()`/ `lapply()`)

Usage

```
exec(.fn, ..., .env = caller_env())
```

Arguments

`.fn`	A function, or function name as a string.
`...`	<dynamic> Arguments for `.fn`.
`.env`	Environment in which to evaluate the call. This will be most useful if `.fn` is a string, or the function has side-effects.

Examples

```
args <- list(x = c(1:10, 100, NA), na.rm = TRUE)
exec("mean", !!!args)
exec("mean", !!!args, trim = 0.2)

fs <- list(a = function() "a", b = function() "b")
lapply(fs, exec)
```

```
# Compare to do.call it will not automatically inline expressions
# into the evaluated call.
x <- 10
args <- exprs(x1 = x + 1, x2 = x * 2)
exec(list, !!!args)
do.call(list, args)

# exec() is not designed to generate pretty function calls. This is
# most easily seen if you call a function that captures the call:
f <- disp ~ cyl
exec("lm", f, data = mtcars)

# If you need finer control over the generated call, you'll need to
# construct it yourself. This may require creating a new environment
# with carefully constructed bindings
data_env <- env(data = mtcars)
eval(expr(lm(!!f, data)), data_env)
```

---

expr *Defuse an R expression*

Description

expr() defuses an R expression with injection support.

It is equivalent to base::bquote().

Arguments

expr	An expression to defuse.

See Also

- Defusing R expressions for an overview.
- enquo() to defuse non-local expressions from function arguments.
- Advanced defusal operators.
- sym() and call2() for building expressions (symbols and calls respectively) programmatically.
- base::eval() and eval_bare() for resuming evaluation of a defused expression.

Examples

```
# R normally returns the result of an expression
1 + 1

# `expr()` defuses the expression that you have supplied and
# returns it instead of its value
expr(1 + 1)
```

```
expr(toupper(letters))

# It supports _injection_ with `!!` and `!!!`. This is a convenient
# way of modifying part of an expression by injecting other
# objects.
var <- "cyl"
expr(with(mtcars, mean(!!sym(var))))

vars <- c("cyl", "am")
expr(with(mtcars, c(!!!syms(vars))))

# Compare to the normal way of building expressions
call("with", call("mean", sym(var)))

call("with", call2("c", !!!syms(vars)))
```

---

exprs_auto_name *Ensure that all elements of a list of expressions are named*

Description

This gives default names to unnamed elements of a list of expressions (or expression wrappers such as formulas or quosures), deparsed with `as_label()`.

Usage

```
exprs_auto_name(
  exprs,
  ...,
  repair_auto = c("minimal", "unique"),
  repair_quiet = FALSE
)

quos_auto_name(quos)
```

Arguments

`exprs`	A list of expressions.
`...`	These dots are for future extensions and must be empty.
`repair_auto`	Whether to repair the automatic names. By default, minimal names are returned. See `?vctrs::vec_as_names` for information about name repairing.
`repair_quiet`	Whether to inform user about repaired names.
`quos`	A list of quosures.

expr_print *Print an expression*

Description

`expr_print()`, powered by `expr_deparse()`, is an alternative printer for R expressions with a few improvements over the base R printer.

- It colourises quosures according to their environment. Quosures from the global environment are printed normally while quosures from local environments are printed in unique colour (or in italic when all colours are taken).
- It wraps inlined objects in angular brackets. For instance, an integer vector unquoted in a function call (e.g. `expr(foo(!!(1:3)))`) is printed like this: `foo(<int: 1L, 2L, 3L>)` while by default R prints the code to create that vector: `foo(1:3)` which is ambiguous.
- It respects the width boundary (from the global option `width`) in more cases.

Usage

```
expr_print(x, ...)

expr_deparse(x, ..., width = peek_option("width"))
```

Arguments

`x`	An object or expression to print.
`...`	Arguments passed to `expr_deparse()`.
`width`	The width of the deparsed or printed expression. Defaults to the global option `width`.

Value

`expr_deparse()` returns a character vector of lines. `expr_print()` returns its input invisibly.

Examples

```
# It supports any object. Non-symbolic objects are always printed
# within angular brackets:
expr_print(1:3)
expr_print(function() NULL)

# Contrast this to how the code to create these objects is printed:
expr_print(quote(1:3))
expr_print(quote(function() NULL))

# The main cause of non-symbolic objects in expressions is
# quasiquotation:
expr_print(expr(foo(!!(1:3))))
```

```
# Quosures from the global environment are printed normally:
expr_print(quo(foo))
expr_print(quo(foo(!!quo(bar))))

# Quosures from local environments are colourised according to
# their environments (if you have crayon installed):
local_quo <- local(quo(foo))
expr_print(local_quo)

wrapper_quo <- local(quo(bar(!!local_quo, baz)))
expr_print(wrapper_quo)
```

`faq-options` *Global options for rlang*

Description

rlang has several options which may be set globally to control behavior. A brief description of each is given here. If any functions are referenced, refer to their documentation for additional details.

- `rlang_interactive`: A logical value used by `is_interactive()`. This can be set to `TRUE` to test interactive behavior in unit tests, for example.
- `rlang_backtrace_on_error`: A character string which controls whether backtraces are displayed with error messages, and the level of detail they print. See rlang_backtrace_on_error for the possible option values.
- `rlang_trace_format_srcrefs`: A logical value used to control whether srcrefs are printed as part of the backtrace.
- `rlang_trace_top_env`: An environment which will be treated as the top-level environment when printing traces. See `trace_back()` for examples.

`fn_body` *Get or set function body*

Description

`fn_body()` is a simple wrapper around `base::body()`. It always returns a \{ expression and throws an error when the input is a primitive function (whereas `body()` returns `NULL`). The setter version preserves attributes, unlike body<-.

Usage

```
fn_body(fn = caller_fn())

fn_body(fn) <- value
```

Arguments

fn	A function. It is looked up in the calling frame if not supplied.
value	New formals or formals names for fn.

Examples

```
# fn_body() is like body() but always returns a block:
fn <- function() do()
body(fn)
fn_body(fn)

# It also throws an error when used on a primitive function:
try(fn_body(base::list))
```

---

fn_env *Return the closure environment of a function*

Description

Closure environments define the scope of functions (see env()). When a function call is evaluated, R creates an evaluation frame that inherits from the closure environment. This makes all objects defined in the closure environment and all its parents available to code executed within the function.

Usage

```
fn_env(fn)

fn_env(x) <- value
```

Arguments

fn, x	A function.
value	A new closure environment for the function.

Details

fn_env() returns the closure environment of fn. There is also an assignment method to set a new closure environment.

Examples

```
env <- child_env("base")
fn <- with_env(env, function() NULL)
identical(fn_env(fn), env)

other_env <- child_env("base")
fn_env(fn) <- other_env
identical(fn_env(fn), other_env)
```

fn_fmls *Extract arguments from a function*

Description

`fn_fmls()` returns a named list of formal arguments. `fn_fmls_names()` returns the names of the arguments. `fn_fmls_syms()` returns formals as a named list of symbols. This is especially useful for forwarding arguments in constructed calls.

Usage

```
fn_fmls(fn = caller_fn())

fn_fmls_names(fn = caller_fn())

fn_fmls_syms(fn = caller_fn())

fn_fmls(fn) <- value

fn_fmls_names(fn) <- value
```

Arguments

fn	A function. It is looked up in the calling frame if not supplied.
value	New formals or formals names for `fn`.

Details

Unlike `formals()`, these helpers throw an error with primitive functions instead of returning `NULL`.

See Also

`call_args()` and `call_args_names()`

Examples

```
# Extract from current call:
fn <- function(a = 1, b = 2) fn_fmls()
fn()

# fn_fmls_syms() makes it easy to forward arguments:
call2("apply", !!! fn_fmls_syms(lapply))

# You can also change the formals:
fn_fmls(fn) <- list(A = 10, B = 20)
fn()

fn_fmls_names(fn) <- c("foo", "bar")
fn()
```

format_error_bullets *Format bullets for error messages*

Description

`format_error_bullets()` takes a character vector and returns a single string (or an empty vector if the input is empty). The elements of the input vector are assembled as a list of bullets, depending on their names:

- Unnamed elements are unindented. They act as titles or subtitles.
- Elements named `"*"` are bulleted with a cyan "bullet" symbol.
- Elements named `"i"` are bulleted with a blue "info" symbol.
- Elements named `"x"` are bulleted with a red "cross" symbol.
- Elements named `"v"` are bulleted with a green "tick" symbol.
- Elements named `"!"` are bulleted with a yellow "warning" symbol.
- Elements named `">"` are bulleted with an "arrow" symbol.
- Elements named `" "` start with an indented line break.

For convenience, if the vector is fully unnamed, the elements are formatted as "*" bullets.

The bullet formatting for errors follows the idea that sentences in error messages are best kept short and simple. The best way to present the information is in the `cnd_body()` method of an error conditon as a bullet list of simple sentences containing a single clause. The info and cross symbols of the bullets provide hints on how to interpret the bullet relative to the general error issue, which should be supplied as `cnd_header()`.

Usage

```
format_error_bullets(x)
```

Arguments

`x`	A named character vector of messages. Named elements are prefixed with the corresponding bullet. Elements named with a single space `" "` trigger a line break from the previous bullet.

Examples

```
# All bullets
writeLines(format_error_bullets(c("foo", "bar")))

# This is equivalent to
writeLines(format_error_bullets(set_names(c("foo", "bar"), "*")))

# Supply named elements to format info, cross, and tick bullets
writeLines(format_error_bullets(c(i = "foo", x = "bar", v = "baz", "*" = "quux")))
```

```
# An unnamed element breaks the line
writeLines(format_error_bullets(c(i = "foo\nbar")))

# A " " element breaks the line within a bullet (with indentation)
writeLines(format_error_bullets(c(i = "foo", " " = "bar")))
```

---

f_rhs *Get or set formula components*

Description

f_rhs extracts the righthand side, f_lhs extracts the lefthand side, and f_env extracts the environment. All functions throw an error if f is not a formula.

Usage

```
f_rhs(f)

f_rhs(x) <- value

f_lhs(f)

f_lhs(x) <- value

f_env(f)

f_env(x) <- value
```

Arguments

f, x	A formula
value	The value to replace with.

Value

f_rhs and f_lhs return language objects (i.e. atomic vectors of length 1, a name, or a call). f_env returns an environment.

Examples

```
f_rhs(~ 1 + 2 + 3)
f_rhs(~ x)
f_rhs(~ "A")
f_rhs(1 ~ 2)

f_lhs(~ y)
f_lhs(x ~ y)

f_env(~ x)
```

f_text *Turn RHS of formula into a string or label*

Description

Equivalent of expr_text() and expr_label() for formulas.

Usage

```
f_text(x, width = 60L, nlines = Inf)

f_name(x)

f_label(x)
```

Arguments

x	A formula.
width	Width of each line.
nlines	Maximum number of lines to extract.

Examples

```
f <- ~ a + b + bc
f_text(f)
f_label(f)

# Names a quoted with ``
f_label(~ x)
# Strings are encoded
f_label(~ "a\nb")
# Long expressions are collapsed
f_label(~ foo({
  1 + 2
  print(x)
}))
```

get_env *Get or set the environment of an object*

Description

These functions dispatch internally with methods for functions, formulas and frames. If called with a missing argument, the environment of the current evaluation frame is returned. If you call get_env() with an environment, it acts as the identity function and the environment is simply returned (this helps simplifying code when writing generic functions for environments).

Usage

```
get_env(env, default = NULL)

set_env(env, new_env = caller_env())

env_poke_parent(env, new_env)
```

Arguments

env	An environment.
default	The default environment in case env does not wrap an environment. If NULL and no environment could be extracted, an error is issued.
new_env	An environment to replace env with.

Details

While set_env() returns a modified copy and does not have side effects, env_poke_parent() operates changes the environment by side effect. This is because environments are uncopyable. Be careful not to change environments that you don't own, e.g. a parent environment of a function from a package.

See Also

quo_get_env() and quo_set_env() for versions of get_env() and set_env() that only work on quosures.

Examples

```
# Environment of closure functions:
fn <- function() "foo"
get_env(fn)

# Or of quosures or formulas:
get_env(~foo)
get_env(quo(foo))


# Provide a default in case the object doesn't bundle an environment.
# Let's create an unevaluated formula:
f <- quote(~foo)

# The following line would fail if run because unevaluated formulas
# don't bundle an environment (they didn't have the chance to
# record one yet):
# get_env(f)

# It is often useful to provide a default when you're writing
# functions accepting formulas as input:
default <- env()
identical(get_env(f, default), default)
```

```
# set_env() can be used to set the enclosure of functions and
# formulas. Let's create a function with a particular environment:
env <- child_env("base")
fn <- set_env(function() NULL, env)

# That function now has `env` as enclosure:
identical(get_env(fn), env)
identical(get_env(fn), current_env())

# set_env() does not work by side effect. Setting a new environment
# for fn has no effect on the original function:
other_env <- child_env(NULL)
set_env(fn, other_env)
identical(get_env(fn), other_env)

# Since set_env() returns a new function with a different
# environment, you'll need to reassign the result:
fn <- set_env(fn, other_env)
identical(get_env(fn), other_env)
```

---

global_entrace *Entrace unexpected errors*

Description

`global_entrace()` enriches base errors, warnings, and messages with rlang features.

- They are assigned a backtrace. You can configure whether to display a backtrace on error with the rlang_backtrace_on_error global option.
- They are recorded in `last_error()`, `last_warnings()`, or `last_messages()`. You can inspect backtraces at any time by calling these functions.

