

Package ‘rlang’

October 23, 2019

Version 0.4.1

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 GPL-3

LazyData true

ByteCompile true

Biarch true

Depends R (>= 3.2.0)

Suggests cli,
covr,
crayon,
magrittr,
methods,
pillar,
rmarkdown,
testthat (>= 2.0.0)

Encoding UTF-8

RoxygenNote 6.1.1

Roxygen list(markdown = TRUE)

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

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

R topics documented:

abort	3
arg_match	6
as_box	7
as_data_mask	7
as_environment	11
as_function	12
as_label	13
as_name	14
as_quosure	15

as_string	16
bare-type-predicates	17
box	18
call2	19
caller_env	20
call_args	21
call_fn	22
call_inspect	23
call_modify	23
call_name	25
call_standardise	26
catch_cnd	27
cnd_signal	27
done	28
empty_env	29
entrace	29
env	30
env_bind	33
env_clone	36
env_depth	36
env_get	37
env_has	38
env_inherits	38
env_name	39
env_names	39
env_parent	40
env_print	42
env_unbind	42
eval_bare	43
eval_tidy	45
exec	47
exprs_auto_name	48
expr_interp	49
expr_label	50
expr_print	51
fn_body	52
fn_env	52
fn_fmils	53
f_rhs	54
f_text	55
get_env	56
has_name	57
inherits_any	58
is_call	59
is_empty	60
is_environment	61
is_expression	61
is_formula	63
is_function	64
is_installed	66
is_integerish	66
is_interactive	67

is_named	68
is_namespace	69
is_symbol	69
is_true	70
is_weakref	70
last_error	71
missing_arg	71
names2	73
new_formula	74
new_function	74
new_quosures	75
new_weakref	76
op-get-attr	77
op-na-default	78
op-null-default	78
pairlist2	79
parse_expr	79
quasiquotation	81
quosure	84
quotation	86
quo_label	90
quo_squash	92
rep_along	92
rlang_backtrace_on_error	93
scalar-type-predicates	94
scoped_bindings	95
scoped_options	96
seq2	97
set_expr	98
set_names	99
sym	100
tidy-dots	100
tidyeval-data	102
trace_back	103
type-predicates	104
vector-construction	106
with_abort	107
with_handlers	108
wref_key	109
zap	110

Index**111**

 abort

Signal an error, warning, or message

Description

These functions are equivalent to base functions `base::stop()`, `base::warning()` and `base::message()`, but make it easy to supply condition metadata:

- Supply `.subclass` to create a classed condition. Typed conditions can be captured or handled selectively, allowing for finer-grained error handling.
- Supply metadata with named `...` arguments. This data will be stored in the condition object and can be examined by handlers.

`interrupt()` allows R code to simulate a user interrupt of the kind that is signalled with Ctrl-C. It is currently not possible to create custom interrupt condition objects.

Usage

```
abort(message = "", .subclass = NULL, ..., trace = NULL,
       call = NULL, parent = NULL, msg, type)
```

```
warn(message, .subclass = NULL, ..., call = NULL, msg, type)
```

```
inform(message, .subclass = NULL, ..., call = NULL, msg, type)
```

```
signal(message, .subclass, ...)
```

```
interrupt()
```

Arguments

<code>message</code>	The message to display. Experimental: Can also be a named character vector, in which case the message is assembled as a list of bullets. See <code>cond_message()</code> to learn how names control the bulleted output.
<code>.subclass</code>	Subclass of the condition. This allows your users to selectively handle the conditions signalled by your functions.
<code>...</code>	Additional data to be stored in the condition object.
<code>trace</code>	A trace object created by <code>trace_back()</code> .
<code>call</code>	Defunct as of rlang 0.4.0. Storing the full backtrace is now preferred to storing a simple call.
<code>parent</code>	A parent condition object created by <code>abort()</code> .
<code>msg, type</code>	These arguments were renamed to <code>message</code> and <code>.subclass</code> and are defunct as of rlang 0.4.0.

Backtrace

Unlike `stop()` and `warning()`, these functions don't include call information by default. This saves you from typing `call = FALSE` and produces cleaner error messages.

A backtrace is always saved into error objects. You can print a simplified backtrace of the last error by calling `last_error()` and a full backtrace with `summary(last_error())`.

You can also display a backtrace with the error message by setting the option `rlang_backtrace_on_error`. It supports the following values:

- "reminder": Invite users to call `rlang::last_error()` to see a backtrace.

- "branch": Display a simplified backtrace.
- "collapse": Display a collapsed backtrace tree.
- "full": Display a full backtrace tree.
- "none": Display nothing.

Mufflable conditions

Signalling a condition with `inform()` or `warn()` causes a message to be displayed in the console. These messages can be muffled with `base::suppressMessages()` or `base::suppressWarnings()`.

On recent R versions (\geq R 3.5.0), interrupts are typically signalled with a "resume" restart. This is however not guaranteed.

Lifecycle

These functions were changed in `rlang` 0.3.0 to take condition metadata with `...`. Consequently:

- All arguments were renamed to be prefixed with a dot, except for `type` which was renamed to `.subclass`.
- `.call` (previously `call`) can no longer be passed positionally.

See Also

[with_abort\(\)](#) to convert all errors to `rlang` errors.

Examples

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

# Signal an error with a message just like stop():
abort("Something bad happened")

# Give a class to the error:
abort("Something bad happened", "somepkg_bad_error")

# This will allow your users to handle the error selectively
tryCatch(
  somepkg_function(),
  somepkg_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("Something bad happened", "somepkg_bad_error", data = 1:10)

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

```

# If you call low-level APIs it is good practice to catch technical
# errors and rethrow them with a more meaningful message. Pass on
# the caught error as `parent` to get a nice decomposition of
# errors and backtraces:
file <- "http://foo.bar/baz"
tryCatch(
  download(file),
  error = function(err) {
    msg <- sprintf("Can't download `%s`", file)
    abort(msg, parent = err)
  })

# Unhandled errors are saved automatically by `abort()` and can be
# retrieved with `last_error()`. The error prints with a simplified
# backtrace:
abort("Saved error?")
last_error()

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

}

```

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.

Usage

```
arg_match(arg, values = NULL)
```

Arguments

arg	A symbol referring to an argument accepting strings.
values	The possible values that arg can take. If NULL, the values are taken from the function definition of the caller frame .

Value

The string supplied to arg.

Examples

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

# This would throw an informative error if run:
# fn("b")
# fn("baz")
```

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

<code>x</code>	An R object.
<code>class</code> , <code>.class</code>	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, <code>class</code> is passed to new_box() .
<code>.x</code>	An R object.
<code>.p</code>	A predicate function.
<code>...</code>	Arguments passed to <code>.p</code> .

as_data_mask

*Create a data mask***Description****Stable**

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, parent = NULL)
```

```
as_data_pronoun(data)
```

```
new_data_mask(bottom, top = bottom, parent = NULL)
```

Arguments

data	A data frame or named vector of masking data.
parent	Deprecated. This argument no longer has any effect. The parent of the data mask is determined from either: <ul style="list-style-type: none"> • The env argument of <code>eval_tidy()</code> • Quosure environments when applicable
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.

Life cycle

The `parent` argument no longer has any effect and is defunct as of rlang 0.4.0. The parent of the data mask is determined from either:

- The `env` argument of `eval_tidy()`
- Quosure environments when applicable

Passing environments to `as_data_mask()` is deprecated as of rlang 0.3.0. Please use `new_data_mask()` instead.

rlang 0.2.0

In early versions of rlang data masks were called overscopes. We think data mask is a more natural name in R. It makes reference to masking in the search path which occurs through the same mechanism (in technical terms, lexical scoping with hierarchically nested environments). We say that objects from user data mask objects in the current environment.

Following this change in terminology, `as_overscope()` and `new_overscope()` were deprecated in rlang 0.2.0 in favour of `as_data_mask()` and `new_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 the values with the pronouns:
eval_tidy(quote(c(.data$a, .data$b, .data$c)), data = mask)
```

as_environment	<i>Coerce to an environment</i>
----------------	---------------------------------

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

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

Life cycle

`as_env()` was soft-deprecated and renamed to `as_environment()` in rlang 0.2.0. This is for consistency as type predicates should not be abbreviated.

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	<i>Convert to function or closure</i>
-------------	---------------------------------------

Description

Stable

- `as_function()` transforms a one-sided formula into a function. This powers the lambda syntax in packages like `purrr`.
- `as_closure()` first passes its argument to `as_function()`. If the result is a primitive function, it regularises it to a proper [closure](#) (see `is_function()` about primitive functions). Some special control flow primitives like `if`, `for`, or `break` can't be coerced to a closure.

Usage

```
as_function(x, env = caller_env())
```

```
is_lambda(x)
```

```
as_closure(x, env = caller_env())
```

Arguments

x	<p>A function or formula.</p> <p>If a function, it is used as is.</p> <p>If a formula, e.g. <code>~ .x + 2</code>, it is converted to a function with up to two arguments: <code>.x</code> (single argument) or <code>.x</code> and <code>.y</code> (two arguments). The <code>.</code> placeholder can be used instead of <code>.x</code>. This allows you to create very compact anonymous functions (lambdas) with up to two inputs. Functions created from formulas have a special class. Use <code>is_lambda()</code> to test for it.</p> <p>Lambdas currently do not support quasiquote, due to the way the arguments are handled internally.</p>
env	Environment in which to fetch the function in case x is a string.

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"))

# Primitive functions are regularised as closures
as_closure(list)
as_closure("list")
```

```

# Operators have `.x` and `.y` as arguments, just like lambda
# functions created with the formula syntax:
as_closure(`+`)
as_closure(`~`)

# Use a regular function for tidy evaluation, also when calling functions
# that use tidy evaluation:
## Bad:
e <- as_function(~ as_label(ensym(.x)))
## Good:
e <- as_function(function(x) as_label(ensym(x)))

e(y)

```

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_string() 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_string\(\)](#). For instance, use as_label() with objects captured with enquos() and as_string() 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>.