Set global entracing in your RProfile with:

```
rlang::global_entrace()
```

Usage

```
global_entrace(enable = TRUE, class = c("error", "warning", "message"))
```

Arguments

`enable`	Whether to enable or disable global handling.
`class`	A character vector of one or several classes of conditions to be entraced.

Inside RMarkdown documents

Call global_entrace() inside an RMarkdown document to cause errors and warnings to be promoted to rlang conditions that include a backtrace. This needs to be done in a separate setup chunk before the first error or warning.

This is useful in conjunction with rlang_backtrace_on_error_report and rlang_backtrace_on_warning_report. To get full entracing in an Rmd document, include this in a setup chunk before the first error or warning is signalled.

````
```{r setup}
rlang::global_entrace()
options(rlang_backtrace_on_warning_report = "full")
options(rlang_backtrace_on_error_report = "full")
```
````

Under the hood

On R 4.0 and newer, global_entrace() installs a global handler with globalCallingHandlers(). On older R versions, entrace() is set as an option(error =) handler. The latter method has the disadvantage that only one handler can be set at a time. This means that you need to manually switch between entrace() and other handlers like recover(). Also this causes a conflict with IDE handlers (e.g. in RStudio).

---

global_handle — *Register default global handlers*

Description

global_handle() sets up a default configuration for error, warning, and message handling. It calls:

- global_entrace() to enable rlang errors and warnings globally.
- global_prompt_install() to recover from packageNotFoundErrors with a user prompt to install the missing package. Note that at the time of writing (R 4.1), there are only very limited situations where this handler works.

Usage

```
global_handle(entrace = TRUE, prompt_install = TRUE)
```

Arguments

entrace	Passed as enable argument to global_entrace().
prompt_install	Passed as enable argument to global_prompt_install().

global_prompt_install *Prompt user to install missing packages*

Description

When enabled, packageNotFoundError thrown by loadNamespace() cause a user prompt to install the missing package and continue without interrupting the current program.

This is similar to how check_installed() prompts users to install required packages. It uses the same install strategy, using pak if available and install.packages() otherwise.

Usage

```
global_prompt_install(enable = TRUE)
```

Arguments

enable	Whether to enable or disable global handling.

glue-operators *Name injection with "{" and "{{"*

Description

Dynamic dots (and data-masked dots which are dynamic by default) have built-in support for names interpolation with the glue package.

```
tibble::tibble(foo = 1)
#> # A tibble: 1 x 1
#>     foo
#>   <dbl>
#> 1     1

foo <- "name"
tibble::tibble("{foo}" := 1)
#> # A tibble: 1 x 1
#>    name
#>   <dbl>
#> 1     1
```

Inside functions, embracing an argument with {{ inserts the expression supplied as argument in the string. This gives an indication on the variable or computation supplied as argument:

```
tib <- function(x) {
  tibble::tibble("var: {{ x }}" := x)
}

tib(1 + 1)
#> # A tibble: 1 x 1
#>   `var: 1 + 1`
#>          <dbl>
#> 1            2
```

See also `englue()` to string-embrace outside of dynamic dots.

```
g <- function(x) {
  englue("var: {{ x }}")
}

g(1 + 1)
#> [1] "var: 1 + 1"
```

Technically, `"{{"` defuses a function argument, calls `as_label()` on the expression supplied as argument, and inserts the result in the string.

`"{"` and `"{{"`:

While `glue::glue()` only supports `"{"`, dynamic dots support both `"{"` and `"{{"`. The double brace variant is similar to the embrace operator `{{` available in data-masked arguments.

In the following example, the embrace operator is used in a glue string to name the result with a default name that represents the expression supplied as argument:

```
my_mean <- function(data, var) {
  data %>% dplyr::summarise("{{ var }}" := mean({{ var }}))
}

mtcars %>% my_mean(cyl)
#> # A tibble: 1 x 1
#>     cyl
#>   <dbl>
#> 1  6.19

mtcars %>% my_mean(cyl * am)
#> # A tibble: 1 x 1
#>   `cyl * am`
#>        <dbl>
#> 1       2.06
```

`"{{"` is only meant for inserting an expression supplied as argument to a function. The result of the expression is not inspected or used. To interpolate a string stored in a variable, use the regular glue operator `"{"` instead:

```
my_mean <- function(data, var, name = "mean") {
  data %>% dplyr::summarise("{name}" := mean({{ var }}))
```

```
}

mtcars %>% my_mean(cyl)
#> # A tibble: 1 x 1
#>    mean
#>   <dbl>
#> 1  6.19

mtcars %>% my_mean(cyl, name = "cyl")
#> # A tibble: 1 x 1
#>     cyl
#>   <dbl>
#> 1  6.19
```

Using the wrong operator causes unexpected results:

```
x <- "name"

list2("{{ x }}" := 1)
#> $`"name"`
#> [1] 1

list2("{x}" := 1)
#> $name
#> [1] 1
```

Ideally, using `{{` on regular objects would be an error. However for technical reasons it is not possible to make a distinction between function arguments and ordinary variables. See Does curly-curly work on regular objects? for more information about this limitation.

Allow overriding default names:

The implementation of `my_mean()` in the previous section forces a default name onto the result. But what if the caller wants to give it a different name? In functions that take dots, it is possible to just supply a named expression to override the default. In a function like `my_mean()` that takes a named argument we need a different approach.

This is where `englue()` becomes useful. We can pull out the default name creation in another user-facing argument like this:

```
my_mean <- function(data, var, name = englue("{{ var }}")) {
  data %>% dplyr::summarise("{name}" := mean({{ var }}))
}
```

Now the user may supply their own name if needed:

```
mtcars %>% my_mean(cyl * am)
#> # A tibble: 1 x 1
#>   `cyl * am`
#>        <dbl>
#> 1       2.06

mtcars %>% my_mean(cyl * am, name = "mean_cyl_am")
```

```
#> # A tibble: 1 x 1
#>   mean_cyl_am
#>         <dbl>
#> 1        2.06
```

What's the deal with `:=`?:

Name injection in dynamic dots was originally implemented with `:=` instead of `=` to allow complex expressions on the LHS:

```
x <- "name"
list2(!!x := 1)
#> $name
#> [1] 1
```

Name-injection with glue operations was an extension of this existing feature and so inherited the same interface. However, there is no technical barrier to using glue strings on the LHS of `=`.

Using glue syntax in packages:

Since rlang does not depend directly on glue, you will have to ensure that glue is installed by adding it to your `Imports:` section.

```
usethis::use_package("glue", "Imports")
```

---

hash *Hashing*

Description

- `hash()` hashes an arbitrary R object.
- `hash_file()` hashes the data contained in a file.

The generated hash is guaranteed to be reproducible across platforms that have the same endianness and are using the same R version.

Usage

```
hash(x)

hash_file(path)
```

Arguments

`x`	An object.
`path`	A character vector of paths to the files to be hashed.

Details

These hashers use the XXH128 hash algorithm of the xxHash library, which generates a 128-bit hash. Both are implemented as streaming hashes, which generate the hash with minimal extra memory usage.

For hash(), objects are converted to binary using R's native serialization tools. On R >= 3.5.0, serialization version 3 is used, otherwise version 2 is used. See serialize() for more information about the serialization version.

Value

- For hash(), a single character string containing the hash.
- For hash_file(), a character vector containing one hash per file.

Examples

```
hash(c(1, 2, 3))
hash(mtcars)

authors <- file.path(R.home("doc"), "AUTHORS")
copying <- file.path(R.home("doc"), "COPYING")
hashes <- hash_file(c(authors, copying))
hashes

# If you need a single hash for multiple files,
# hash the result of `hash_file()`
hash(hashes)
```

---

has_name *Does an object have an element with this name?*

Description

This function returns a logical value that indicates if a data frame or another named object contains an element with a specific name. Note that has_name() only works with vectors. For instance, environments need the specialised function env_has().

Usage

```
has_name(x, name)
```

Arguments

x	A data frame or another named object
name	Element name(s) to check

Details

Unnamed objects are treated as if all names are empty strings. NA input gives FALSE as output.

Value

A logical vector of the same length as name

Examples

```
has_name(iris, "Species")
has_name(mtcars, "gears")
```

---

inherits_any — *Does an object inherit from a set of classes?*

Description

- inherits_any() is like base::inherits() but is more explicit about its behaviour with multiple classes. If classes contains several elements and the object inherits from at least one of them, inherits_any() returns TRUE.
- inherits_all() tests that an object inherits from all of the classes in the supplied order. This is usually the best way to test for inheritance of multiple classes.
- inherits_only() tests that the class vectors are identical. It is a shortcut for identical(class(x), class).

Usage

```
inherits_any(x, class)

inherits_all(x, class)

inherits_only(x, class)
```

Arguments

x	An object to test for inheritance.
class	A character vector of classes.

Examples

```
obj <- structure(list(), class = c("foo", "bar", "baz"))

# With the _any variant only one class must match:
inherits_any(obj, c("foobar", "bazbaz"))
inherits_any(obj, c("foo", "bazbaz"))

# With the _all variant all classes must match:
inherits_all(obj, c("foo", "bazbaz"))
inherits_all(obj, c("foo", "baz"))

# The order of classes must match as well:
```

```
inherits_all(obj, c("baz", "foo"))

# inherits_only() checks that the class vectors are identical:
inherits_only(obj, c("foo", "baz"))
inherits_only(obj, c("foo", "bar", "baz"))
```

`inject` *Inject objects in an R expression*

Description

`inject()` evaluates an expression with injection support. There are three main usages:

- Splicing lists of arguments in a function call.
- Inline objects or other expressions in an expression with `!!` and `!!!`. For instance to create functions or formulas programmatically.
- Pass arguments to NSE functions that defuse their arguments without injection support (see for instance `enquo0()`). You can use `{{ arg }}` with functions documented to support quosures. Otherwise, use `!!enexpr(arg)`.

Usage

```
inject(expr, env = caller_env())
```

Arguments

`expr`	An argument to evaluate. This argument is immediately evaluated in `env` (the current environment by default) with injected objects and expressions.
`env`	The environment in which to evaluate `expr`. Defaults to the current environment. For expert use only.

Examples

```
# inject() simply evaluates its argument with injection
# support. These expressions are equivalent:
2 * 3
inject(2 * 3)
inject(!!2 * !!3)

# Injection with `!!` can be useful to insert objects or
# expressions within other expressions, like formulas:
lhs <- sym("foo")
rhs <- sym("bar")
inject(!!lhs ~ !!rhs + 10)

# Injection with `!!!` splices lists of arguments in function
# calls:
args <- list(na.rm = TRUE, finite = 0.2)
inject(mean(1:10, !!!args))
```

injection-operator — *Injection operator* `!!`

Description

The injection operator `!!` injects a value or expression inside another expression. In other words, it modifies a piece of code before R evaluates it.

There are two main cases for injection. You can inject constant values to work around issues of scoping ambiguity, and you can inject defused expressions like symbolised column names.

Where does `!!` work?

`!!` does not work everywhere, you can only use it within certain special functions:

- Functions taking defused and data-masked arguments.

  Technically, this means function arguments defused with `{{` or en-prefixed operators like `enquo()`, `enexpr()`, etc.
- Inside `inject()`.

All data-masking verbs in the tidyverse support injection operators out of the box. With base functions, you need to use `inject()` to enable `!!`. Using `!!` out of context may lead to incorrect results, see What happens if I use injection operators out of context?.

The examples below are built around the base function `with()`. Since it's not a tidyverse function we will use `inject()` to enable `!!` usage.

Injecting values

Data-masking functions like `with()` are handy because you can refer to column names in your computations. This comes at the price of data mask ambiguity: if you have defined an env-variable of the same name as a data-variable, you get a name collisions. This collision is always resolved by giving precedence to the data-variable (it masks the env-variable):

```
cyl <- c(100, 110)
with(mtcars, mean(cyl))
#> [1] 6.1875
```

The injection operator offers one way of solving this. Use it to inject the env-variable inside the data-masked expression:

```
inject(
  with(mtcars, mean(!!cyl))
)
#> [1] 105
```

Note that the `.env` pronoun is a simpler way of solving the ambiguity. See The data mask ambiguity for more about this.

Injecting expressions

Injection is also useful for modifying parts of a defused expression. In the following example we use the symbolise-and-inject pattern to inject a column name inside a data-masked expression.

```
var <- sym("cyl")
inject(
  with(mtcars, mean(!!var))
)
#> [1] 6.1875
```

Since `with()` is a base function, you can't inject quosures, only naked symbols and calls. This isn't a problem here because we're injecting the name of a data frame column. If the environment is important, try injecting a pre-computed value instead.

When do I need `!!`?

With tidyverse APIs, injecting expressions with `!!` is no longer a common pattern. First, the `.env` pronoun solves the ambiguity problem in a more intuitive way:

```
cyl <- 100
mtcars %>% dplyr::mutate(cyl = cyl * .env$cyl)
```

Second, the embrace operator `{{` makes the defuse-and-inject pattern easier to learn and use.

```
my_mean <- function(data, var) {
  data %>% dplyr::summarise(mean({{ var }}))
}

# Equivalent to
my_mean <- function(data, var) {
  data %>% dplyr::summarise(mean(!!enquo(var)))
}
```

`!!` is a good tool to learn for advanced applications but our hope is that it isn't needed for common data analysis cases.

See Also

- Injecting with !!, !!!, and glue syntax
- Metaprogramming patterns

`is_call` *Is object a call?*

Description

This function tests if `x` is a call. This is a pattern-matching predicate that returns `FALSE` if `name` and `n` are supplied and the call does not match these properties.

Usage

```
is_call(x, name = NULL, n = NULL, ns = NULL)
```

Arguments

`x`	An object to test. Formulas and quosures are treated literally.
`name`	An optional name that the call should match. It is passed to `sym()` before matching. This argument is vectorised and you can supply a vector of names to match. In this case, `is_call()` returns `TRUE` if at least one name matches.
`n`	An optional number of arguments that the call should match.
`ns`	The namespace of the call. If `NULL`, the namespace doesn't participate in the pattern-matching. If an empty string `""` and `x` is a namespaced call, `is_call()` returns `FALSE`. If any other string, `is_call()` checks that `x` is namespaced within `ns`. Can be a character vector of namespaces, in which case the call has to match at least one of them, otherwise `is_call()` returns `FALSE`.

See Also

`is_expression()`

Examples

```
is_call(quote(foo(bar)))

# You can pattern-match the call with additional arguments:
is_call(quote(foo(bar)), "foo")
is_call(quote(foo(bar)), "bar")
is_call(quote(foo(bar)), quote(foo))

# Match the number of arguments with is_call():
is_call(quote(foo(bar)), "foo", 1)
is_call(quote(foo(bar)), "foo", 2)


# By default, namespaced calls are tested unqualified:
ns_expr <- quote(base::list())
is_call(ns_expr, "list")
```

```
# You can also specify whether the call shouldn't be namespaced by
# supplying an empty string:
is_call(ns_expr, "list", ns = "")

# Or if it should have a namespace:
is_call(ns_expr, "list", ns = "utils")
is_call(ns_expr, "list", ns = "base")

# You can supply multiple namespaces:
is_call(ns_expr, "list", ns = c("utils", "base"))
is_call(ns_expr, "list", ns = c("utils", "stats"))

# If one of them is "", unnamespaced calls will match as well:
is_call(quote(list()), "list", ns = "base")
is_call(quote(list()), "list", ns = c("base", ""))
is_call(quote(base::list()), "list", ns = c("base", ""))


# The name argument is vectorised so you can supply a list of names
# to match with:
is_call(quote(foo(bar)), c("bar", "baz"))
is_call(quote(foo(bar)), c("bar", "foo"))
is_call(quote(base::list), c("::", ":::", "$", "@"))
```

`is_empty` *Is object an empty vector or NULL?*

Description

Is object an empty vector or NULL?

Usage

```
is_empty(x)
```

Arguments

x	object to test

Examples

```
is_empty(NULL)
is_empty(list())
is_empty(list(NULL))
```

is_environment *Is object an environment?*

Description

`is_bare_environment()` tests whether `x` is an environment without a s3 or s4 class.

Usage

```
is_environment(x)

is_bare_environment(x)
```

Arguments

x	object to test

is_expression *Is an object an expression?*

Description

In rlang, an *expression* is the return type of `parse_expr()`, the set of objects that can be obtained from parsing R code. Under this definition expressions include numbers, strings, `NULL`, symbols, and function calls. These objects can be classified as:

- Symbolic objects, i.e. symbols and function calls (for which `is_symbolic()` returns `TRUE`)
- Syntactic literals, i.e. scalar atomic objects and `NULL` (testable with `is_syntactic_literal()`)

`is_expression()` returns `TRUE` if the input is either a symbolic object or a syntactic literal. If a call, the elements of the call must all be expressions as well. Unparsable calls are not considered expressions in this narrow definition.

Note that in base R, there exists `expression()` vectors, a data type similar to a list that supports special attributes created by the parser called source references. This data type is not supported in rlang.

Usage

```
is_expression(x)

is_syntactic_literal(x)

is_symbolic(x)
```

Arguments

x	An object to test.

Details

`is_symbolic()` returns `TRUE` for symbols and calls (objects with type `language`). Symbolic objects are replaced by their value during evaluation. Literals are the complement of symbolic objects. They are their own value and return themselves during evaluation.

`is_syntactic_literal()` is a predicate that returns `TRUE` for the subset of literals that are created by R when parsing text (see `parse_expr()`): numbers, strings and `NULL`. Along with symbols, these literals are the terminating nodes in an AST.

Note that in the most general sense, a literal is any R object that evaluates to itself and that can be evaluated in the empty environment. For instance, `quote(c(1, 2))` is not a literal, it is a call. However, the result of evaluating it in `base_env()` is a literal(in this case an atomic vector).

As the data structure for function arguments, pairlists are also a kind of language objects. However, since they are mostly an internal data structure and can't be returned as is by the parser, `is_expression()` returns `FALSE` for pairlists.

See Also

`is_call()` for a call predicate.

Examples

```
q1 <- quote(1)
is_expression(q1)
is_syntactic_literal(q1)

q2 <- quote(x)
is_expression(q2)
is_symbol(q2)

q3 <- quote(x + 1)
is_expression(q3)
is_call(q3)


# Atomic expressions are the terminating nodes of a call tree:
# NULL or a scalar atomic vector:
is_syntactic_literal("string")
is_syntactic_literal(NULL)

is_syntactic_literal(letters)
is_syntactic_literal(quote(call()))

# Parsable literals have the property of being self-quoting:
identical("foo", quote("foo"))
identical(1L, quote(1L))
identical(NULL, quote(NULL))
```

```
# Like any literals, they can be evaluated within the empty
# environment:
eval_bare(quote(1L), empty_env())

# Whereas it would fail for symbolic expressions:
# eval_bare(quote(c(1L, 2L)), empty_env())


# Pairlists are also language objects representing argument lists.
# You will usually encounter them with extracted formals:
fmls <- formals(is_expression)
typeof(fmls)

# Since they are mostly an internal data structure, is_expression()
# returns FALSE for pairlists, so you will have to check explicitly
# for them:
is_expression(fmls)
is_pairlist(fmls)
```

---

is_formula *Is object a formula?*

Description

`is_formula()` tests whether `x` is a call to `~`. `is_bare_formula()` tests in addition that `x` does not inherit from anything else than `"formula"`.

Note: When we first implemented `is_formula()`, we thought it best to treat unevaluated formulas as formulas by default (see section below). Now we think this default introduces too many edge cases in normal code. We recommend always supplying `scoped = TRUE`. Unevaluated formulas can be handled via a `is_call(x, "~")` branch.

Usage

```
is_formula(x, scoped = NULL, lhs = NULL)

is_bare_formula(x, scoped = TRUE, lhs = NULL)
```

Arguments

`x`	An object to test.
`scoped`	A boolean indicating whether the quosure is scoped, that is, has a valid environment attribute and inherits from `"formula"`. If `NULL`, the scope is not inspected.
`lhs`	A boolean indicating whether the formula has a left-hand side. If `NULL`, the LHS is not inspected and `is_formula()` returns `TRUE` for both one- and two-sided formulas.

Dealing with unevaluated formulas

At parse time, a formula is a simple call to ~ and it does not have a class or an environment. Once evaluated, the ~ call becomes a properly structured formula. Unevaluated formulas arise by quotation, e.g. ~~foo, quote(~foo), or substitute(arg) with arg being supplied a formula. Use the scoped argument to check whether the formula carries an environment.

Examples

```
is_formula(~10)
is_formula(10)

# If you don't supply `lhs`, both one-sided and two-sided formulas
# will return `TRUE`
is_formula(disp ~ am)
is_formula(~am)

# You can also specify whether you expect a LHS:
is_formula(disp ~ am, lhs = TRUE)
is_formula(disp ~ am, lhs = FALSE)
is_formula(~am, lhs = TRUE)
is_formula(~am, lhs = FALSE)

# Handling of unevaluated formulas is a bit tricky. These formulas
# are special because they don't inherit from `"formula"` and they
# don't carry an environment (they are not scoped):
f <- quote(~foo)
f_env(f)

# By default unevaluated formulas are treated as formulas
is_formula(f)

# Supply `scoped = TRUE` to ensure you have an evaluated formula
is_formula(f, scoped = TRUE)

# By default unevaluated formulas not treated as bare formulas
is_bare_formula(f)

# If you supply `scoped = TRUE`, they will be considered bare
# formulas even though they don't inherit from `"formula"`
is_bare_formula(f, scoped = TRUE)
```

---

is_function *Is object a function?*

Description

The R language defines two different types of functions: primitive functions, which are low-level, and closures, which are the regular kind of functions.

Usage

```
is_function(x)

is_closure(x)

is_primitive(x)

is_primitive_eager(x)

is_primitive_lazy(x)
```

Arguments

x	Object to be tested.

Details

Closures are functions written in R, named after the way their arguments are scoped within nested environments (see https://en.wikipedia.org/wiki/Closure_(computer_programming)). The root environment of the closure is called the closure environment. When closures are evaluated, a new environment called the evaluation frame is created with the closure environment as parent. This is where the body of the closure is evaluated. These closure frames appear on the evaluation stack, as opposed to primitive functions which do not necessarily have their own evaluation frame and never appear on the stack.

Primitive functions are more efficient than closures for two reasons. First, they are written entirely in fast low-level code. Second, the mechanism by which they are passed arguments is more efficient because they often do not need the full procedure of argument matching (dealing with positional versus named arguments, partial matching, etc). One practical consequence of the special way in which primitives are passed arguments is that they technically do not have formal arguments, and `formals()` will return `NULL` if called on a primitive function. Finally, primitive functions can either take arguments lazily, like R closures do, or evaluate them eagerly before being passed on to the C code. The former kind of primitives are called "special" in R terminology, while the latter is referred to as "builtin". `is_primitive_eager()` and `is_primitive_lazy()` allow you to check whether a primitive function evaluates arguments eagerly or lazily.

You will also encounter the distinction between primitive and internal functions in technical documentation. Like primitive functions, internal functions are defined at a low level and written in C. However, internal functions have no representation in the R language. Instead, they are called via a call to `base::.Internal()` within a regular closure. This ensures that they appear as normal R function objects: they obey all the usual rules of argument passing, and they appear on the evaluation stack as any other closures. As a result, `fn_fmls()` does not need to look in the `.ArgsEnv` environment to obtain a representation of their arguments, and there is no way of querying from R whether they are lazy ('special' in R terminology) or eager ('builtin').

You can call primitive functions with `.Primitive()` and internal functions with `.Internal()`. However, calling internal functions in a package is forbidden by CRAN's policy because they are considered part of the private API. They often assume that they have been called with correctly formed arguments, and may cause R to crash if you call them with unexpected objects.

Examples

```
# Primitive functions are not closures:
is_closure(base::c)
is_primitive(base::c)

# On the other hand, internal functions are wrapped in a closure
# and appear as such from the R side:
is_closure(base::eval)

# Both closures and primitives are functions:
is_function(base::c)
is_function(base::eval)

# Many primitive functions evaluate arguments eagerly:
is_primitive_eager(base::c)
is_primitive_eager(base::list)
is_primitive_eager(base::`+`)

# However, primitives that operate on expressions, like quote() or
# substitute(), are lazy:
is_primitive_lazy(base::quote)
is_primitive_lazy(base::substitute)
```

---

is_installed — *Are packages installed in any of the libraries?*

Description

These functions check that packages are installed with minimal side effects. If installed, the packages will be loaded but not attached.

- `is_installed()` doesn't interact with the user. It simply returns `TRUE` or `FALSE` depending on whether the packages are installed.
- In interactive sessions, `check_installed()` asks the user whether to install missing packages. If the user accepts, the packages are installed with `pak::pkg_install()` if available, or `utils::install.packages()` otherwise. If the session is non interactive or if the user chooses not to install the packages, the current evaluation is aborted.

You can disable the prompt by setting the `rlib_restart_package_not_found` global option to `FALSE`. In that case, missing packages always cause an error.

Usage

```
is_installed(pkg, ..., version = NULL, compare = NULL)

check_installed(
  pkg,
  reason = NULL,
```

```
  ...,
  version = NULL,
  compare = NULL,
  action = NULL,
  call = caller_env()
)
```

Arguments

pkg	The package names. Can include version requirements, e.g. "pkg (>= 1.0.0)".
...	These dots must be empty.
version	Minimum versions for pkg. If supplied, must be the same length as pkg. NA elements stand for any versions.
compare	A character vector of comparison operators to use for version. If supplied, must be the same length as version. If NULL, >= is used as default for all elements. NA elements in compare are also set to >= by default.
reason	Optional string indicating why is pkg needed. Appears in error messages (if non-interactive) and user prompts (if interactive).
action	An optional function taking pkg and ... arguments. It is called by check_installed() when the user chooses to update outdated packages. The function is passed the missing and outdated packages as a character vector of names.
call	The execution environment of a currently running function, e.g. caller_env(). The function will be mentioned in error messages as the source of the error. See the call argument of abort() for more information.

Value

is_installed() returns TRUE if *all* package names provided in pkg are installed, FALSE otherwise. check_installed() either doesn't return or returns NULL.

Handling package not found errors

check_installed() signals error conditions of class rlib_error_package_not_found. The error includes pkg and version fields. They are vectorised and may include several packages.

The error is signalled with a rlib_restart_package_not_found restart on the stack to allow handlers to install the required packages. To do so, add a calling handler for rlib_error_package_not_found, install the required packages, and invoke the restart without arguments. This restarts the check from scratch.

The condition is not signalled in non-interactive sessions, in the restarting case, or if the rlib_restart_package_not_found user option is set to FALSE.