Note that simple symbols should generally be transformed to strings with [as_name\(\)](#). Labelling is not a well defined operation and no assumption should be made about how the label is created. On the other hand, as_name() only works with symbols and is a well defined, deterministic operation.

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(fooobar))

# 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 gets 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.

Expect `as_name()` to gain [name-repairing](#) features in the future.

Note that `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).

Usage

```
as_name(x)
```

Arguments

`x` A string or symbol, possibly wrapped in a [quosure](#). If a string, the attributes are removed, if any.

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_quosure

Coerce object to quosure

Description

While `new_quosure()` wraps any R object (including expressions, formulas, or other quosures) into a quosure, `as_quosure()` converts formulas and quosures and does not double-wrap.

Usage

```
as_quosure(x, env = NULL)

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

Arguments

x	An object to convert. Either an expression or a formula.
env	The environment in which the expression should be evaluated. Only used for symbols and calls. This should typically be the environment in which the expression was created.
expr	The expression wrapped by the quosure.

Life cycle

- `as_quosure()` now requires an explicit default environment for creating quosures from symbols and calls.
- `as_quosureish()` is deprecated as of rlang 0.2.0. This function assumes that quosures are formulas which is currently true but might not be in the future.

See Also

[quo\(\)](#), [is_quosure\(\)](#)

Examples

```
# as_quosure() converts expressions or any R object to a validly
# scoped quosure:
env <- env(var = "thing")
as_quosure(quote(var), env)

# The environment is ignored for formulas:
as_quosure(~foo, env)
as_quosure(~foo)

# However you must supply it for symbols and calls:
try(as_quosure(quote(var)))
```

`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_integer(x, n = NULL)

is_bare_numeric(x, n = NULL)

is_bare_character(x, n = NULL, encoding = 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.
encoding	Defunct as of rlang 0.4.0.

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()`, `is_bare_atomic()` does not return TRUE for NULL.
- 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

<code>class</code>	For <code>new_box()</code> , an additional class for the boxed value (in addition to <code>rLang_box</code>). For <code>is_box()</code> , a class or vector of classes passed to <code>inherits_all()</code> .
<code>...</code>	Additional attributes passed to <code>base::structure()</code> .
<code>x</code> , <code>.x</code>	An R object.
<code>box</code>	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")
```

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

<code>.fn</code>	Function to call. Must be a callable object: a string, symbol, call, or a function.
<code>...</code>	Arguments to the call either in or out of a list. These dots support tidy dots features. Empty arguments are preserved.
<code>.ns</code>	Namespace with which to prefix <code>.fn</code> . Must be a string or symbol.

Difference with base constructors

[call2\(\)](#) is more flexible and convenient 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")
```

```
ns_call <- as.call(list(as.name(":"), as.name("list"), as.name("base")))
as.call(list(ns_call, 1, 2))
```

- `call2()` has [tidy dots](#) support and you can splice lists of arguments with `!!!`. With base R, you need to use `as.call()` instead of `call()` if the arguments are in a list.

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

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

as.call(c(list(as.name("mean")), 1:10), args))
```

Life cycle

In `rlang` 0.2.0 `lang()` was soft-deprecated and renamed to `call2()`.

In early versions of `rlang` calls were called "language" objects in order to follow the R type nomenclature as returned by `base::typeof()`. The goal was to avoid adding to the confusion between S modes and R types. With hindsight we find it is better to use more meaningful type names.

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_env

Get the current or caller environment

Description

- The current environment is the execution environment of the current function (the one currently being evaluated).
- The caller environment is the execution environment of the function that called the current function.

Usage

```
caller_env(n = 1)

current_env()
```

Arguments

`n` Number of frames to go back.

See Also

[caller_frame\(\)](#) and [current_frame\(\)](#)

Examples

```
# Let's create a function that returns its current environment and
# its caller environment:
fn <- function() list(current = current_env(), caller = caller_env())

# The current environment is an unique execution environment
# created when `fn()` was called. The caller environment is the
# global env because that's where we called `fn()`.
fn()

# Let's call `fn()` again but this time within a function:
g <- function() fn()

# Now the caller environment is also an unique execution environment.
# This is the exec env created by R for our call to g():
g()
```

call_args

Extract arguments from a call

Description

Extract arguments from a call

Usage

```
call_args(call)
```

```
call_args_names(call)
```

Arguments

`call` Can be a call or a quosure that wraps a call.

Value

A named list of arguments.

Life cycle

In rlang 0.2.0, `lang_args()` and `lang_args_names()` were deprecated and renamed to `call_args()` and `call_args_names()`. See lifecycle section in [call2\(\)](#) for more about this change.

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_fn

Extract function from a call

Description

If a frame or formula, the function will be retrieved from the associated environment. Otherwise, it is looked up in the calling frame.

Usage

```
call_fn(call, env = caller_env())
```

Arguments

call	Can be a call or a quosure that wraps a call.
env	The environment where to find the definition of the function quoted in call in case call is not wrapped in a quosure.

Life cycle

In rlang 0.2.0, lang_fn() was deprecated and renamed to call_fn(). See lifecycle section in [call2\(\)](#) for more about this change.

See Also

[call_name\(\)](#)

Examples

```
# Extract from a quoted call:
call_fn(quote(matrix()))
call_fn(quo(matrix()))

# Extract the calling function
test <- function() call_fn(call_frame())
test()
```

call_inspect	<i>Inspect a call</i>
--------------	-----------------------

Description

This function is useful for quick testing and debugging when you manipulate expressions and calls. It lets you check that a function is called with the right arguments. This can be useful in unit tests for instance. Note that this is just a simple wrapper around `base::match.call()`.

Usage

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

Arguments

... Arguments to display in the returned call.

Examples

```
call_inspect(foo(bar), "" %>% identity())
```

call_modify	<i>Modify the arguments of a call</i>
-------------	---------------------------------------

Description

If you are working with a user-supplied call, make sure the arguments are standardised with `call_standardise()` 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.
...	Named or unnamed expressions (constants, names or calls) used to modify the call. Use <code>zap()</code> to remove arguments. These dots support <code>tidy dots</code> features. Empty arguments are allowed and 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	Soft-deprecated as of rlang 0.3.0. Please call <code>call_standardise()</code> manually.

Value

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

Life cycle

- The `.standardise` argument is deprecated as of rlang 0.3.0.
- In rlang 0.2.0, `lang_modify()` was deprecated and renamed to `call_modify()`. See lifecycle section in [call2\(\)](#) for more about this change.

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 = NULL, 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 because it might contain unmatched arguments:
user_call <- quote(matrix(x, nc = 3))
call_modify(user_call, ncol = 1)

# Standardising applies the usual argument matching rules:
user_call <- call_standardise(user_call)
user_call
call_modify(user_call, ncol = 1)
```



```
# You can also modify quosures inplace:
f <- quo(matrix(bar))
call_modify(f, quote(foo))

# 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 = NULL, bar = missing_arg())

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

call_name	<i>Extract function name or namespaced of a call</i>
-----------	--

Description

Extract function name or namespaced of a call

Usage

```
call_name(call)
```

```
call_ns(call)
```

Arguments

call Can be a call or a quosure that wraps a call.

Value

A string with the function name, or NULL if the function is anonymous.

Life cycle

In rlang 0.2.0, lang_name() was deprecated and renamed to call_name(). See lifecycle section in [call2\(\)](#) for more about this change.

See Also

[call_fn\(\)](#)

Examples

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

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

# Anonymous and subsetting 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()))
```

call_standardise	<i>Standardise a call</i>
------------------	---------------------------

Description

This is essentially equivalent to `base::match.call()`, but with experimental handling of primitive functions.

Usage

```
call_standardise(call, env = caller_env())
```

Arguments

call	Can be a call or a quosure that wraps a call.
env	The environment where to find the definition of the function quoted in call in case call is not wrapped in a quosure.

Value

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

Life cycle

In rlang 0.2.0, `lang_standardise()` was deprecated and renamed to `call_standardise()`. See lifecycle section in [call2\(\)](#) for more about this change.

catch_cnd	<i>Catch a condition</i>
-----------	--------------------------

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

<code>expr</code>	Expression to be evaluated with a catching condition handler.
<code>classes</code>	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(cnd_signal("my_condition", .msg = "a condition"))
```

cnd_signal	<i>Signal a condition object</i>
------------	----------------------------------

Description

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()`.

Use `cnd_type()` to determine the type of a condition.

Usage

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

Arguments

`cnd` A condition object (see `cnd()`).
`.cnd`, `.muffleable`
 These arguments are deprecated.