Examples

```
is_installed("utils")
is_installed(c("base", "ggplot5"))
is_installed(c("base", "ggplot5"), version = c(NA, "5.1.0"))
```

is_integerish — *Is a vector integer-like?*

Description

These predicates check whether R considers a number vector to be integer-like, according to its own tolerance check (which is in fact delegated to the C library). This function is not adapted to data analysis, see the help for `base::is.integer()` for examples of how to check for whole numbers.

Things to consider when checking for integer-like doubles:

- This check can be expensive because the whole double vector has to be traversed and checked.
- Large double values may be integerish but may still not be coercible to integer. This is because integers in R only support values up to `2^31 - 1` while numbers stored as double can be much larger.

Usage

```
is_integerish(x, n = NULL, finite = NULL)

is_bare_integerish(x, n = NULL, finite = NULL)

is_scalar_integerish(x, finite = NULL)
```

Arguments

`x`	Object to be tested.
`n`	Expected length of a vector.
`finite`	Whether all values of the vector are finite. The non-finite values are `NA`, `Inf`, `-Inf` and `NaN`. Setting this to something other than `NULL` can be expensive because the whole vector needs to be traversed and checked.

See Also

`is_bare_numeric()` for testing whether an object is a base numeric type (a bare double or integer vector).

Examples

```
is_integerish(10L)
is_integerish(10.0)
is_integerish(10.0, n = 2)
is_integerish(10.000001)
is_integerish(TRUE)
```

is_interactive *Is R running interactively?*

Description

Like base::interactive(), is_interactive() returns TRUE when the function runs interactively and FALSE when it runs in batch mode. It also checks, in this order:

- The rlang_interactive global option. If set to a single TRUE or FALSE, is_interactive() returns that value immediately. This escape hatch is useful in unit tests or to manually turn on interactive features in RMarkdown outputs.
- Whether knitr or testthat is in progress, in which case is_interactive() returns FALSE.

with_interactive() and local_interactive() set the global option conveniently.

Usage

```
is_interactive()

local_interactive(value = TRUE, frame = caller_env())

with_interactive(expr, value = TRUE)
```

Arguments

value	A single TRUE or FALSE. This overrides the return value of is_interactive().
frame	The environment of a running function which defines the scope of the temporary options. When the function returns, the options are reset to their original values.
expr	An expression to evaluate with interactivity set to value.

is_named *Is object named?*

Description

- is_named() is a scalar predicate that checks that x has a names attribute and that none of the names are missing or empty (NA or "").
- is_named2() is like is_named() but always returns TRUE for empty vectors, even those that don't have a names attribute. In other words, it tests for the property that each element of a vector is named. is_named2() composes well with names2() whereas is_named() composes with names().
- have_name() is a vectorised variant.

Usage

```
is_named(x)

is_named2(x)

have_name(x)
```

Arguments

x	A vector to test.

Details

`is_named()` always returns `TRUE` for empty vectors because

Value

`is_named()` and `is_named2()` are scalar predicates that return `TRUE` or `FALSE`. `have_name()` is vectorised and returns a logical vector as long as the input.

Examples

```
# is_named() is a scalar predicate about the whole vector of names:
is_named(c(a = 1, b = 2))
is_named(c(a = 1, 2))

# Unlike is_named2(), is_named() returns `FALSE` for empty vectors
# that don't have a `names` attribute.
is_named(list())
is_named2(list())

# have_name() is a vectorised predicate
have_name(c(a = 1, b = 2))
have_name(c(a = 1, 2))

# Empty and missing names are treated as invalid:
invalid <- set_names(letters[1:5])
names(invalid)[1] <- ""
names(invalid)[3] <- NA

is_named(invalid)
have_name(invalid)

# A data frame normally has valid, unique names
is_named(mtcars)
have_name(mtcars)

# A matrix usually doesn't because the names are stored in a
# different attribute
mat <- matrix(1:4, 2)
colnames(mat) <- c("a", "b")
```

```
is_named(mat)
names(mat)
```

---

is_namespace *Is an object a namespace environment?*

Description

Is an object a namespace environment?

Usage

```
is_namespace(x)
```

Arguments

x	An object to test.

---

is_symbol *Is object a symbol?*

Description

Is object a symbol?

Usage

```
is_symbol(x, name = NULL)
```

Arguments

x	An object to test.
name	An optional name or vector of names that the symbol should match.

is_true *Is object identical to TRUE or FALSE?*

Description

These functions bypass R's automatic conversion rules and check that x is literally TRUE or FALSE.

Usage

```
is_true(x)

is_false(x)
```

Arguments

x	object to test

Examples

```
is_true(TRUE)
is_true(1)

is_false(FALSE)
is_false(0)
```

is_weakref *Is object a weak reference?*

Description

Is object a weak reference?

Usage

```
is_weakref(x)
```

Arguments

x	An object to test.

last_error — *Last* `abort()` *error*

Description

- `last_error()` returns the last error entraced by `abort()` or `global_entrace()`. The error is printed with a backtrace in simplified form.
- `last_trace()` is a shortcut to return the backtrace stored in the last error. This backtrace is printed in full form.

Usage

```
last_error()

last_trace(drop = NULL)
```

Arguments

drop	Whether to drop technical calls. These are hidden from users by default, set `drop` to `FALSE` to see the full backtrace.

See Also

- `rlang_backtrace_on_error` to control what is displayed when an error is thrown.
- `global_entrace()` to enable `last_error()` logging for all errors.
- `last_warnings()` and `last_messages()`.

last_warnings — *Display last messages and warnings*

Description

`last_warnings()` and `last_messages()` return a list of all warnings and messages that occurred during the last R command.

`global_entrace()` must be active in order to log the messages and warnings.

By default the warnings and messages are printed with a simplified backtrace, like `last_error()`. Use `summary()` to print the conditions with a full backtrace.

Usage

```
last_warnings(n = NULL)

last_messages(n = NULL)
```

Arguments

n How many warnings or messages to display. Defaults to all.

Examples

Enable backtrace capture with `global_entrace()`:

```
global_entrace()
```

Signal some warnings in nested functions. The warnings inform about which function emitted a warning but they don't provide information about the call stack:

```
f <- function() { warning("foo"); g() }
g <- function() { warning("bar", immediate. = TRUE); h() }
h <- function() warning("baz")

f()
#> Warning in g() : bar
#> Warning messages:
#> 1: In f() : foo
#> 2: In h() : baz
```

Call `last_warnings()` to see backtraces for each of these warnings:

```
last_warnings()
#> [[1]]
#> <warning/rlang_warning>
#> Warning in `f()`:
#> foo
#> Backtrace:
#>     x
#>  1. \-global f()
#>
#> [[2]]
#> <warning/rlang_warning>
#> Warning in `g()`:
#> bar
#> Backtrace:
#>     x
#>  1. \-global f()
#>  2.   \-global g()
#>
#> [[3]]
#> <warning/rlang_warning>
#> Warning in `h()`:
#> baz
#> Backtrace:
#>     x
```

```
#>  1. \-global f()
#>  2.   \-global g()
#>  3.     \-global h()
```

This works similarly with messages:

```
f <- function() { inform("Hey!"); g() }
g <- function() { inform("Hi!"); h() }
h <- function() inform("Hello!")

f()
#> Hey!
#> Hi!
#> Hello!

rlang::last_messages()
#> [[1]]
#> <message/rlang_message>
#> Message:
#> Hey!
#> ---
#> Backtrace:
#>     x
#>  1. \-global f()
#>
#> [[2]]
#> <message/rlang_message>
#> Message:
#> Hi!
#> ---
#> Backtrace:
#>     x
#>  1. \-global f()
#>  2.   \-global g()
#>
#> [[3]]
#> <message/rlang_message>
#> Message:
#> Hello!
#> ---
#> Backtrace:
#>     x
#>  1. \-global f()
#>  2.   \-global g()
#>  3.     \-global h()
```

See Also

last_error()

list2 *Collect dynamic dots in a list*

Description

list2(...) is equivalent to list(...) with a few additional features, collectively called dynamic dots. While list2() hard-code these features, dots_list() is a lower-level version that offers more control.

Usage

```
list2(...)

dots_list(
  ...,
  .named = FALSE,
  .ignore_empty = c("trailing", "none", "all"),
  .preserve_empty = FALSE,
  .homonyms = c("keep", "first", "last", "error"),
  .check_assign = FALSE
)
```

Arguments

...	Arguments to collect in a list. These dots are dynamic.
.named	If TRUE, unnamed inputs are automatically named with as_label(). This is equivalent to applying exprs_auto_name() on the result. If FALSE, unnamed elements are left as is and, if fully unnamed, the list is given minimal names (a vector of ""). If NULL, fully unnamed results are left with NULL names.
.ignore_empty	Whether to ignore empty arguments. Can be one of "trailing", "none", "all". If "trailing", only the last argument is ignored if it is empty.
.preserve_empty	Whether to preserve the empty arguments that were not ignored. If TRUE, empty arguments are stored with missing_arg() values. If FALSE (the default) an error is thrown when an empty argument is detected.
.homonyms	How to treat arguments with the same name. The default, "keep", preserves these arguments. Set .homonyms to "first" to only keep the first occurrences, to "last" to keep the last occurrences, and to "error" to raise an informative error and indicate what arguments have duplicated names.
.check_assign	Whether to check for <- calls. When TRUE a warning recommends users to use = if they meant to match a function parameter or wrap the <- call in curly braces otherwise. This ensures assignments are explicit.

Details

For historical reasons, dots_list() creates a named list by default. By comparison list2() implements the preferred behaviour of only creating a names vector when a name is supplied.

Value

A list containing the ... inputs.

Examples

```
# Let's create a function that takes a variable number of arguments:
numeric <- function(...) {
  dots <- list2(...)
  num <- as.numeric(dots)
  set_names(num, names(dots))
}
numeric(1, 2, 3)

# The main difference with list(...) is that list2(...) enables
# the `!!!` syntax to splice lists:
x <- list(2, 3)
numeric(1, !!! x, 4)

# As well as unquoting of names:
nm <- "yup!"
numeric(!!nm := 1)


# One useful application of splicing is to work around exact and
# partial matching of arguments. Let's create a function taking
# named arguments and dots:
fn <- function(data, ...) {
  list2(...)
}

# You normally cannot pass an argument named `data` through the dots
# as it will match `fn`'s `data` argument. The splicing syntax
# provides a workaround:
fn("wrong!", data = letters)  # exact matching of `data`
fn("wrong!", dat = letters)   # partial matching of `data`
fn(some_data, !!!list(data = letters))  # no matching

# Empty trailing arguments are allowed:
list2(1, )

# But non-trailing empty arguments cause an error:
try(list2(1, , ))

# Use the more configurable `dots_list()` function to preserve all
# empty arguments:
list3 <- function(...) dots_list(..., .preserve_empty = TRUE)

# Note how the last empty argument is still ignored because
# `.ignore_empty` defaults to "trailing":
list3(1, , )

# The list with preserved empty arguments is equivalent to:
```

```
list(1, missing_arg())


# Arguments with duplicated names are kept by default:
list2(a = 1, a = 2, b = 3, b = 4, 5, 6)

# Use the `.homonyms` argument to keep only the first of these:
dots_list(a = 1, a = 2, b = 3, b = 4, 5, 6, .homonyms = "first")

# Or the last:
dots_list(a = 1, a = 2, b = 3, b = 4, 5, 6, .homonyms = "last")

# Or raise an informative error:
try(dots_list(a = 1, a = 2, b = 3, b = 4, 5, 6, .homonyms = "error"))


# dots_list() can be configured to warn when a `<-` call is
# detected:
my_list <- function(...) dots_list(..., .check_assign = TRUE)
my_list(a <- 1)

# There is no warning if the assignment is wrapped in braces.
# This requires users to be explicit about their intent:
my_list({ a <- 1 })
```

`local_bindings` *Temporarily change bindings of an environment*

Description

- `local_bindings()` temporarily changes bindings in `.env` (which is by default the caller environment). The bindings are reset to their original values when the current frame (or an arbitrary one if you specify `.frame`) goes out of scope.
- `with_bindings()` evaluates `expr` with temporary bindings. When `with_bindings()` returns, bindings are reset to their original values. It is a simple wrapper around `local_bindings()`.

Usage

```
local_bindings(..., .env = .frame, .frame = caller_env())

with_bindings(.expr, ..., .env = caller_env())
```

Arguments

`...`	Pairs of names and values. These dots support splicing (with value semantics) and name unquoting.
`.env`	An environment.

`.frame`	The frame environment that determines the scope of the temporary bindings. When that frame is popped from the call stack, bindings are switched back to their original values.
`.expr`	An expression to evaluate with temporary bindings.

Value

`local_bindings()` returns the values of old bindings invisibly; `with_bindings()` returns the value of `expr`.

Examples

```
foo <- "foo"
bar <- "bar"

# `foo` will be temporarily rebinded while executing `expr`
with_bindings(paste(foo, bar), foo = "rebinded")
paste(foo, bar)
```

`local_error_call` *Set local error call in an execution environment*

Description

`local_error_call()` is an alternative to explicitly passing a `call` argument to `abort()`. It sets the call (or a value that indicates where to find the call, see below) in a local binding that is automatically picked up by `abort()`.

Usage

```
local_error_call(call, frame = caller_env())
```

Arguments

`call` This can be:

- A call to be used as context for an error thrown in that execution environment.
- The `NULL` value to show no context.
- An execution environment, e.g. as returned by `caller_env()`. The `sys.call()` for that environment is taken as context.

`frame` The execution environment in which to set the local error call.