Lifecycle

- `.cnd` has been renamed to `cnd` and is deprecated as of rlang 0.3.0.
- The `.muffleable` argument is deprecated as of rlang 0.3.0 and no longer has any effect. Non-critical conditions are always signalled with a muffle restart.
- Creating a condition object with `cnd_signal()` is deprecated as of rlang 0.3.0. Please use `signal()` instead.

See Also

`abort()`, `warn()` and `inform()` for creating and signalling structured R conditions. See `with_handlers()` for establishing condition handlers.

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)
```

empty_env	<i>Get the empty environment</i>
-----------	----------------------------------

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())
```

entrace	<i>Add backtrace from error handler</i>
---------	---

Description

`entrace()` interrupts an error throw to add an [rlang backtrace](#) to the error. The error throw is immediately resumed. `cond_entrace()` adds a backtrace to a condition object, without any other effect. Both functions should be called directly from an error handler.

Set the error global option to `quote(rlang::entrace())` to transform base errors to rlang errors. These enriched errors include a backtrace. The RProfile is a good place to set the handler. See [rlang_backtrace_on_error](#) for details.

`entrace()` also works as a [calling](#) handler, though it is often more practical to use the higher-level function `with_abort()`.

Usage

```
entrace(cond, ..., top = NULL, bottom = NULL)
```

```
cond_entrace(cond, ..., top = NULL, bottom = NULL)
```

Arguments

<code>cond</code>	When <code>entrace()</code> is used as a calling handler, <code>cond</code> is the condition to handle.
<code>...</code>	Unused. These dots are for future extensions.
<code>top</code>	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 <code>trace_back()</code> 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.
<code>bottom</code>	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 <code>caller_env()</code> .

See Also

`with_abort()` to promote conditions to rlang errors. `cond_entrace()` to manually add a backtrace to a condition.

Examples

```
if (FALSE) { # Not run

# Set the error handler in your RProfile like this:
if (requireNamespace("rlang", quietly = TRUE)) {
  options(error = rlang::entrace)
}

}
```

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(...)
```

```
child_env(.parent, ...)
```

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

Arguments

- `...`, `data` Named values. You can supply one unnamed to specify a custom parent, otherwise it defaults to the current environment. These dots support [tidy dots](#) features.
- `.parent`, `parent` A parent environment. Can be an object supported by [as_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.

Life cycle

- `child_env()` is in the questioning stage. It is redundant now that `env()` accepts parent environments.

See Also

[env_has\(\)](#), [env_bind\(\)](#).

Examples

```
# env() creates a new environment which has the current environment
# as parent
env <- env(a = 1, b = "foo")
env$b
identical(env_parent(env), current_env())

# Supply one unnamed argument to override the default:
```

```

env <- env(base_env(), a = 1, b = "foo")
identical(env_parent(env), base_env())

# child_env() lets you specify a parent:
child <- child_env(env, c = "bar")
identical(env_parent(child), env)

# This child environment owns `c` but inherits `a` and `b` from `env`:
env_has(child, c("a", "b", "c", "d"))
env_has(child, c("a", "b", "c", "d"), inherit = TRUE)

# `parent` is passed to as_environment() to provide handy
# shortcuts. Pass a string to create a child of a package
# environment:
child_env("rlang")
env_parent(child_env("rlang"))

# Or `NULL` to create a child of the empty environment:
child_env(NULL)
env_parent(child_env(NULL))

# The base package environment is often a good default choice for a
# parent environment because it contains all standard base
# functions. Also note that it will never inherit from other loaded
# package environments since R keeps the base package at the tail
# of the search path:
base_child <- child_env("base")
env_has(base_child, c("lapply", "("), inherit = TRUE)

# On the other hand, a child of the empty environment doesn't even
# see a definition for `(`
empty_child <- child_env(NULL)
env_has(empty_child, c("lapply", "("), inherit = TRUE)

# Note that all other package environments inherit from base_env()
# as well:
rlang_child <- child_env("rlang")
env_has(rlang_child, "env", inherit = TRUE) # rlang function
env_has(rlang_child, "lapply", inherit = TRUE) # base function

# 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).

Usage

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

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

```
env_bind_active(.env, ...)
```

Arguments

<code>.env</code>	An environment.
<code>...</code>	Pairs of names and expressions, values or functions. Pass <code>zap()</code> objects to remove bindings. These dots support tidy dots features.
<code>.eval_env</code>	The environment where the expressions will be evaluated when the symbols are forced.

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.

Life cycle

Passing an environment wrapper like a formula or a function instead of an environment is soft-deprecated as of rlang 0.3.0. This internal genericity was causing confusion (see issue #427). You should now extract the environment separately before calling these functions.

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 <- "fieval"
  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_clone *Clone an environment*

Description

This creates a new environment containing exactly the same objects, optionally with a new parent.

Usage

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

Arguments

env	An environment.
parent	The parent of the cloned environment.

Examples

```
env <- env(!!! mtcars)
clone <- env_clone(env)
identical(env, clone)
identical(env$cyl, clone$cyl)
```

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 environment 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)
```

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

Arguments

env	An environment.
nm, nms	Names of bindings. nm must be a single 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.

Value

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

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	<i>Does an environment have or see bindings?</i>
---------	--

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 containing the names of the bindings to remove.
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	<i>Does environment inherit from another environment?</i>
--------------	---

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_name	<i>Label of an environment</i>
----------	--------------------------------

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	<i>Names and numbers of symbols bound in an environment</i>
-----------	---

Description

`env_names()` returns object names from an environment 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	<i>Get parent environments</i>
------------	--------------------------------

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(), sentinel = NULL)

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.
sentinel	This argument is defunct, please use last instead.

Value

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

Life cycle

The sentinel argument of env_tail() has been deprecated in rlang 0.2.0 and renamed to last. It is defunct as of rlang 0.4.0.

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_print	<i>Pretty-print an environment</i>
-----------	------------------------------------

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 succinctly 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

Usage

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

Arguments

env An environment, or object that can be converted to an environment by [get_env\(\)](#).

env_unbind	<i>Remove bindings from an environment</i>
------------	--

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 containing the names of the bindings 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

```

data <- set_names(as.list(letters), letters)
env_bind(environment(), !!! data)
env_has(environment(), letters)

# env_unbind() removes bindings:
env_unbind(environment(), letters)
env_has(environment(), letters)

# With inherit = TRUE, it removes bindings in parent environments
# as well:
parent <- child_env(NULL, foo = "a")
env <- child_env(parent, foo = "b")
env_unbind(env, "foo", inherit = TRUE)
env_has(env, "foo", inherit = TRUE)

```

eval_bare

Evaluate an expression in an environment

Description**Stable**

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 environment 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 buried 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

Stable

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 lets you be explicit about where to find values and throw errors if you try to access non-existent values.

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 <code>as_data_mask()</code> or <code>new_data_mask()</code> . 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.

Data masking

Data masking refers to how columns or objects inside data have priority over objects defined in env (or in the quosure environment, if applicable). If there is a column `var` in data and an object `var` in env, and `expr` refers to `var`, the column has priority:

```
var <- "this one?"
data <- data.frame(var = rep("Or that one?", 3))

within <- function(data, expr) {
  eval_tidy(enquo(expr), data)
}

within(data, toupper(var))
#> [1] "OR THAT ONE?" "OR THAT ONE?" "OR THAT ONE?"
```

Because the columns or objects in data are always found first, before objects from env, we say that the data "masks" the environment.

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 quasiquotation support, it is recommended to use `eval_tidy()` because users will likely unquote 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. 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)))
```

Life cycle

rlang 0.3.0

Passing an environment to `data` is deprecated. Please construct an `rlang` data mask with `new_data_mask()`.

See Also

[quasiquotation](#) for the second leg of the tidy evaluation framework.

Examples

```
# With simple quoted expressions eval_tidy() works the same way as
# eval():
apple <- "apple"
kiwi <- "kiwi"
expr <- quote(paste(apple, kiwi))
expr

eval(expr)
eval_tidy(expr)

# Both accept a data mask as argument:
data <- list(apple = "CARROT", kiwi = "TOMATO")
eval(expr, data)
eval_tidy(expr, data)

# In addition eval_tidy() has support for quosures:
with_data <- function(data, expr) {
  quo <- enquo(expr)
```

```

    eval_tidy(quo, data)
  }
  with_data(NULL, apple)
  with_data(data, apple)
  with_data(data, list(apple, kiwi))

# Secondly eval_tidy() installs handy pronouns that allow users to
# be explicit about where to find symbols:
with_data(data, .data$apple)
with_data(data, .env$apple)

# Note that instead of using `.env` it is often equivalent and may
# be preferred to unquote a value. There are two differences. First
# unquoting happens earlier, when the quosure is created. Secondly,
# subsetting `.env` with the `$` operator may be brittle because
# `$` does not look through the parents of the environment.
#
# For instance using `.env$name` in a magrittr pipeline is an
# instance where this poses problem, because the magrittr pipe
# currently (as of v1.5.0) evaluates its operands in a *child* of
# the current environment (this child environment is where it
# defines the pronoun `.`).
## Not run:
data %>% with_data(!kiwi)      # "kiwi"
data %>% with_data(.env$kiwi)  # NULL

## End(Not run)

```

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 [tidy-dots](#))
- To call every function stored in a list (in conjunction with `map()`/`lapply()`)

Usage

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

Arguments

<code>.fn</code>	A function, or function name as a string.
<code>...</code>	Arguments to function. These dots support tidy-dots features.
<code>.env</code>	Environment in which to evaluate the call. This will be most useful if <code>f</code> 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)

```

```
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). `exprs_auto_name()` deparses the expressions with `expr_name()` by default. `quos_auto_name()` deparses with `quo_name()`.