Motivation for setting local error calls

By default abort() uses the function call of its caller as context in error messages:

```
foo <- function() abort("Uh oh.")
foo()
#> Error in `foo()`: Uh oh.
```

This is not always appropriate. For example a function that checks an input on the behalf of another function should reference the latter, not the former:

```
arg_check <- function(arg,
                      error_arg = as_string(substitute(arg))) {
  abort(cli::format_error("{.arg {error_arg}} is failing."))
}

foo <- function(x) arg_check(x)
foo()
#> Error in `arg_check()`: `x` is failing.
```

The mismatch is clear in the example above. `arg_check()` does not have any `x` argument and so it is confusing to present `arg_check()` as being the relevant context for the failure of the `x` argument.

One way around this is to take a `call` or `error_call` argument and pass it to `abort()`. Here we name this argument `error_call` for consistency with `error_arg` which is prefixed because there is an existing `arg` argument. In other situations, taking `arg` and `call` arguments might be appropriate.

```
arg_check <- function(arg,
                      error_arg = as_string(substitute(arg)),
                      error_call = caller_env()) {
  abort(
    cli::format_error("{.arg {error_arg}} is failing."),
    call = error_call
  )
}

foo <- function(x) arg_check(x)
foo()
#> Error in `foo()`: `x` is failing.
```

This is the generally recommended pattern for argument checking functions. If you mention an argument in an error message, provide your callers a way to supply a different argument name and a different error call. `abort()` stores the error call in the `call` condition field which is then used to generate the "in" part of error messages.

In more complex cases it's often burdensome to pass the relevant call around, for instance if your checking and throwing code is structured into many different functions. In this case, use `local_error_call()` to set the call locally or instruct `abort()` to climb the call stack one level to find the relevant call. In the following example, the complexity is not so important that sparing the argument passing makes a big difference. However this illustrates the pattern:

```
arg_check <- function(arg,
                      error_arg = caller_arg(arg),
                      error_call = caller_env()) {
  # Set the local error call
  local_error_call(error_call)

  my_classed_stop(
    cli::format_error("{.arg {error_arg}} is failing.")
  )
}

my_classed_stop <- function(message) {
  # Forward the local error call to the caller's
  local_error_call(caller_env())

  abort(message, class = "my_class")
}

foo <- function(x) arg_check(x)
foo()
#> Error in `foo()`: `x` is failing.
```

Error call flags in performance-critical functions

The call argument can also be the string "caller". This is equivalent to caller_env() or parent.frame() but has a lower overhead because call stack introspection is only performed when an error is triggered. Note that eagerly calling caller_env() is fast enough in almost all cases.

If your function needs to be really fast, assign the error call flag directly instead of calling local_error_call():

```
.__error_call__. <- "caller"
```

Examples

```
# Set a context for error messages
function() {
  local_error_call(quote(foo()))
  local_error_call(sys.call())
}

# Disable the context
function() {
  local_error_call(NULL)
}

# Use the caller's context
function() {
  local_error_call(caller_env())
}
```

local_options *Change global options*

Description

- `local_options()` changes options for the duration of a stack frame (by default the current one). Options are set back to their old values when the frame returns.
- `with_options()` changes options while an expression is evaluated. Options are restored when the expression returns.
- `push_options()` adds or changes options permanently.
- `peek_option()` and `peek_options()` return option values. The former returns the option directly while the latter returns a list.

Usage

```
local_options(..., .frame = caller_env())

with_options(.expr, ...)

push_options(...)

peek_options(...)

peek_option(name)
```

Arguments

`...`	For `local_options()` and `push_options()`, named values defining new option values. For `peek_options()`, strings or character vectors of option names.
`.frame`	The environment of a stack frame which defines the scope of the temporary options. When the frame returns, the options are set back to their original values.
`.expr`	An expression to evaluate with temporary options.
`name`	An option name as string.

Value

For `local_options()` and `push_options()`, the old option values. `peek_option()` returns the current value of an option while the plural `peek_options()` returns a list of current option values.

Life cycle

These functions are experimental.

Examples

```
# Store and retrieve a global option:
push_options(my_option = 10)
peek_option("my_option")

# Change the option temporarily:
with_options(my_option = 100, peek_option("my_option"))
peek_option("my_option")

# The scoped variant is useful within functions:
fn <- function() {
  local_options(my_option = 100)
  peek_option("my_option")
}
fn()
peek_option("my_option")

# The plural peek returns a named list:
peek_options("my_option")
peek_options("my_option", "digits")
```

---

missing_arg *Generate or handle a missing argument*

Description

These functions help using the missing argument as a regular R object.

- `missing_arg()` generates a missing argument.
- `is_missing()` is like `base::missing()` but also supports testing for missing arguments contained in other objects like lists. It is also more consistent with default arguments which are never treated as missing (see section below).
- `maybe_missing()` is useful to pass down an input that might be missing to another function, potentially substituting by a default value. It avoids triggering an "argument is missing" error.

Usage

```
missing_arg()

is_missing(x)

maybe_missing(x, default = missing_arg())
```

Arguments

`x`	An object that might be the missing argument.
`default`	The object to return if the input is missing, defaults to `missing_arg()`.

Other ways to reify the missing argument

- `base::quote(expr = )` is the canonical way to create a missing argument object.
- `expr()` called without argument creates a missing argument.
- `quo()` called without argument creates an empty quosure, i.e. a quosure containing the missing argument object.

`is_missing()` and default arguments

The base function `missing()` makes a distinction between default values supplied explicitly and default values generated through a missing argument:

```
fn <- function(x = 1) base::missing(x)

fn()
#> [1] TRUE
fn(1)
#> [1] FALSE
```

This only happens within a function. If the default value has been generated in a calling function, it is never treated as missing:

```
caller <- function(x = 1) fn(x)
caller()
#> [1] FALSE
```

`rlang::is_missing()` simplifies these rules by never treating default arguments as missing, even in internal contexts:

```
fn <- function(x = 1) rlang::is_missing(x)

fn()
#> [1] FALSE
fn(1)
#> [1] FALSE
```

This is a little less flexible because you can't specialise behaviour based on implicitly supplied default values. However, this makes the behaviour of `is_missing()` and functions using it simpler to understand.

Fragility of the missing argument object

The missing argument is an object that triggers an error if and only if it is the result of evaluating a symbol. No error is produced when a function call evaluates to the missing argument object. For instance, it is possible to bind the missing argument to a variable with an expression like `x[[1]] <- missing_arg()`. Likewise, `x[[1]]` is safe to use as argument, e.g. `list(x[[1]])` even when the result is the missing object.

However, as soon as the missing argument is passed down between functions through a bare variable, it is likely to cause a missing argument error:

```
x <- missing_arg()
list(x)
#> Error:
#> ! argument "x" is missing, with no default
```

To work around this, `is_missing()` and `maybe_missing(x)` use a bit of magic to determine if the input is the missing argument without triggering a missing error.

```
x <- missing_arg()
list(maybe_missing(x))
#> [[1]]
#>
```

`maybe_missing()` is particularly useful for prototyping meta-programming algorithms in R. The missing argument is a likely input when computing on the language because it is a standard object in formals lists. While C functions are always allowed to return the missing argument and pass it to other C functions, this is not the case on the R side. If you're implementing your meta-programming algorithm in R, use `maybe_missing()` when an input might be the missing argument object.

Examples

```
# The missing argument usually arises inside a function when the
# user omits an argument that does not have a default:
fn <- function(x) is_missing(x)
fn()

# Creating a missing argument can also be useful to generate calls
args <- list(1, missing_arg(), 3, missing_arg())
quo(fn(!!! args))

# Other ways to create that object include:
quote(expr = )
expr()

# It is perfectly valid to generate and assign the missing
# argument in a list.
x <- missing_arg()
l <- list(missing_arg())

# Just don't evaluate a symbol that contains the empty argument.
# Evaluating the object `x` that we created above would trigger an
# error.
# x  # Not run

# On the other hand accessing a missing argument contained in a
# list does not trigger an error because subsetting is a function
# call:
l[[1]]
is.null(l[[1]])

# In case you really need to access a symbol that might contain the
```

```
# empty argument object, use maybe_missing():
maybe_missing(x)
is.null(maybe_missing(x))
is_missing(maybe_missing(x))


# Note that base::missing() only works on symbols and does not
# support complex expressions. For this reason the following lines
# would throw an error:

#> missing(missing_arg())
#> missing(l[[1]])

# while is_missing() will work as expected:
is_missing(missing_arg())
is_missing(l[[1]])
```

---

names2 *Get names of a vector*

Description

`names2()` always returns a character vector, even when an object does not have a `names` attribute. In this case, it returns a vector of empty names `""`. It also standardises missing names to `""`.

The replacement variant `names2<-` never adds `NA` names and instead fills unnamed vectors with `""`.

Usage

```
names2(x)

names2(x) <- value
```

Arguments

`x`	A vector.
`value`	New names.

Examples

```
names2(letters)

# It also takes care of standardising missing names:
x <- set_names(1:3, c("a", NA, "b"))
names2(x)

# Replacing names with the base `names<-` function may introduce
# `NA` values when the vector is unnamed:
x <- 1:3
names(x)[1:2] <- "foo"
```

```
names(x)

# Use the `names2<-` variant to avoid this
x <- 1:3
names2(x)[1:2] <- "foo"
names(x)
```

---

new_formula *Create a formula*

Description

Create a formula

Usage

```
new_formula(lhs, rhs, env = caller_env())
```

Arguments

lhs, rhs	A call, name, or atomic vector.
env	An environment.

Value

A formula object.

See Also

new_quosure()

Examples

```
new_formula(quote(a), quote(b))
new_formula(NULL, quote(b))
```

new_function *Create a function*

Description

This constructs a new function given its three components: list of arguments, body code and parent environment.

Usage

```
new_function(args, body, env = caller_env())
```

Arguments

args	A named list or pairlist of default arguments. Note that if you want arguments that don't have defaults, you'll need to use the special function `pairlist2()`. If you need quoted defaults, use `exprs()`.
body	A language object representing the code inside the function. Usually this will be most easily generated with `base::quote()`
env	The parent environment of the function, defaults to the calling environment of `new_function()`

Examples

```
f <- function() letters
g <- new_function(NULL, quote(letters))
identical(f, g)

# Pass a list or pairlist of named arguments to create a function
# with parameters. The name becomes the parameter name and the
# argument the default value for this parameter:
new_function(list(x = 10), quote(x))
new_function(pairlist2(x = 10), quote(x))

# Use `exprs()` to create quoted defaults. Compare:
new_function(pairlist2(x = 5 + 5), quote(x))
new_function(exprs(x = 5 + 5), quote(x))

# Pass empty arguments to omit defaults. `list()` doesn't allow
# empty arguments but `pairlist2()` does:
new_function(pairlist2(x = , y = 5 + 5), quote(x + y))
new_function(exprs(x = , y = 5 + 5), quote(x + y))
```

new_quosure — *Create a quosure from components*

Description

- `new_quosure()` wraps any R object (including expressions, formulas, or other quosures) into a quosure.
- `as_quosure()` is similar but it does not rewrap formulas and quosures.

Usage

```
new_quosure(expr, env = caller_env())

as_quosure(x, env = NULL)

is_quosure(x)
```

Arguments

expr	An expression to wrap in a quosure.
env	The environment in which the expression should be evaluated. Only used for symbols and calls. This should normally be the environment in which the expression was created.
x	An object to test.

See Also

- `enquo()` and `quo()` for creating a quosure by argument defusal.
- What are quosures and when are they needed?

Examples

```
# `new_quosure()` creates a quosure from its components. These are
# equivalent:
new_quosure(quote(foo), current_env())

quo(foo)

# `new_quosure()` always rewraps its input into a new quosure, even
# if the input is itself a quosure:
new_quosure(quo(foo))

# This is unlike `as_quosure()` which preserves its input if it's
# already a quosure:
as_quosure(quo(foo))
```

```
# `as_quosure()` uses the supplied environment with naked expressions:
env <- env(var = "thing")
as_quosure(quote(var), env)

# If the expression already carries an environment, this
# environment is preserved. This is the case for formulas and
# quosures:
as_quosure(~foo, env)

as_quosure(~foo)

# An environment must be supplied when the input is a naked
# expression:
try(
  as_quosure(quote(var))
)
```

---

new_quosures *Create a list of quosures*

Description

This small S3 class provides methods for `[` and `c()` and ensures the following invariants:

- The list only contains quosures.
- It is always named, possibly with a vector of empty strings.

`new_quosures()` takes a list of quosures and adds the `quosures` class and a vector of empty names if needed. `as_quosures()` calls `as_quosure()` on all elements before creating the `quosures` object.

Usage

```
new_quosures(x)

as_quosures(x, env, named = FALSE)

is_quosures(x)
```

Arguments

x	A list of quosures or objects to coerce to quosures.
env	The default environment for the new quosures.
named	Whether to name the list with `quos_auto_name()`.

`new_weakref` *Create a weak reference*

Description

A weak reference is a special R object which makes it possible to keep a reference to an object without preventing garbage collection of that object. It can also be used to keep data about an object without preventing GC of the object, similar to WeakMaps in JavaScript.

Objects in R are considered *reachable* if they can be accessed by following a chain of references, starting from a *root node*; root nodes are specially-designated R objects, and include the global environment and base environment. As long as the key is reachable, the value will not be garbage collected. This is true even if the weak reference object becomes unreachable. The key effectively prevents the weak reference and its value from being collected, according to the following chain of ownership: `weakref <- key -> value`.

When the key becomes unreachable, the key and value in the weak reference object are replaced by `NULL`, and the finalizer is scheduled to execute.

Usage

```
new_weakref(key, value = NULL, finalizer = NULL, on_quit = FALSE)
```

Arguments

key	The key for the weak reference. Must be a reference object – that is, an environment or external pointer.
value	The value for the weak reference. This can be `NULL`, if you want to use the weak reference like a weak pointer.
finalizer	A function that is run after the key becomes unreachable.
on_quit	Should the finalizer be run when R exits?

See Also

`is_weakref()`, `wref_key()` and `wref_value()`.

Examples

```
e <- env()

# Create a weak reference to e
w <- new_weakref(e, finalizer = function(e) message("finalized"))

# Get the key object from the weak reference
identical(wref_key(w), e)

# When the regular reference (the `e` binding) is removed and a GC occurs,
# the weak reference will not keep the object alive.
rm(e)
```

```
gc()
identical(wref_key(w), NULL)


# A weak reference with a key and value. The value contains data about the
# key.
k <- env()
v <- list(1, 2, 3)
w <- new_weakref(k, v)

identical(wref_key(w), k)
identical(wref_value(w), v)

# When v is removed, the weak ref keeps it alive because k is still reachable.
rm(v)
gc()
identical(wref_value(w), list(1, 2, 3))

# When k is removed, the weak ref does not keep k or v alive.
rm(k)
gc()
identical(wref_key(w), NULL)
identical(wref_value(w), NULL)
```

---

on_load *Run expressions on load*

Description

- on_load() registers expressions to be run on the user's machine each time the package is loaded in memory. This is by contrast to normal R package code which is run once at build time on the packager's machine (e.g. CRAN).

  on_load() expressions require run_on_load() to be called inside .onLoad().
- on_package_load() registers expressions to be run each time another package is loaded.

on_load() is for your own package and runs expressions when the namespace is not *sealed* yet. This means you can modify existing binding or create new ones. This is not the case with on_package_load() which runs expressions after a foreign package has finished loading, at which point its namespace is sealed.

Usage

```
on_load(expr, env = parent.frame(), ns = topenv(env))

run_on_load(ns = topenv(parent.frame()))

on_package_load(pkg, expr, env = parent.frame())
```

Arguments

expr	An expression to run on load.
env	The environment in which to evaluate `expr`. Defaults to the current environment, which is your package namespace if you run `on_load()` at top level.
ns	The namespace in which to hook `expr`.
pkg	Package to hook expression into.

When should I run expressions on load?

There are two main use cases for running expressions on load:

1. When a side effect, such as registering a method with `s3_register()`, must occur in the user session rather than the package builder session.
2. To avoid hard-coding objects from other packages in your namespace. If you assign `foo::bar` or the result of `foo::baz()` in your package, they become constants. Any upstream changes in the `foo` package will not be reflected in the objects you've assigned in your namespace. This often breaks assumptions made by the authors of `foo` and causes all sorts of issues.

   Recreating the foreign objects each time your package is loaded makes sure that any such changes will be taken into account. In technical terms, running an expression on load introduces *indirection*.

Comparison with `.onLoad()`

`on_load()` has the advantage that hooked expressions can appear in any file, in context. This is unlike `.onLoad()` which gathers disparate expressions in a single block.

`on_load()` is implemented via `.onLoad()` and requires `run_on_load()` to be called from that hook.

The expressions inside `on_load()` do not undergo static analysis by `R CMD check`. Therefore, it is advisable to only use simple function calls inside `on_load()`.

Examples

```
quote({  # Not run

# First add `run_on_load()` to your `.onLoad()` hook,
# then use `on_load()` anywhere in your package
.onLoad <- function(lib, pkg) {
  run_on_load()
}

# Register a method on load
on_load({
  s3_register("foo::bar", "my_class")
})

# Assign an object on load
var <- NULL
on_load({
```

```
  var <- foo()
})

# To use `on_package_load()` at top level, wrap it in `on_load()`
on_load({
  on_package_load("foo", message("foo is loaded"))
})

# In functions it can be called directly
f <- function() on_package_load("foo", message("foo is loaded"))

})
```

---

op-get-attr *Infix attribute accessor and setter*

Description

This operator extracts or sets attributes for regular objects and S4 fields for S4 objects.

Usage

```
x %@% name

x %@% name <- value
```

Arguments

x	Object
name	Attribute name
value	New value for attribute name.

Examples

```
# Unlike `@`, this operator extracts attributes for any kind of
# objects:
factor(1:3) %@% "levels"
mtcars %@% class

mtcars %@% class <- NULL
mtcars

# It also works on S4 objects:
.Person <- setClass("Person", slots = c(name = "character", species = "character"))
fievel <- .Person(name = "Fievel", species = "mouse")
fievel %@% name
```

op-null-default *Default value for* NULL

Description

This infix function makes it easy to replace NULLs with a default value. It's inspired by the way that Ruby's or operation (||) works.

Usage

```
x %||% y
```

Arguments

x, y	If x is NULL, will return y; otherwise returns x.

Examples

```
1 %||% 2
NULL %||% 2
```

pairlist2 *Collect dynamic dots in a pairlist*

Description

This pairlist constructor uses dynamic dots. Use it to manually create argument lists for calls or parameter lists for functions.

Usage

```
pairlist2(...)
```

Arguments

...	<dynamic> Arguments stored in the pairlist. Empty arguments are preserved.

Examples

```
# Unlike `exprs()`, `pairlist2()` evaluates its arguments.
new_function(pairlist2(x = 1, y = 3 * 6), quote(x * y))
new_function(exprs(x = 1, y = 3 * 6), quote(x * y))

# It preserves missing arguments, which is useful for creating
# parameters without defaults:
new_function(pairlist2(x = , y = 3 * 6), quote(x * y))
```

parse_expr *Parse R code*

Description

These functions parse and transform text into R expressions. This is the first step to interpret or evaluate a piece of R code written by a programmer.

- `parse_expr()` returns one expression. If the text contains more than one expression (separated by semicolons or new lines), an error is issued. On the other hand `parse_exprs()` can handle multiple expressions. It always returns a list of expressions (compare to `base::parse()` which returns a base::expression vector). All functions also support R connections.
- `parse_expr()` concatenates `x` with `\\n` separators prior to parsing in order to support the roundtrip `parse_expr(expr_deparse(x))` (deparsed expressions might be multiline). On the other hand, `parse_exprs()` doesn't do any concatenation because it's designed to support named inputs. The names are matched to the expressions in the output, which is useful when a single named string creates multiple expressions.

  In other words, `parse_expr()` supports vector of lines whereas `parse_exprs()` expects vectors of complete deparsed expressions.
- `parse_quo()` and `parse_quos()` are variants that create a quosure. Supply `env = current_env()` if you're parsing code to be evaluated in your current context. Supply `env = global_env()` when you're parsing external user input to be evaluated in user context.

  Unlike quosures created with `enquo()`, `enquos()`, or `{{`, a parsed quosure never contains injected quosures. It is thus safe to evaluate them with `eval()` instead of `eval_tidy()`, though the latter is more convenient as you don't need to extract `expr` and `env`.

Usage

```
parse_expr(x)

parse_exprs(x)

parse_quo(x, env)

parse_quos(x, env)
```

Arguments

`x`	Text containing expressions to parse_expr for `parse_expr()` and `parse_exprs()`. Can also be an R connection, for instance to a file. If the supplied connection is not open, it will be automatically closed and destroyed.
`env`	The environment for the quosures. The global environment (the default) may be the right choice when you are parsing external user inputs. You might also want to evaluate the R code in an isolated context (perhaps a child of the global environment or of the base environment).

Details

Unlike base::parse(), these functions never retain source reference information, as doing so is slow and rarely necessary.

Value

parse_expr() returns an expression, parse_exprs() returns a list of expressions. Note that for the plural variants the length of the output may be greater than the length of the input. This would happen is one of the strings contain several expressions (such as "foo; bar"). The names of x are preserved (and recycled in case of multiple expressions). The _quo suffixed variants return quosures.

See Also

base::parse()

Examples

```
# parse_expr() can parse any R expression:
parse_expr("mtcars %>% dplyr::mutate(cyl_prime = cyl / sd(cyl))")

# A string can contain several expressions separated by ; or \n
parse_exprs("NULL; list()\n foo(bar)")

# Use names to figure out which input produced an expression:
parse_exprs(c(foo = "1; 2", bar = "3"))

# You can also parse source files by passing a R connection. Let's
# create a file containing R code:
path <- tempfile("my-file.R")
cat("1; 2; mtcars", file = path)

# We can now parse it by supplying a connection:
parse_exprs(file(path))
```

---

qq_show *Show injected expression*

Description

qq_show() helps examining injected expressions inside a function. This is useful for learning about injection and for debugging injection code.

Arguments

expr An expression involving injection operators.

Examples

qq_show() shows the intermediary expression before it is evaluated by R:

```
list2(!!!1:3)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3

qq_show(list2(!!!1:3))
#> list2(1L, 2L, 3L)
```

It is especially useful inside functions to reveal what an injected expression looks like:

```
my_mean <- function(data, var) {
  qq_show(data %>% dplyr::summarise(mean({{ var }})))
}

mtcars %>% my_mean(cyl)
#> data %>% dplyr::summarise(mean(^cyl))
```

See Also

- Injecting with !!, !!!, and glue syntax

---

quosure-tools — *Quosure getters, setters and predicates*

Description

These tools inspect and modify quosures, a type of defused expression that includes a reference to the context where it was created. A quosure is guaranteed to evaluate in its original environment and can refer to local objects safely.

- You can access the quosure components with quo_get_expr() and quo_get_env().
- The quo_ prefixed predicates test the expression of a quosure, quo_is_missing(), quo_is_symbol(), etc.

All quo_ prefixed functions expect a quosure and will fail if supplied another type of object. Make sure the input is a quosure with is_quosure().

Usage

```
quo_is_missing(quo)

quo_is_symbol(quo, name = NULL)

quo_is_call(quo, name = NULL, n = NULL, ns = NULL)

quo_is_symbolic(quo)

quo_is_null(quo)

quo_get_expr(quo)

quo_get_env(quo)

quo_set_expr(quo, expr)

quo_set_env(quo, env)
```

Arguments

quo	A quosure to test.
name	The name of the symbol or function call. If `NULL` the name is not tested.
n	An optional number of arguments that the call should match.
ns	The namespace of the call. If `NULL`, the namespace doesn't participate in the pattern-matching. If an empty string `""` and `x` is a namespaced call, `is_call()` returns `FALSE`. If any other string, `is_call()` checks that `x` is namespaced within `ns`. Can be a character vector of namespaces, in which case the call has to match at least one of them, otherwise `is_call()` returns `FALSE`.
expr	A new expression for the quosure.
env	A new environment for the quosure.

Empty quosures and missing arguments

When missing arguments are captured as quosures, either through `enquo()` or `quos()`, they are returned as an empty quosure. These quosures contain the missing argument and typically have the empty environment as enclosure.

Use `quo_is_missing()` to test for a missing argument defused with `enquo()`.

See Also

- `quo()` for creating quosures by argument defusal.
- `new_quosure()` and `as_quosure()` for assembling quosures from components.
- What are quosures and when are they needed? for an overview.

Examples

```
quo <- quo(my_quosure)
quo


# Access and set the components of a quosure:
quo_get_expr(quo)
quo_get_env(quo)

quo <- quo_set_expr(quo, quote(baz))
quo <- quo_set_env(quo, empty_env())
quo

# Test wether an object is a quosure:
is_quosure(quo)

# If it is a quosure, you can use the specialised type predicates
# to check what is inside it:
quo_is_symbol(quo)
quo_is_call(quo)
quo_is_null(quo)

# quo_is_missing() checks for a special kind of quosure, the one
# that contains the missing argument:
quo()
quo_is_missing(quo())

fn <- function(arg) enquo(arg)
fn()
quo_is_missing(fn())
```

---

quo_squash *Squash a quosure*

Description

`quo_squash()` flattens all nested quosures within an expression. For example it transforms `^foo(^bar(), ^baz)` to the bare expression `foo(bar(), baz)`.

This operation is safe if the squashed quosure is used for labelling or printing (see `as_label()`, but note that `as_label()` squashes quosures automatically). However if the squashed quosure is evaluated, all expressions of the flattened quosures are resolved in a single environment. This is a source of bugs so it is good practice to set `warn` to `TRUE` to let the user know about the lossy squashing.

Usage

```
quo_squash(quo, warn = FALSE)
```

Arguments

quo	A quosure or expression.
warn	Whether to warn if the quosure contains other quosures (those will be collapsed). This is useful when you use quo_squash() in order to make a non-tidyeval API compatible with quosures. In that case, getting rid of the nested quosures is likely to cause subtle bugs and it is good practice to warn the user about it.

Examples

```
# Quosures can contain nested quosures:
quo <- quo(wrapper(!!quo(wrappee)))
quo

# quo_squash() flattens all the quosures and returns a simple expression:
quo_squash(quo)
```

---

rep_along *Create vectors matching the length of a given vector*

Description

These functions take the idea of seq_along() and apply it to repeating values.

Usage

```
rep_along(along, x)

rep_named(names, x)
```

Arguments

along	Vector whose length determine how many times x is repeated.
x	Values to repeat.
names	Names for the new vector. The length of names determines how many times x is repeated.