Usage

```
exprs_auto_name(exprs, width = NULL, printer = NULL)
```

```
quos_auto_name(quos, width = NULL)
```

Arguments

<code>exprs</code>	A list of expressions.
<code>width</code>	Deprecated. Maximum width of names.
<code>printer</code>	Deprecated. A function that takes an expression and converts it to a string. This function must take an expression as the first argument and width as the second argument.
<code>quos</code>	A list of quosures.

`expr_interp`*Process unquote operators in a captured expression*

Description

While all capturing functions in the tidy evaluation framework perform unquote on capture (most notably `quo()`), `expr_interp()` manually processes unquoting operators in expressions that are already captured. `expr_interp()` should be called in all user-facing functions expecting a formula as argument to provide the same quasiquotation functionality as NSE functions.

Usage

```
expr_interp(x, env = NULL)
```

Arguments

<code>x</code>	A function, raw expression, or formula to interpolate.
<code>env</code>	The environment in which unquoted expressions should be evaluated. By default, the formula or closure environment if a formula or a function, or the current environment otherwise.

Examples

```
# All tidy NSE functions like quo() unquote on capture:
quo(list(!(1 + 2)))

# expr_interp() is meant to provide the same functionality when you
# have a formula or expression that might contain unquoting
# operators:
f <- ~list(!(1 + 2))
expr_interp(f)

# Note that only the outer formula is unquoted (which is a reason
# to use expr_interp() as early as possible in all user-facing
# functions):
f <- ~list(~!(1 + 2), !(1 + 2))
expr_interp(f)

# Another purpose for expr_interp() is to interpolate a closure's
# body. This is useful to inline a function within another. The
# important limitation is that all formal arguments of the inlined
# function should be defined in the receiving function:
other_fn <- function(x) toupper(x)

fn <- expr_interp(function(x) {
  x <- paste0(x, "_suffix")
  !!! body(other_fn)
})
fn
fn("foo")
```

expr_label	<i>Turn an expression to a label</i>
------------	--------------------------------------

Description

Questioning

expr_text() turns the expression into a single string, which might be multi-line. expr_name() is suitable for formatting names. It works best with symbols and scalar types, but also accepts calls. expr_label() formats the expression nicely for use in messages.

Usage

```
expr_label(expr)
```

```
expr_name(expr)
```

```
expr_text(expr, width = 60L, nlines = Inf)
```

Arguments

expr	An expression to labellise.
width	Width of each line.
nlines	Maximum number of lines to extract.

Life cycle

These functions are in the questioning stage because they are redundant with the quo_ variants and do not handle quosures.

Examples

```
# To labellise a function argument, first capture it with
# substitute():
fn <- function(x) expr_label(substitute(x))
fn(x:y)

# Strings are encoded
expr_label("a\nb")

# Names and expressions are quoted with ``
expr_label(quote(x))
expr_label(quote(a + b + c))

# Long expressions are collapsed
expr_label(quote(foo({
  1 + 2
  print(x)
})))
```

expr_print	<i>Print an expression</i>
------------	----------------------------

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, width = peek_option("width"))
```

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

Arguments

<code>x</code>	An object or expression to print.
<code>width</code>	The width of the deparsed or printed expression. Defaults to the global option <code>width</code> .

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)
```

fn_body	<i>Get or set function body</i>
---------	---------------------------------

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	<i>Return the closure environment of a function</i>
--------	---

Description

Closure environments define the scope of functions (see `env()`). When a function call is evaluated, R creates an evaluation frame (see `ctxt_stack()`) 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_fm1s()
fn()

# fn_fm1s_syms() makes it easy to forward arguments:
call2("apply", !!! fn_fm1s_syms(lapply))

# You can also change the formals:
fn_fm1s(fn) <- list(A = 10, B = 20)
fn()

fn_fm1s_names(fn) <- c("foo", "bar")
fn()
```

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 (see `ctxt_stack()`) 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

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

Life cycle

- Using `get_env()` without supplying `env` is deprecated as of rlang 0.3.0. Please use `current_env()` to retrieve the current environment.
- Passing environment wrappers like formulas or functions instead of bare environments is deprecated as of rlang 0.3.0. This internal genericity was causing confusion (see issue #427). You should now extract the environment separately before calling these functions.

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)

```

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	<i>Does an object inherit from a set of classes?</i>
--------------	--

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"))
```

is_call	<i>Is object a call?</i>
---------	--------------------------

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. `is_unary_call()` and `is_binary_call()` hardcode `n` to 1 and 2.

Usage

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

Arguments

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

Life cycle

`is_lang()` has been soft-deprecated and renamed to `is_call()` in rlang 0.2.0 and similarly for `is_unary_lang()` and `is_binary_lang()`. This renaming follows the general switch from "language" to "call" in the rlang type nomenclature. See lifecycle section in [call2\(\)](#).

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	<i>Is object an environment?</i>
----------------	----------------------------------

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	<i>Is an object an expression?</i>
---------------	------------------------------------

Description

is_expression() tests for expressions, the set of objects that can be obtained from parsing R code. An expression can be one of two things: either a symbolic object (for which is_symbolic() returns TRUE), or a syntactic literal (testable with is_syntactic_literal()). Technically, calls can contain any R object, not necessarily symbolic objects or syntactic literals. However, this only happens in artificial situations. Expressions as we define them only contain numbers, strings, NULL, symbols, and calls: this is the complete set of R objects that can be created when R parses source code (e.g. from using [parse_expr\(\)](#)).

Note that we are using the term expression in its colloquial sense and not to refer to [expression\(\)](#) vectors, a data type that wraps expressions in a vector and which isn't used much in modern R code.

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).

Pairlists are also a kind of language objects. However, since they are mostly an internal data structure, `is_expression()` returns FALSE for pairlists. You can use `is_pairlist()` to explicitly check for them. Pairlists are the data structure for function arguments. They usually do not arise from R code because subsetting a call is a type-preserving operation. However, you can obtain the pairlist of arguments by taking the CDR of the call object from C code. The `rlang` function `node_cdr()` will do it from R. Another way in which pairlist of arguments arise is by extracting the argument list of a closure with `base::formals()` or `fn_fmls()`.

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	<i>Is object a formula?</i>
------------	-----------------------------

Description

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

Usage

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

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

Arguments

x	An object to test.
scoped	A boolean indicating whether the quosure is scoped, that is, has a valid environment attribute. If NULL, the scope is not inspected.
lhs	A boolean indicating whether the formula or definition has a left-hand side. If NULL, the LHS is not inspected.

Details

The scoped argument patterns-match on whether the scoped bundled with the quosure is valid or not. Invalid scopes may happen in nested quotations like `~~expr`, where the outer quosure is validly scoped but not the inner one. This is because `~` saves the environment when it is evaluated, and quoted formulas are by definition not evaluated.

Examples

```
x <- disp ~ am
is_formula(x)

is_formula(~10)
is_formula(10)

is_formula(quo(foo))
is_bare_formula(quo(foo))

# Note that unevaluated formulas are treated as bare formulas even
# though they don't inherit from "formula":
f <- quote(~foo)
is_bare_formula(f)

# However you can specify `scoped` if you need the predicate to
# return FALSE for these unevaluated formulas:
is_bare_formula(f, scoped = TRUE)
is_bare_formula(eval(f), scoped = TRUE)
```

is_function	<i>Is object a function?</i>
-------------	------------------------------

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_\(computer_programming\)](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 (see `ctxt_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)

# Primitive functions never appear in evaluation stacks:
is_primitive(base::`[[`)
is_primitive(base::list)
list(ctxt_stack())[[1]]

# While closures do:
identity(identity(ctxt_stack()))

# 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	<i>Is a package installed in the library?</i>
--------------	---

Description

This checks that a package is installed with minimal side effects. If installed, the package will be loaded but not attached.

Usage

```
is_installed(pkg)
```

Arguments

pkg The name of a package.

Value

TRUE if the package is installed, FALSE otherwise.

Examples

```
is_installed("utils")
is_installed("ggplot5")
```

is_integerish	<i>Is a vector integer-like?</i>
---------------	----------------------------------

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	<i>Is R running interactively?</i>
----------------	------------------------------------

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, an RStudio notebook, or testthat is in progress.

`with_interactive()` and `scoped_interactive()` set the global option conveniently.

Usage

```
is_interactive()

scoped_interactive(value = TRUE, frame = caller_env())

with_interactive(expr, value = TRUE)
```

Arguments

value	A single TRUE or FALSE. This overrides the return value of <code>is_interactive()</code> .
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	<i>Is object named?</i>
----------	-------------------------

Description

`is_named()` checks that `x` has names attributes, and that none of the names are missing or empty (NA or ""). `is_dictionaryish()` checks that an object is a dictionary: that it has actual names and in addition that there are no duplicated names. `have_name()` is a vectorised version of `is_named()`.

Usage