See Also

new-vector

Examples

```
x <- 0:5
rep_along(x, 1:2)
rep_along(x, 1)

# Create fresh vectors by repeating missing values:
rep_along(x, na_int)
rep_along(x, na_chr)

# rep_named() repeats a value along a names vectors
rep_named(c("foo", "bar"), list(letters))
```

---

rlang_backtrace_on_error

Display backtrace on error

Description

rlang errors carry a backtrace that can be inspected by calling `last_error()`. You can also control the default display of the backtrace by setting the option `rlang_backtrace_on_error` to one of the following values:

- `"none"` show nothing.
- `"reminder"`, the default in interactive sessions, displays a reminder that you can see the backtrace with `last_error()`.
- `"branch"` displays a simplified backtrace.
- `"full"`, the default in non-interactive sessions, displays the full tree.

rlang errors are normally thrown with `abort()`. If you promote base errors to rlang errors with `global_entrace()`, `rlang_backtrace_on_error` applies to all errors.

Promote base errors to rlang errors

You can use `options(error = rlang::entrace)` to promote base errors to rlang errors. This does two things:

- It saves the base error as an rlang object so you can call `last_error()` to print the backtrace or inspect its data.
- It prints the backtrace for the current error according to the `rlang_backtrace_on_error` option.

Warnings and errors in RMarkdown

The display of errors depends on whether they're expected (i.e. chunk option `error = TRUE`) or unexpected:

- Expected errors are controlled by the global option "rlang_backtrace_on_error_report" (note the _report suffix). The default is "none" so that your expected errors don't include a reminder to run rlang::last_error(). Customise this option if you want to demonstrate what the error backtrace will look like.

  You can also use last_error() to display the trace like you would in your session, but it currently only works in the next chunk.

- Unexpected errors are controlled by the global option "rlang_backtrace_on_error". The default is "branch" so you'll see a simplified backtrace in the knitr output to help you figure out what went wrong.

When knitr is running (as determined by the knitr.in.progress global option), the default top environment for backtraces is set to the chunk environment knitr::knit_global(). This ensures that the part of the call stack belonging to knitr does not end up in backtraces. If needed, you can override this by setting the rlang_trace_top_env global option.

Similarly to rlang_backtrace_on_error_report, you can set rlang_backtrace_on_warning_report inside RMarkdown documents to tweak the display of warnings. This is useful in conjunction with global_entrace(). Because of technical limitations, there is currently no corresponding rlang_backtrace_on_warning option for normal R sessions.

To get full entracing in an Rmd document, include this in a setup chunk before the first error or warning is signalled.

````
```{r setup}
rlang::global_entrace()
options(rlang_backtrace_on_warning_report = "full")
options(rlang_backtrace_on_error_report = "full")
```
````

See Also

rlang_backtrace_on_warning

Examples

```
# Display a simplified backtrace on error for both base and rlang
# errors:

# options(
#   rlang_backtrace_on_error = "branch",
#   error = rlang::entrace
# )
# stop("foo")
```

rlang_error   *Errors of class* `rlang_error`

Description

`abort()` and `error_cnd()` create errors of class `"rlang_error"`. The differences with base errors are:

- Implementing `conditionMessage()` methods for subclasses of `"rlang_error"` is undefined behaviour. Instead, implement the `cnd_header()` method (and possibly `cnd_body()` and `cnd_footer()`). These methods return character vectors which are assembled by rlang when needed: when `conditionMessage.rlang_error()` is called (e.g. via `try()`), when the error is displayed through `print()` or `format()`, and of course when the error is displayed to the user by `abort()`.
- `cnd_header()`, `cnd_body()`, and `cnd_footer()` methods can be overridden by storing closures in the `header`, `body`, and `footer` fields of the condition. This is useful to lazily generate messages based on state captured in the closure environment.
- **[Experimental]** The `use_cli_format` condition field instructs whether to use cli (or rlang's fallback method if cli is not installed) to format the error message at print time.

  In this case, the `message` field may be a character vector of header and bullets. These are formatted at the last moment to take the context into account (starting position on the screen and indentation).

  See `local_use_cli()` for automatically setting this field in errors thrown with `abort()` within your package.

scalar-type-predicates   *Scalar type predicates*

Description

These predicates check for a given type and whether the vector is "scalar", that is, of length 1.

In addition to the length check, `is_string()` and `is_bool()` return `FALSE` if their input is missing. This is useful for type-checking arguments, when your function expects a single string or a single `TRUE` or `FALSE`.

Usage

```
is_scalar_list(x)

is_scalar_atomic(x)

is_scalar_vector(x)
```

```
is_scalar_integer(x)

is_scalar_double(x)

is_scalar_complex(x)

is_scalar_character(x)

is_scalar_logical(x)

is_scalar_raw(x)

is_string(x, string = NULL)

is_scalar_bytes(x)

is_bool(x)
```

Arguments

x	object to be tested.
string	A string to compare to x. If a character vector, returns TRUE if at least one element is equal to x.

See Also

type-predicates, bare-type-predicates

---

seq2 *Increasing sequence of integers in an interval*

Description

These helpers take two endpoints and return the sequence of all integers within that interval. For seq2_along(), the upper endpoint is taken from the length of a vector. Unlike base::seq(), they return an empty vector if the starting point is a larger integer than the end point.

Usage

```
seq2(from, to)

seq2_along(from, x)
```

Arguments

from	The starting point of the sequence.
to	The end point.
x	A vector whose length is the end point.

Value

An integer vector containing a strictly increasing sequence.

Examples

```
seq2(2, 10)
seq2(10, 2)
seq(10, 2)

seq2_along(10, letters)
```

---

set_names   *Set names of a vector*

Description

This is equivalent to stats::setNames(), with more features and stricter argument checking.

Usage

```
set_names(x, nm = x, ...)
```

Arguments

x	Vector to name.
nm, ...	Vector of names, the same length as x. If length 1, nm is recycled to the length of x following the recycling rules of the tidyverse.. You can specify names in the following ways: • If not supplied, x will be named to as.character(x). • If x already has names, you can provide a function or formula to transform the existing names. In that case, ... is passed to the function. • Otherwise if ... is supplied, x is named to c(nm, ...). • If nm is NULL, the names are removed (if present).

Life cycle

set_names() is stable and exported in purrr.

Examples

```
set_names(1:4, c("a", "b", "c", "d"))
set_names(1:4, letters[1:4])
set_names(1:4, "a", "b", "c", "d")

# If the second argument is ommitted a vector is named with itself
set_names(letters[1:5])
```

```
# Alternatively you can supply a function
set_names(1:10, ~ letters[seq_along(.)])
set_names(head(mtcars), toupper)

# If the input vector is unnamed, it is first named after itself
# before the function is applied:
set_names(letters, toupper)

# `...` is passed to the function:
set_names(head(mtcars), paste0, "_foo")

# If length 1, the second argument is recycled to the length of the first:
set_names(1:3, "foo")
set_names(list(), "")
```

---

splice *Splice values at dots collection time*

Description

splice() is an advanced feature of dynamic dots. It is rarely needed but can solve performance issues in edge cases.

The splicing operator !!! operates both in values contexts like list2() and dots_list(), and in metaprogramming contexts like expr(), enquos(), or inject(). While the end result looks the same, the implementation is different and much more efficient in the value cases. This difference in implementation may cause performance issues for instance when going from:

```
xs <- list(2, 3)
list2(1, !!!xs, 4)
```

to:

```
inject(list2(1, !!!xs, 4))
```

In the former case, the performant value-splicing is used. In the latter case, the slow metaprogramming splicing is used.

A common practical case where this may occur is when code is wrapped inside a tidyeval context like dplyr::mutate(). In this case, the metaprogramming operator !!! will take over the value-splicing operator, causing an unexpected slowdown.

To avoid this in performance-critical code, use splice() instead of !!!:

```
# These both use the fast splicing:
list2(1, splice(xs), 4)
inject(list2(1, splice(xs), 4))
```

Note that splice() behaves differently than !!!. The splicing happens later and is processed by list2() or dots_list(). It does not work in any other tidyeval context than these list collectors.

Usage

```
splice(x)

is_spliced(x)

is_spliced_bare(x)
```

Arguments

x	A list or vector to splice non-eagerly.

---

splice-operator *Splice operator* !!!

Description

The splice operator `!!!` implemented in dynamic dots injects a list of arguments into a function call. It belongs to the family of injection operators and provides the same functionality as `do.call()`.

The two main cases for splice injection are:

- Turning a list of inputs into distinct arguments. This is especially useful with functions that take data in `...`, such as `base::rbind()`.

  ```
  dfs <- list(mtcars, mtcars)
  inject(rbind(!!!dfs))
  ```

- Injecting defused expressions like symbolised column names.

  For tidyverse APIs, this second case is no longer as useful since dplyr 1.0 and the `across()` operator.

Where does `!!!` work?

`!!!` does not work everywhere, you can only use it within certain special functions:

- Functions taking dynamic dots like `list2()`.
- Functions taking defused and data-masked arguments, which are dynamic by default.
- Inside `inject()`.

Most tidyverse functions support `!!!` out of the box. With base functions you need to use `inject()` to enable `!!!`.

Using the operator out of context may lead to incorrect results, see What happens if I use injection operators out of context?.

Splicing a list of arguments

Take a function like base::rbind() that takes data in .... This sort of functions takes a variable number of arguments.

```
df1 <- data.frame(x = 1)
df2 <- data.frame(x = 2)

rbind(df1, df2)
#>   x
#> 1 1
#> 2 2
```

Passing individual arguments is only possible for a fixed amount of arguments. When the arguments are in a list whose length is variable (and potentially very large), we need a programmatic approach like the splicing syntax !!!:

```
dfs <- list(df1, df2)

inject(rbind(!!!dfs))
#>   x
#> 1 1
#> 2 2
```

Because rbind() is a base function we used inject() to explicitly enable !!!. However, many functions implement dynamic dots with !!! implicitly enabled out of the box.

```
tidyr::expand_grid(x = 1:2, y = c("a", "b"))
#> # A tibble: 4 x 2
#>       x y
#>   <int> <chr>
#> 1     1 a
#> 2     1 b
#> 3     2 a
#> 4     2 b

xs <- list(x = 1:2, y = c("a", "b"))
tidyr::expand_grid(!!!xs)
#> # A tibble: 4 x 2
#>       x y
#>   <int> <chr>
#> 1     1 a
#> 2     1 b
#> 3     2 a
#> 4     2 b
```

Note how the expanded grid has the right column names. That's because we spliced a *named* list. Splicing causes each name of the list to become an argument name.

```
tidyr::expand_grid(!!!set_names(xs, toupper))
#> # A tibble: 4 x 2
#>       X Y
#>   <int> <chr>
#> 1     1 a
#> 2     1 b
#> 3     2 a
#> 4     2 b
```

Splicing a list of expressions

Another usage for `!!!` is to inject defused expressions into data-masked dots. However this usage is no longer a common pattern for programming with tidyverse functions and we recommend using other patterns if possible.

First, instead of using the defuse-and-inject pattern with `...`, you can simply pass them on as you normally would. These two expressions are completely equivalent:

```
my_group_by <- function(.data, ...) {
  .data %>% dplyr::group_by(!!!enquos(...))
}

# This equivalent syntax is preferred
my_group_by <- function(.data, ...) {
  .data %>% dplyr::group_by(...)
}
```

Second, more complex applications such as transformation patterns can be solved with the `across()` operation introduced in dplyr 1.0. Say you want to take the `mean()` of all expressions in `...`. Before `across()`, you had to defuse the `...` expressions, wrap them in a call to `mean()`, and inject them in `summarise()`.

```
my_mean <- function(.data, ...) {
  # Defuse dots and auto-name them
  exprs <- enquos(..., .named = TRUE)

  # Wrap the expressions in a call to `mean()`
  exprs <- purrr::map(exprs, ~ call("mean", .x, na.rm = TRUE))

  # Inject them
  .data %>% dplyr::summarise(!!!exprs)
}
```

It is much easier to use `across()` instead:

```
my_mean <- function(.data, ...) {
  .data %>% dplyr::summarise(across(c(...), ~ mean(.x, na.rm = TRUE)))
}
```

Performance of injected dots and dynamic dots

Take this dynamic dots function:

```
n_args <- function(...) {
  length(list2(...))
}
```

Because it takes dynamic dots you can splice with `!!!` out of the box.

```
n_args(1, 2)
#> [1] 2

n_args(!!!mtcars)
#> [1] 11
```

Equivalently you could enable `!!!` explicitly with `inject()`.

```
inject(n_args(!!!mtcars))
#> [1] 11
```

While the result is the same, what is going on under the hood is completely different. `list2()` is a dots collector that special-cases `!!!` arguments. On the other hand, `inject()` operates on the language and creates a function call containing as many arguments as there are elements in the spliced list. If you supply a list of size 1e6, `inject()` is creating one million arguments before evaluation. This can be much slower.

```
xs <- rep(list(1), 1e6)

system.time(
  n_args(!!!xs)
)
#>    user  system elapsed
#>   0.009   0.000   0.009

system.time(
  inject(n_args(!!!xs))
)
#>    user  system elapsed
#>   0.445   0.012   0.457
```

The same issue occurs when functions taking dynamic dots are called inside a data-masking function like `dplyr::mutate()`. The mechanism that enables `!!!` injection in these arguments is the same as in `inject()`.

See Also

- Injecting with !!, !!!, and glue syntax
- `inject()`
- `exec()`

stack *Get properties of the current or caller frame*

Description

These accessors retrieve properties of frames on the call stack. The prefix indicates for which frame a property should be accessed:

- From the current frame with `current_` accessors.
- From a calling frame with `caller_` accessors.
- From a matching frame with `frame_` accessors.