```
is_named(x)
```

```
is_dictionaryish(x)
```

```
have_name(x)
```

Arguments

`x` An object to test.

Value

`is_named()` and `is_dictionaryish()` are scalar predicates and return TRUE or FALSE. `have_name()` is vectorised and returns a logical vector as long as the input.

Examples

```
# A data frame usually has valid, unique names
is_named(mtcars)
have_name(mtcars)
is_dictionaryish(mtcars)

# But data frames can also have duplicated columns:
dups <- cbind(mtcars, cyl = seq_len(nrow(mtcars)))
is_dictionaryish(dups)

# The names are still valid:
is_named(dups)
have_name(dups)

# For empty objects the semantics are slightly different.
# is_dictionaryish() returns TRUE for empty objects:
is_dictionaryish(list())

# But is_named() will only return TRUE if there is a names
# attribute (a zero-length character vector in this case):
x <- set_names(list(), character(0))
is_named(x)

# Empty and missing names are invalid:
```

```
invalid <- dups
names(invalid)[2] <- ""
names(invalid)[5] <- NA

# is_named() performs a global check while have_name() can show you
# where the problem is:
is_named(invalid)
have_name(invalid)

# have_name() will work even with vectors that don't have a names
# attribute:
have_name(letters)
```

is_namespace	<i>Is an object a namespace environment?</i>
--------------	--

Description

Is an object a namespace environment?

Usage

```
is_namespace(x)
```

Arguments

x	An object to test.
---	--------------------

is_symbol	<i>Is object a symbol?</i>
-----------	----------------------------

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	<i>Is object identical to TRUE or FALSE?</i>
---------	--

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	<i>Is object a weak reference?</i>
------------	------------------------------------

Description

Is object a weak reference?

Usage

```
is_weakref(x)
```

Arguments

x	An object to test.
---	--------------------

last_error	<i>Last abort() error</i>
------------	---------------------------

Description

- `last_error()` returns the last error thrown with `abort()`. 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()
```

missing_arg	<i>Generate or handle a missing argument</i>
-------------	--

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.
- `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

<code>x</code>	An object that might be the missing argument.
<code>default</code>	The object to return if the input is missing, defaults to <code>missing_arg()</code> .

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.

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. This means that expressions like `x[[1]] <- missing_arg()` are perfectly safe. Likewise, `x[[1]]` is safe even if the result is the missing object.

However, as soon as the missing argument is passed down between functions through an argument, you're at risk of triggering a missing error. This is because arguments are passed through symbols. 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.

`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 formal 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.

Life cycle

- `missing_arg()` and `is_missing()` are stable.
- Like the rest of rlang, `maybe_missing()` is maturing.

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(1[[1]])

# while is_missing() will work as expected:
is_missing(missing_arg())
is_missing(1[[1]])
```

names2	<i>Get names of a vector</i>
--------	------------------------------

Description

Stable

This names getter 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 `""`.

Usage

```
names2(x)
```

Arguments

x A vector.

Life cycle

names2() is stable.

Examples

```
names2(letters)

# It also takes care of standardising missing names:
x <- set_names(1:3, c("a", NA, "b"))
names2(x)
```

new_formula	<i>Create a formula</i>
-------------	-------------------------

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	<i>Create a function</i>
--------------	--------------------------

Description**Stable**

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_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

<code>x</code>	A list of quosures or objects to coerce to quosures.
<code>env</code>	The default environment for the new quosures.
<code>named</code>	Whether to name the list with <code>quos_auto_name()</code> .

 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)

```

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"))
fieval <- .Person(name = "Fieval", species = "mouse")
fieval %@@ name

```

op-na-default *Replace missing values*

Description

This infix function is similar to `%||%` but is vectorised and provides a default value for missing elements. It is faster than using `base::ifelse()` and does not perform type conversions.

Usage

```
x %||% y
```

Arguments

`x` The original values.
`y` The replacement values. Must be of length 1 or the same length as `x`.

See Also

[op-null-default](#)

Examples

```
c("a", "b", NA, "c") %||% "default"
c(1L, NA, 3L, NA, NA) %||% (6L:10L)
```

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

Create pairlists with splicing support

Description

This pairlist constructor supports [tidy dots](#) features like `!!!`. Use it to manually create argument lists for calls or parameter lists for functions.

Usage

```
pairlist2(...)
```

Arguments

... 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.

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. Depending on the use case, a good default might be the global environment but 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

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.

The versions suffixed with _quo and _quos return [quosures](#) rather than raw expressions.

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 if one of the strings contains several expressions (such as "foo; bar").

Life cycle

- parse_quosure() and parse_quosures() were soft-deprecated in rlang 0.2.0 and renamed to parse_quo() and parse_quos(). This is consistent with the rule that abbreviated suffixes indicate the return type of a function.

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)")

# 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))
```

quasiquote	<i>Quasiquote of an expression</i>
------------	------------------------------------

Description

Quasiquote is the mechanism that makes it possible to program flexibly with tidy evaluation grammars like `dplyr`. It is enabled in all functions quoting their arguments with `enquo()`, `enexpr()`, or the plural variants.

Quasiquote is the combination of quoting an expression while allowing immediate evaluation (unquoting) of part of that expression. We provide both syntactic operators and functional forms for unquoting.

- The `!!` operator unquotes its argument. It gets evaluated immediately in the surrounding context.
- The `!!!` operator unquotes and splices its argument. The argument should represent a list or a vector. Each element will be embedded in the surrounding call, i.e. each element is inserted as an argument. If the vector is named, the names are used as argument names.
If the vector is a classed object (like a factor), it is converted to a list with `base::as.list()` to ensure proper dispatch. If it is an S4 objects, it is converted to a list with `methods::as()`.
- The `{{ }}` operator quotes and unquotes its argument in one go, a pattern that we call **interpolation**. It is an alias for `!!enquo(arg)`.
Like `enquo()`, `{{ }}` is used in functions to capture an argument as a quoted expression. This expression is immediately unquoted in place.

Use `qq_show()` to experiment with quasiquote or debug the effect of unquoting operators. `qq_show()` quotes its input, processes unquoted parts, and prints the result with `expr_print()`. This expression printer has a clearer output than the base R printer (see the [documentation topic](#)).

Usage

```
qq_show(expr)
```

Arguments

`expr` An expression to be quasiquoted.

Unquoting names

When a function takes multiple named arguments (e.g. `dplyr::mutate()`), it is difficult to supply a variable as name. Since the LHS of `=` is quoted, giving the name of a variable results in the argument having the name of the variable rather than the name stored in that variable. This problem is right up the alley for the unquoting operator `!!`. If you were able to unquote the variable when supplying the name, the argument would be named after the content of that variable.

Unfortunately R is very strict about the kind of expressions supported on the LHS of `=`. This is why we have made the more flexible `:=` operator an alias of `=`. You can use it to supply names, e.g. `a := b` is equivalent to `a = b`. Since its syntax is more flexible you can unquote on the LHS:

```
name <- "Jane"

list2(!!name := 1 + 2)
exprs(!!name := 1 + 2)
quos(!!name := 1 + 2)
```

Like `=`, the `:=` operator expects strings or symbols on its LHS.

Note that unquoting on the LHS of `:=` only works in top level expressions. These are all valid:

```
exprs(!!nm := x)
tibble(!!nm := x)
list2(!!nm := x)
```

But deep-unquoting names isn't supported:

```
expr(foo(!!nm := x))
exprs(foo(!!nm := x))
```

Theory

Formally, `quo()` and `expr()` are quasiquote functions, `!!` is the unquote operator, and `!!!` is the unquote-splice operator. These terms have a rich history in Lisp languages, and live on in modern languages like [Julia](#) and [Racket](#).

Life cycle

- Calling `UQ()` and `UQS()` with the `rlang` namespace qualifier is deprecated as of `rlang` 0.3.0. Just use the unqualified forms instead:

```
# Bad
rlang:::expr(mean(rlang:::UQ(var) * 100))

# Ok
rlang:::expr(mean(UQ(var) * 100))

# Good
rlang:::expr(mean(!!var * 100))
```

Supporting namespace qualifiers complicates the implementation of unquotation and is misleading as to the nature of unquoting operators (which are syntactic operators that operates at quotation-time rather than function calls at evaluation-time).

- `UQ()` and `UQS()` were soft-deprecated in `rlang` 0.2.0 in order to make the syntax of quasiquote more consistent. The prefix forms are now `!!`()` and `!!!`()` which is consistent with other R operators (e.g. ``+` (a, b)` is the prefix form of `a + b`).

Note that the prefix forms are not as relevant as before because `!!` now has the right operator precedence, i.e. the same as unary `-` or `+`. It is thus safe to mingle it with other operators, e.g. `!!a + !!b` does the right thing. In addition the parser now strips one level of parentheses around unquoted expressions. This way `(!!"foo")(...)` expands to `foo(...)`. These changes make the prefix forms less useful.

Finally, the named functional forms `UQ()` and `UQS()` were misleading because they suggested that existing knowledge about functions is applicable to quasiquote. This was reinforced by the visible definitions of these functions exported by `rlang` and by the tidy eval parser interpreting `rlang:::UQ()` as `!!`. In reality unquoting is *not* a function call, it is a syntactic operation. The operator form makes it clearer that unquoting is special.

Examples

```
# Interpolation with {{ }} is the easiest way to forward
# arguments to tidy eval functions:
if (is_attached("package:dplyr")) {
```

```

# Forward all arguments involving data frame columns by
# interpolating them within other data masked arguments.
# Here we interpolate `arg` in a `summarise()` call:
my_function <- function(data, arg) {
  summarise(data, avg = mean({{ arg }}), na.rm = TRUE)
}

my_function(mtcars, cyl)
my_function(mtcars, cyl * 10)

# The operator is just a shortcut for `!!enquo()`:
my_function <- function(data, arg) {
  summarise(data, avg = mean(!enquo(arg), na.rm = TRUE))
}

my_function(mtcars, cyl)

}

# Quasiquote functions quote expressions like base::quote()
quote(how_many(this))
expr(how_many(this))
quo(how_many(this))

# In addition, they support unquoting. Let's store symbols
# (i.e. object names) in variables:
this <- sym("apples")
that <- sym("oranges")

# With unquotation you can insert the contents of these variables
# inside the quoted expression:
expr(how_many(!this))
expr(how_many(!that))

# You can also insert values:
expr(how_many(!(1 + 2)))
quo(how_many(!(1 + 2)))

# Note that when you unquote complex objects into an expression,
# the base R printer may be a bit misleading. For instance compare
# the output of `expr()` and `quo()` (which uses a custom printer)
# when we unquote an integer vector:
expr(how_many(!(1:10)))
quo(how_many(!(1:10)))

# This is why it's often useful to use qq_show() to examine the
# result of unquotation operators. It uses the same printer as
# quosures but does not return anything:
qq_show(how_many(!(1:10)))

# Use `!!!` to add multiple arguments to a function. Its argument
# should evaluate to a list or vector:
args <- list(1:3, na.rm = TRUE)
quo(mean(!!!args))

```

```

# You can combine the two
var <- quote(xyz)
extra_args <- list(trim = 0.9, na.rm = TRUE)
quo(mean(!var , !!!extra_args))

# The plural versions have support for the `:=` operator.
# Like `=`, `:=` creates named arguments:
quo(mouse1 := bernard, mouse2 = bianca)

# The `:=` is mainly useful to unquote names. Unlike `=` it
# supports `!!` on its LHS:
var <- "unquote me!"
quo(!var := bernard, mouse2 = bianca)

# All these features apply to dots captured by enquos():
fn <- function(...) enquos(...)
fn(!!!args, !var := penny)

# Unquoting is especially useful for building an expression by
# expanding around a variable part (the unquoted part):
quo1 <- quo(toupper(foo))
quo1

quo2 <- quo(paste(!quo1, bar))
quo2

quo3 <- quo(list(!quo2, !!!syms(letters[1:5])))
quo3

```

quosure

Quosure getters, setters and testers

Description

A quosure is a type of [quoted expression](#) that includes a reference to the context where it was created. A quosure is thus guaranteed to evaluate in its original environment and can refer to local objects.

You can access the quosure components (its expression and its environment) with:

- [get_expr\(\)](#) and [get_env\(\)](#). These getters also support other kinds of objects such as formulas.
- [quo_get_expr\(\)](#) and [quo_get_env\(\)](#). These getters only work with quosures and throw an error with other types of input.

Test if an object is a quosure with [is_quosure\(\)](#). If you know an object is a quosure, use the [quo_](#)-prefixed predicates to check its contents, [quo_is_missing\(\)](#), [quo_is_symbol\(\)](#), etc.

Usage

```

is_quosure(x)

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

x	An object to test.
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, <code>is_call()</code> returns FALSE. If any other string, <code>is_call()</code> 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 <code>is_call()</code> returns FALSE.
expr	A new expression for the quosure.
env	A new environment for the quosure.

Quosured constants

A quosure usually does not carry environments for [constant objects](#) like strings or numbers. `quo()` and `enquo()` only capture an environment for [symbolic expressions](#). For instance, all of these return the [empty environment](#):

```

quo_get_env(quo("constant"))
quo_get_env(quo(100))
quo_get_env(quo(NA))

```

On the other hand, quosures capture the environment of symbolic expressions, i.e. expressions whose meaning depends on the environment in which they are evaluated and what objects are defined there:

```

quo_get_env(quo(some_object))
quo_get_env(quo(some_function()))

```

Empty quosures

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.

Life cycle

- `is_quosure()` is stable.
- `quo_get_expr()` and `quo_get_env()` are stable.

See Also

`quo()` for creating quosures by quotation; `as_quosure()` and `new_quosure()` for constructing quosures manually.

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 whether 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())
```

Description

Stable

Quotation is a mechanism by which an expression supplied as argument is captured by a function. Instead of seeing the value of the argument, the function sees the recipe (the R code) to make that value. This is possible because R [expressions](#) are representable as regular objects in R:

- Calls represent the action of calling a function to compute a new value. Evaluating a call causes that value to be computed. Calls typically involve symbols to reference R objects.
- Symbols represent the name that is given to an object in a particular context (an [environment](#)).

We call objects containing calls and symbols [expressions](#). There are two ways to create R expressions. First you can **build** calls and symbols from parts and pieces (see [sym\(\)](#), [syms\(\)](#) and [call2\(\)](#)). The other way is by *quotation* or *quasiquotation*, i.e. by intercepting an expression instead of evaluating it.

Usage

```
expr(expr)
```

```
enexpr(arg)
```

```
exprs(..., .named = FALSE, .ignore_empty = c("trailing", "none",
  "all"), .unquote_names = TRUE)
```

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

```
ensym(arg)
```

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

```
quo(expr)
```

```
enquo(arg)
```

```
quos(..., .named = FALSE, .ignore_empty = c("trailing", "none", "all"),
  .unquote_names = TRUE)
```

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

Arguments

<code>expr</code>	An expression.
<code>arg</code>	A symbol representing an argument. The expression supplied to that argument will be captured instead of being evaluated.
<code>...</code>	For <code>enexprs()</code> , <code>ensyms()</code> and <code>enquos()</code> , names of arguments to capture without evaluation (including <code>...</code>). For <code>exprs()</code> and <code>quos()</code> , the expressions to capture unevaluated (including expressions contained in <code>...</code>).

<code>.named</code>	Whether to ensure all dots are named. Unnamed elements are processed with <code>quo_name()</code> to build a default name. See also <code>quos_auto_name()</code> .
<code>.ignore_empty</code>	Whether to ignore empty arguments. Can be one of "trailing", "none", "all". If "trailing", only the last argument is ignored if it is empty. Note that "trailing" applies only to arguments passed in <code>...</code> , not to named arguments. On the other hand, "all" also applies to named arguments.
<code>.unquote_names</code>	Whether to treat <code>:=</code> as <code>=</code> . Unlike <code>=</code> , the <code>:=</code> syntax supports <code>!!</code> unquoting on the LHS.
<code>.homonyms</code>	How to treat arguments with the same name. The default, "keep", preserves these arguments. Set <code>.homonyms</code> 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.
<code>.check_assign</code>	Whether to check for <code><-</code> calls passed in dots. When TRUE and a <code><-</code> call is detected, a warning is issued to advise users to use <code>=</code> if they meant to match a function parameter, or wrap the <code><-</code> call in braces otherwise. This ensures assignments are explicit.

User expressions versus your expressions

There are two points of view when it comes to capturing an expression:

- You can capture the expressions supplied by *the user* of your function. This is the purpose of `ensym()`, `enexpr()` and `enquo()` and their plural variants. These functions take an argument name and capture the expression that was supplied to that argument.
- You can capture the expressions that *you* supply. To this end use `expr()` and `quo()` and their plural variants `exprs()` and `quos()`.

Capture raw expressions

- `enexpr()` and `expr()` capture a single raw expression.
- `enexprs()` and `exprs()` capture a list of raw expressions including expressions contained in `...`
- `ensym()` and `ensyms()` are variants of `enexpr()` and `enexprs()` that check the captured expression is either a string (which they convert to symbol) or a symbol. If anything else is supplied they throw an error.

In terms of base functions, `enexpr(arg)` corresponds to `base::substitute(arg)` (though that function also features complex substitution semantics) and `expr()` is like `quote()` (and `bquote()` if we consider unquotation syntax). The plural variant `exprs()` is equivalent to `base::alist()`. Finally there is no function in base R that is equivalent to `enexprs()` but you can reproduce its behaviour with `eval(substitute(alist(...)))`.

Capture expressions in quosures

`quo()` and `enquo()` are similar to their `expr` counterparts but capture both the expression and its environment in an object called a quosure. This wrapper contains a reference to the original environment in which that expression was captured. Keeping track of the environments of expressions is important because this is where functions and objects mentioned in the expression are defined.

Quosures are objects that can be evaluated with `eval_tidy()` just like symbols or function calls. Since they always evaluate in their original environment, quosures can be seen as vehicles that allow expressions to travel from function to function but that beam back instantly to their original environment upon evaluation.

See the [quosure](#) help topic about tools to work with quosures.

Quasiquotation

All quotation functions in `rlang` have support for [unquoting operators](#). The combination of quotation and unquotation is called *quasiquotation*.