The suffix indicates which property to retrieve:

- `_fn` accessors return the function running in the frame.
- `_call` accessors return the defused call with which the function running in the frame was invoked.
- `_env` accessors return the execution environment of the function running in the frame.

Usage

```
current_call()

current_fn()

current_env()

caller_call(n = 1)

caller_fn(n = 1)

caller_env(n = 1)

frame_call(frame = caller_env())

frame_fn(frame = caller_env())
```

Arguments

`n`	The number of callers to go back.
`frame`	A frame environment of a currently running function, as returned by `caller_env()`. `NULL` is returned if the environment does not exist on the stack.

See Also

`caller_env()` and `current_env()`

sym — *Create a symbol or list of symbols*

Description

Symbols are a kind of defused expression that represent objects in environments.

- `sym()` and `syms()` take strings as input and turn them into symbols.
- `data_sym()` and `data_syms()` create calls of the form `.data$foo` instead of symbols. Subsetting the `.data` pronoun is more robust when you expect a data-variable. See The data mask ambiguity.

Only tidy eval APIs support the `.data` pronoun. With base R functions, use simple symbols created with `sym()` or `syms()`.

Usage

```
sym(x)

syms(x)

data_sym(x)

data_syms(x)
```

Arguments

`x`	For `sym()` and `data_sym()`, a string. For `syms()` and `data_syms()`, a list of strings.

Value

For `sym()` and `syms()`, a symbol or list of symbols. For `data_sym()` and `data_syms()`, calls of the form `.data$foo`.

See Also

- Defusing R expressions
- Metaprogramming patterns

Examples

```
# Create a symbol
sym("cyl")

# Create a list of symbols
syms(c("cyl", "am"))
```

```
# Symbolised names refer to variables
eval(sym("cyl"), mtcars)

# Beware of scoping issues
Cyl <- "wrong"
eval(sym("Cyl"), mtcars)

# Data symbols are explicitly scoped in the data mask
try(eval_tidy(data_sym("Cyl"), mtcars))

# These can only be used with tidy eval functions
try(eval(data_sym("Cyl"), mtcars))

# The empty string returns the missing argument:
sym("")

# This way sym() and as_string() are inverse of each other:
as_string(missing_arg())
sym(as_string(missing_arg()))
```

`trace_back` *Capture a backtrace*

Description

A backtrace captures the sequence of calls that lead to the current function (sometimes called the call stack). Because of lazy evaluation, the call stack in R is actually a tree, which the `print()` method for this object will reveal.

Users rarely need to call `trace_back()` manually. Instead, signalling an error with `abort()` or setting up `global_entrace()` is the most common way to create backtraces when an error is thrown. Inspect the backtrace created for the most recent error with `last_error()`.

`trace_length()` returns the number of frames in a backtrace.

Usage

```
trace_back(top = NULL, bottom = NULL)

trace_length(trace)
```

Arguments

`top` The first frame environment to be included in the backtrace. This becomes the top of the backtrace tree and represents the oldest call in the backtrace.

This is needed in particular when you call `trace_back()` indirectly or from a larger context, for example in tests or inside an RMarkdown document where you don't want all of the knitr evaluation mechanisms to appear in the backtrace.

	If not supplied, the `rlang_trace_top_env` global option is consulted. This makes it possible to trim the embedding context for all backtraces created while the option is set. If knitr is in progress, the default value for this option is `knitr::knit_global()` so that the knitr context is trimmed out of backtraces.
`bottom`	The last frame environment to be included in the backtrace. This becomes the rightmost leaf of the backtrace tree and represents the youngest call in the backtrace. Set this when you would like to capture a backtrace without the capture context. Can also be an integer that will be passed to `caller_env()`.
`trace`	A backtrace created by `trace_back()`.

Examples

```
# Trim backtraces automatically (this improves the generated
# documentation for the rlang website and the same trick can be
# useful within knitr documents):
options(rlang_trace_top_env = current_env())

f <- function() g()
g <- function() h()
h <- function() trace_back()

# When no lazy evaluation is involved the backtrace is linear
# (i.e. every call has only one child)
f()

# Lazy evaluation introduces a tree like structure
identity(identity(f()))
identity(try(f()))
try(identity(f()))

# When printing, you can request to simplify this tree to only show
# the direct sequence of calls that lead to `trace_back()`
x <- try(identity(f()))
x
print(x, simplify = "branch")

# With a little cunning you can also use it to capture the
# tree from within a base NSE function
x <- NULL
with(mtcars, {x <<- f(); 10})
x


# Restore default top env for next example
options(rlang_trace_top_env = NULL)

# When code is executed indirectly, i.e. via source or within an
# RMarkdown document, you'll tend to get a lot of guff at the beginning
# related to the execution environment:
conn <- textConnection("summary(f())")
```

```
source(conn, echo = TRUE, local = TRUE)
close(conn)

# To automatically strip this off, specify which frame should be
# the top of the backtrace. This will automatically trim off calls
# prior to that frame:
top <- current_env()
h <- function() trace_back(top)

conn <- textConnection("summary(f())")
source(conn, echo = TRUE, local = TRUE)
close(conn)
```

`try_fetch` *Try an expression with condition handlers*

Description

[Experimental]

`try_fetch()` establishes handlers for conditions of a given class (`"error"`, `"warning"`, `"message"`, ...). Handlers are functions that take a condition object as argument and are called when the corresponding condition class has been signalled.

A condition handler can:

- **Recover from conditions** with a value. In this case the computation of `expr` is aborted and the recovery value is returned from `try_fetch()`. Error recovery is useful when you don't want errors to abruptly interrupt your program but resume at the catching site instead.

  ```
  # Recover with the value 0
  try_fetch(1 + "", error = function(cnd) 0)
  ```

- **Rethrow conditions**, e.g. using `abort(msg, parent = cnd)`. See the `parent` argument of `abort()`. This is typically done to add information to low-level errors about the high-level context in which they occurred.

  ```
  try_fetch(1 + "", error = function(cnd) abort("Failed.", parent = cnd))
  ```

- **Inspect conditions**, for instance to log data about warnings or errors. In this case, the handler must return the `zap()` sentinel to instruct `try_fetch()` to ignore (or zap) that particular handler. The next matching handler is called if any, and errors bubble up to the user if no handler remains.

  ```
  log <- NULL
  try_fetch(1 + "", error = function(cnd) {
    log <<- cnd
    zap()
  })
  ```

Whereas `tryCatch()` catches conditions (discarding any running code along the way) and then calls the handler, `try_fetch()` first calls the handler with the condition on top of the currently running code (fetches it where it stands) and then catches the return value. This is a subtle difference that has implications for the debuggability of your functions. See the comparison with `tryCatch()` section below.

Another difference between `try_fetch()` and the base equivalent is that errors are matched across chains, see the `parent` argument of `abort()`. This is a useful property that makes `try_fetch()` insensitive to changes of implementation or context of evaluation that cause a classed error to suddenly get chained to a contextual error. Note that some chained conditions are not inherited, see the `.inherit` argument of `abort()` or `warn()`. In particular, downgraded conditions (e.g. from error to warning or from warning to message) are not matched across parents.

Usage

```
try_fetch(expr, ...)
```

Arguments

`expr`	An R expression.
`...`	<`dynamic-dots`> Named condition handlers. The names specify the condition class for which a handler will be called.

Stack overflows

A stack overflow occurs when a program keeps adding to itself until the stack memory (whose size is very limited unlike heap memory) is exhausted.

```
# A function that calls itself indefinitely causes stack overflows
f <- function() f()
f()
#> Error: C stack usage  9525680 is too close to the limit
```

Because memory is very limited when these errors happen, it is not possible to call the handlers on the existing program stack. Instead, error conditions are first caught by `try_fetch()` and only then error handlers are called. Catching the error interrupts the program up to the `try_fetch()` context, which allows R to reclaim stack memory.

The practical implication is that error handlers should never assume that the whole call stack is preserved. For instance a `trace_back()` capture might miss frames.

Note that error handlers are only run for stack overflows on R >= 4.2. On older versions of R the handlers are simply not run. This is because these errors do not inherit from the class `stackOverflowError` before R 4.2. Consider using `tryCatch()` instead with critical error handlers that need to capture all errors on old versions of R.

Comparison with `tryCatch()`

`try_fetch()` generalises `tryCatch()` and `withCallingHandlers()` in a single function. It reproduces the behaviour of both calling and exiting handlers depending on the return value of the handler. If the handler returns the `zap()` sentinel, it is taken as a calling handler that declines to recover from a condition. Otherwise, it is taken as an exiting handler which returns a value from the catching site.

The important difference between `tryCatch()` and `try_fetch()` is that the program in `expr` is still fully running when an error handler is called. Because the call stack is preserved, this makes it possible to capture a full backtrace from within the handler, e.g. when rethrowing the error with `abort(parent = cnd)`. Technically, `try_fetch()` is more similar to (and implemented on top of) `base::withCallingHandlers()` than `tryCatch()`.

`type-predicates` *Type predicates*

Description

These type predicates aim to make type testing in R more consistent. They are wrappers around `base::typeof()`, so operate at a level beneath S3/S4 etc.

Usage

```
is_list(x, n = NULL)

is_atomic(x, n = NULL)

is_vector(x, n = NULL)

is_integer(x, n = NULL)

is_double(x, n = NULL, finite = NULL)

is_complex(x, n = NULL, finite = NULL)

is_character(x, n = NULL)

is_logical(x, n = NULL)

is_raw(x, n = NULL)

is_bytes(x, n = NULL)

is_null(x)
```

Arguments

`x`	Object to be tested.
`n`	Expected length of a vector.
`finite`	Whether all values of the vector are finite. The non-finite values are `NA`, `Inf`, `-Inf` and `NaN`. Setting this to something other than `NULL` can be expensive because the whole vector needs to be traversed and checked.

Details

Compared to base R functions:

- The predicates for vectors include the `n` argument for pattern-matching on the vector length.
- Unlike `is.atomic()` in R < 4.4.0, `is_atomic()` does not return `TRUE` for `NULL`. Starting in R 4.4.0 `is.atomic(NULL)` returns FALSE.
- Unlike `is.vector()`, `is_vector()` tests if an object is an atomic vector or a list. `is.vector` checks for the presence of attributes (other than name).

See Also

bare-type-predicates scalar-type-predicates

---

vector-construction *Create vectors*

Description

[Questioning]

The atomic vector constructors are equivalent to `c()` but:

- They allow you to be more explicit about the output type. Implicit coercions (e.g. from integer to logical) follow the rules described in vector-coercion.
- They use dynamic dots.

Usage

```
lgl(...)

int(...)

dbl(...)

cpl(...)

chr(...)

bytes(...)
```

Arguments

`...`	Components of the new vector. Bare lists and explicitly spliced lists are spliced.

Life cycle

- All the abbreviated constructors such as `lgl()` will probably be moved to the vctrs package at some point. This is why they are marked as questioning.
- Automatic splicing is soft-deprecated and will trigger a warning in a future version. Please splice explicitly with `!!!`.

Examples

```
# These constructors are like a typed version of c():
c(TRUE, FALSE)
lgl(TRUE, FALSE)

# They follow a restricted set of coercion rules:
int(TRUE, FALSE, 20)

# Lists can be spliced:
dbl(10, !!! list(1, 2L), TRUE)


# They splice names a bit differently than c(). The latter
# automatically composes inner and outer names:
c(a = c(A = 10), b = c(B = 20, C = 30))

# On the other hand, rlang's constructors use the inner names and issue a
# warning to inform the user that the outer names are ignored:
dbl(a = c(A = 10), b = c(B = 20, C = 30))
dbl(a = c(1, 2))

# As an exception, it is allowed to provide an outer name when the
# inner vector is an unnamed scalar atomic:
dbl(a = 1)

# Spliced lists behave the same way:
dbl(!!! list(a = 1))
dbl(!!! list(a = c(A = 1)))

# bytes() accepts integerish inputs
bytes(1:10)
bytes(0x01, 0xff, c(0x03, 0x05), list(10, 20, 30L))
```

---

wref_key *Get key/value from a weak reference object*

Description

Get key/value from a weak reference object

Usage

```
wref_key(x)

wref_value(x)
```

Arguments

x	A weak reference object.

See Also

is_weakref() and new_weakref().

---

zap *Create zap objects*

Description

zap() creates a sentinel object that indicates that an object should be removed. For instance, named zaps instruct env_bind() and call_modify() to remove those objects from the environment or the call.

The advantage of zap objects is that they unambiguously signal the intent of removing an object. Sentinels like NULL or missing_arg() are ambiguous because they represent valid R objects.

Usage

```
zap()

is_zap(x)
```

Arguments

x	An object to test.

Examples

```
# Create one zap object:
zap()

# Create a list of zaps:
rep(list(zap()), 3)
rep_named(c("foo", "bar"), list(zap()))
```

zap_srcref *Zap source references*

Description

There are a number of situations where R creates source references:

- Reading R code from a file with `source()` and `parse()` might save source references inside calls to `function` and `{`.
- `sys.call()` includes a source reference if possible.
- Creating a closure stores the source reference from the call to `function`, if any.

These source references take up space and might cause a number of issues. `zap_srcref()` recursively walks through expressions and functions to remove all source references.

Usage

```
zap_srcref(x)
```

Arguments

`x`	An R object. Functions and calls are walked recursively.

Index