Unquotation provides a way to refer to variables during quotation. Variables are problematic when quoting because a captured expression is essentially a constant, just like a string is a constant. For instance in all the following cases `apple` is a constant: `~apple`, `"apple"` and `expr(apple)`. Unquoting allows you to introduce a part of variability within a captured expression.

- In the case of `enexpr()` and `enquo()`, unquoting provides an escape hatch to the users of your function that allows them to manipulate the expression that you capture.
- In the case of `expr()` and `quo()`, quasiquotation lets you build a complex expressions where some parts are constant (the parts that are captured) and some parts are variable (the parts that are unquoted).

See the [quasiquotation](#) help topic for more about this as well as [the chapter in Advanced R](#).

Examples

```
# expr() and exprs() capture expressions that you supply:
expr(symbol)
exprs(several, such, symbols)

# enexpr() and enexprs() capture expressions that your user supplied:
expr_inputs <- function(arg, ...) {
  user_exprs <- enexprs(arg, ...)
  user_exprs
}
expr_inputs(hello)
expr_inputs(hello, bonjour, ciao)

# ensym() and ensyms() provide additional type checking to ensure
# the user calling your function has supplied bare object names:
sym_inputs <- function(...) {
  user_symbols <- ensyms(...)
  user_symbols
}
sym_inputs(hello, "bonjour")
## sym_inputs(say(hello)) # Error: Must supply symbols or strings
expr_inputs(say(hello))

# All these quoting functions have quasiquotation support. This
# means that you can unquote (evaluate and inline) part of the
# captured expression:
what <- sym("bonjour")
expr(say(what))
expr(say(!what))

# This also applies to expressions supplied by the user. This is
# like an escape hatch that allows control over the captured
# expression:
expr_inputs(say(!what), !what)

# Finally, you can capture expressions as quosures. A quosure is an
```

```

# object that contains both the expression and its environment:
quo <- quo(letters)
quo

get_expr(quo)
get_env(quo)

# Quosures can be evaluated with eval_tidy():
eval_tidy(quo)

# They have the nice property that you can pass them around from
# context to context (that is, from function to function) and they
# still evaluate in their original environment:
multiply_expr_by_10 <- function(expr) {
  # We capture the user expression and its environment:
  expr <- enquos(expr)

  # Then create an object that only exists in this function:
  local_ten <- 10

  # Now let's create a multiplication expression that (a) inlines
  # the user expression as LHS (still wrapped in its quosure) and
  # (b) refers to the local object in the RHS:
  quo(!expr * local_ten)
}
quo <- multiply_expr_by_10(2 + 3)

# The local parts of the quosure are printed in colour if your
# terminal is capable of displaying colours:
quo

# All the quosures in the expression evaluate in their original
# context. The local objects are looked up properly and we get the
# expected result:
eval_tidy(quo)

```

quo_label

Format quosures for printing or labelling

Description

Questioning

Note: You should now use `as_label()` or `as_name()` instead of `quo_name()`. See life cycle section below.

These functions take an arbitrary R object, typically an [expression](#), and represent it as a string.

- `quo_name()` returns an abbreviated representation of the object as a single line string. It is suitable for default names.
- `quo_text()` returns a multiline string. For instance block expressions like `{ foo; bar }` are represented on 4 lines (one for each symbol, and the curly braces on their own lines).

These deparsers are only suitable for creating default names or printing output at the console. The behaviour of your functions should not depend on deparsed objects. If you are looking for a way of transforming symbols to strings, use `as_string()` instead of `quo_name()`. Unlike deparsing, the transformation between symbols and strings is non-lossy and well defined.

Usage

```
quo_label(quo)

quo_text(quo, width = 60L, nlines = Inf)

quo_name(quo)
```

Arguments

quo	A quosure or expression.
width	Width of each line.
nlines	Maximum number of lines to extract.

Life cycle

These functions are in the questioning life cycle stage.

- `as_label()` and `as_name()` should be used instead of `quo_name()`. `as_label()` transforms any R object to a string but should only be used to create a default name. Labelisation is not a well defined operation and no assumption should be made about the label. On the other hand, `as_name()` only works with (possibly quosured) symbols, but is a well defined and deterministic operation.
- We don't have a good replacement for `quo_text()` yet. See <https://github.com/r-lib/rlang/issues/636> to follow discussions about a new deparsing API.

See Also

[expr_label\(\)](#), [f_label\(\)](#)

Examples

```
# Quosures can contain nested quosures:
quo <- quo(foo(!! quo(bar)))
quo

# quo_squash() unwraps all quosures and returns a raw expression:
quo_squash(quo)

# This is used by quo_text() and quo_label():
quo_text(quo)

# Compare to the unwrapped expression:
expr_text(quo)

# quo_name() is helpful when you need really short labels:
quo_name(quo(sym))
quo_name(quo(!! sym))
```

quo_squash	<i>Squash a quosure</i>
------------	-------------------------

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 `quo_label()` or `quo_name()`). 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 <code>quo_squash()</code> 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.

Life cycle

This function replaces `quo_expr()` which was deprecated in rlang 0.2.0. `quo_expr()` was a misnomer because it implied that it was a mere expression accessor for quosures whereas it was really a lossy operation that squashed all nested quosures.

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	<i>Create vectors matching the length of a given vector</i>
-----------	---

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

Errors thrown with `abort()` automatically save a backtrace that can be inspected by calling `last_error()`. Optionally, you can also display the backtrace alongside the error message by setting the option `rlang_backtrace_on_error` to one of the following values:

- "reminder": Display a reminder that the backtrace can be inspected by calling `rlang::last_error()`.
- "branch": Display a simplified backtrace.
- "collapse": Display a collapsed backtrace tree.
- "full": Display the full backtrace tree.

Promote base errors to rlang errors

Call `options(error = rlang::entrace)` to instrument base errors with rlang features. This handler does two things:

- It saves the base error as an rlang object. This allows you to 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.

Examples

```
# Display a simplified backtrace on error for both base and rlang
# errors:

# options(
#   rlang_backtrace_on_error = "branch",
#   error = rlang::entrace
# )
# stop("foo")
```

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_character(x, encoding = NULL)

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.
encoding	Defunct as of rlang 0.4.0.
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](#)

scoped_bindings	<i>Temporarily change bindings of an environment</i>
-----------------	--

Description

- `scoped_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 `scoped_bindings()`.

Usage

```
scoped_bindings(..., .env = .frame, .frame = caller_env())
```

```
with_bindings(.expr, ..., .env = caller_env())
```

Arguments

<code>...</code>	Pairs of names and values. These dots support splicing (with value semantics) and name unquoting.
<code>.env</code>	An environment.
<code>.frame</code>	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.
<code>.expr</code>	An expression to evaluate with temporary bindings.

Value

`scoped_bindings()` returns the values of old bindings invisibly; `with_bindings()` returns the value of `expr`.

Examples

```
foo <- "foo"
bar <- "bar"

# `foo` will be temporarily rebounded while executing `expr`
with_bindings(paste(foo, bar), foo = "rebound")
paste(foo, bar)
```

scoped_options *Change global options*

Description

- `scoped_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

```
scoped_options(..., .frame = caller_env())

with_options(.expr, ...)

push_options(...)

peek_options(...)

peek_option(name)
```

Arguments

<code>...</code>	For <code>scoped_options()</code> and <code>push_options()</code> , named values defining new option values. For <code>peek_options()</code> , strings or character vectors of option names.
<code>.frame</code>	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.
<code>.expr</code>	An expression to evaluate with temporary options.
<code>name</code>	An option name as string.

Value

For `scoped_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() {
  scoped_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")

```

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

<code>from</code>	The starting point of the sequence.
<code>to</code>	The end point.
<code>x</code>	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_expr`*Set and get an expression*

Description

These helpers are useful to make your function work generically with quosures and raw expressions. First call `get_expr()` to extract an expression. Once you're done processing the expression, call `set_expr()` on the original object to update the expression. You can return the result of `set_expr()`, either a formula or an expression depending on the input type. Note that `set_expr()` does not change its input, it creates a new object.

Usage

```
set_expr(x, value)
```

```
get_expr(x, default = x)
```

Arguments

<code>x</code>	An expression, closure, or one-sided formula. In addition, <code>set_expr()</code> accept frames.
<code>value</code>	An updated expression.
<code>default</code>	A default expression to return when <code>x</code> is not an expression wrapper. Defaults to <code>x</code> itself.

Value

The updated original input for `set_expr()`. A raw expression for `get_expr()`.

See Also

[quo_get_expr\(\)](#) and [quo_set_expr\(\)](#) for versions of [get_expr\(\)](#) and [set_expr\(\)](#) that only work on quosures.

Examples

```
f <- ~foo(bar)
e <- quote(foo(bar))
frame <- identity(identity(ctxt_frame()))

get_expr(f)
get_expr(e)
get_expr(frame)

set_expr(f, quote(baz))
set_expr(e, quote(baz))
```

`set_names`*Set names of a vector*

Description

Stable

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`.

You can specify names in the following ways:

- If you do nothing, `x` will be named with itself.
- 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.
- If `nm` is `NULL`, the names are removed (if present).
- In all other cases, `nm` and `...` are coerced to character.

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 omitted 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")
```

sym	<i>Create a symbol or list of symbols</i>
-----	---

Description

These functions take strings as input and turn them into symbols.

Usage

```
sym(x)

syms(x)
```

Arguments

x A string or list of strings.

Value

A symbol for `sym()` and a list of symbols for `syms()`.

Examples

```
# 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()))
```

tidy-dots	<i>Collect dots as lists</i>
-----------	------------------------------

Description

`list2()` is equivalent to `list(...)` with a few additional features:

- You can splice other lists with the `unquote-splice !!!` operator.
- You can unquote names by using the `unquote !!` operator on the left-hand side of `:=`.
- Trailing commas are ignored, making it easier to copy and paste arguments.

For lack of a better name, these features are collectively called "tidy dots".

`dots_list()` is a lower-level version of `list2()` that offers additional parameters for dots capture.

Usage

```
dots_list(..., .ignore_empty = c("trailing", "none", "all"),
  .preserve_empty = FALSE, .homonyms = c("keep", "first", "last",
  "error"), .check_assign = FALSE)

list2(...)
```

Arguments

...	Arguments to collect with <code>!!!</code> support.
<code>.ignore_empty</code>	Whether to ignore empty arguments. Can be one of "trailing", "none", "all". If "trailing", only the last argument is ignored if it is empty.
<code>.preserve_empty</code>	Whether to preserve the empty arguments that were not ignored. If TRUE, empty arguments are stored with <code>missing_arg()</code> values. If FALSE (the default) an error is thrown when an empty argument is detected.
<code>.homonyms</code>	How to treat arguments with the same name. The default, "keep", preserves these arguments. Set <code>.homonyms</code> 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.
<code>.check_assign</code>	Whether to check for <code><-</code> calls passed in dots. When TRUE and a <code><-</code> call is detected, a warning is issued to advise users to use <code>=</code> if they meant to match a function parameter, or wrap the <code><-</code> call in braces otherwise. This ensures assignments are explicit.

Value

A list of arguments. This list is always named: unnamed arguments are named with the empty string "".

See Also

`exprs()` for extracting dots without evaluation.

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 arguments trigger an error by default:
try(fn(, ))

# You can choose to preserve empty arguments instead:
list3 <- function(...) dots_list(..., .preserve_empty = TRUE)

# Note how the last empty argument is still ignored because
# `.ignore_empty` defaults to "trailing":
list3(, )

# The list with preserved empty arguments is equivalent to:
list(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 })

```

Description

This pronoun allows you to be explicit when you refer to an object inside the data. Referring to the `.data` pronoun rather than to the original data frame has several advantages:

- It makes it easy to refer to column names stored as strings. If `var` contains the column "height", the pronoun will subset that column:

```

var <- "height"
dplyr::summarise(df, mean(.data[[var]]))

```

The index variable `var` is `unquoted`, which ensures a column named `var` in the data frame cannot mask it. This makes the pronoun safe to use in functions and packages.

- Sometimes a computation is not about the whole data but about a subset. For example if you supply a grouped data frame to a dplyr verb, the `.data` pronoun contains the group subset.
- It lets dplyr know that you're referring to a column from the data which is helpful to generate correct queries when the source is a database.

The `.data` object exported here is useful to import in your package namespace to avoid a R CMD check note when referring to objects from the data mask.

Usage

```
.data
```

trace_back	<i>Capture a backtrace</i>
------------	----------------------------

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 `summary()` method of this object will reveal.

Usage

```
trace_back(top = NULL, bottom = NULL)
```

```
trace_length(trace)
```

Arguments

<code>top</code>	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 <code>trace_back()</code> 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.
<code>bottom</code>	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 <code>caller_env()</code> .
<code>trace</code>	A backtrace created by <code>trace_back()</code> .

Details

`trace_length()` returns the number of frames in a backtrace.

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)

```


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_character(x, n = NULL, encoding = NULL)
is_logical(x, n = NULL)
is_raw(x, n = NULL)
is_bytes(x, n = NULL)
is_null(x)
```

Arguments

<code>x</code>	Object to be tested.
<code>n</code>	Expected length of a vector.
<code>finite</code>	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.
<code>encoding</code>	Defunct as of rlang 0.4.0.

Details

Compared to base R functions:

- The predicates for vectors include the `n` argument for pattern-matching on the vector length.
- Unlike `is.atomic()`, `is_atomic()` does not return TRUE for NULL.
- 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](#)

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 [tidy dots](#) and thus support splicing with `!!!`.

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 ctors 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))

```

with_abort

Promote all errors to rlang errors

Description

with_abort() promotes conditions as if they were thrown with [abort\(\)](#). These errors embed a [backtrace](#). They are particularly suitable to be set as *parent errors* (see *parent* argument of [abort\(\)](#)).

Usage

```
with_abort(expr, classes = "error")
```

Arguments

expr	An expression run in a context where errors are promoted to rlang errors.
classes	Character vector of condition classes that should be promoted to rlang errors.

Details

with_abort() installs a [calling handler](#) for errors and rethrows non-rlang errors with [abort\(\)](#). However, error handlers installed *within* with_abort() have priority. For this reason, you should use [tryCatch\(\)](#) and [exiting](#) handlers outside with_abort() rather than inside.

Examples

```

# with_abort() automatically casts simple errors thrown by stop()
# to rlang errors. It is handy for rethrowing low level
# errors. The backtraces are then segmented between the low level
# and high level contexts.
f <- function() g()
g <- function() stop("Low level error")

high_level <- function() {

```

```

with_handlers(
  with_abort(f()),
  error = ~ abort("High level error", parent = .)
)
}

```

with_handlers

Establish handlers on the stack

Description

Condition handlers are functions established on the evaluation stack (see `ctxt_stack()`) that are called by R when a condition is signalled (see `cond_signal()` and `abort()` for two common signal functions). They come in two types:

- Exiting handlers aborts all code currently run between `with_handlers()` and the point where the condition has been raised. `with_handlers()` passes the return value of the handler to its caller.
- Calling handlers, which are executed from inside the signalling functions. Their return values are ignored, only their side effects matters. Valid side effects are writing a log message, or jumping out of the signalling context by [invoking a restart](#) or using `return_from()`. If the raised condition was an error, this interrupts the aborting process.

If a calling handler returns normally, it effectively declines to handle the condition and other handlers on the stack (calling or exiting) are given a chance to handle the condition.

Handlers are exiting by default, use `calling()` to create a calling handler.

Usage

```
with_handlers(.expr, ...)
```

```
calling(handler)
```

Arguments

<code>.expr</code>	An expression to execute in a context where new handlers are established. The underscored version takes a quoted expression or a quoted formula.
<code>...</code>	Named handlers. These should be functions of one argument. These handlers are treated as exiting by default. Use <code>calling()</code> to specify a calling handler. These dots support tidy dots features and are passed to <code>as_function()</code> to enable the formula shortcut for lambda functions.
<code>handler</code>	A handler function that takes a condition as argument. This is passed to <code>as_function()</code> and can thus be a formula describing a lambda function.

Life cycle

`exiting()` is soft-deprecated as of rlang 0.4.0 because `with_handlers()` now treats handlers as exiting by default.

Examples

```

# Signal a condition with signal():
fn <- function() {
  g()
  cat("called?\n")
  "fn() return value"
}
g <- function() {
  h()
  cat("called?\n")
}
h <- function() {
  signal("A foobar condition occurred", "foo")
  cat("called?\n")
}

# Exiting handlers jump to with_handlers() before being
# executed. Their return value is handed over:
handler <- function(c) "handler return value"
with_handlers(fn(), foo = handler)

# Calling handlers are called in turn and their return value is
# ignored. Returning just means they are declining to take charge of
# the condition. However, they can produce side-effects such as
# displaying a message:
some_handler <- function(c) cat("some handler!\n")
other_handler <- function(c) cat("other handler!\n")
with_handlers(fn(), foo = calling(some_handler), foo = calling(other_handler))

# If a calling handler jumps to an earlier context, it takes
# charge of the condition and no other handler gets a chance to
# deal with it. The canonical way of transferring control is by
# jumping to a restart. See with_restarts() and restarting()
# documentation for more on this:
exiting_handler <- function(c) rst_jump("rst_foo")
fn2 <- function() {
  with_restarts(g(), rst_foo = function() "restart value")
}
with_handlers(fn2(), foo = calling(exiting_handler), foo = calling(other_handler))

```

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	<i>Create zap objects</i>
-----	---------------------------

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()))
```

Index

!! (quasiquote), 81
!!! (quasiquote), 81
*Topic **datasets**
 tidyeval-data, 102
*Topic **experimental**
 new_weakref, 76
 scoped_options, 96
.Internal(), 65
.Primitive(), 65
.data, 9
.data (tidyeval-data), 102
:= (quasiquote), 81

abort, 3
abort(), 4, 27, 28, 71, 93, 107, 108
active bindings, 8
add_backtrace
 (rlang_backtrace_on_error), 93
arg_match, 6
as_box, 7
as_box_if (as_box), 7
as_closure (as_function), 12
as_data_mask, 7
as_data_mask(), 31, 45
as_data_pronoun (as_data_mask), 7
as_environment, 11
as_environment(), 31
as_function, 12
as_function(), 33, 108
as_label, 13
as_label(), 14, 90, 91
as_name, 14
as_name(), 13, 14, 17, 90, 91
as_quosure, 15
as_quosure(), 75, 86
as_quosures (new_quosures), 75
as_string, 16
as_string(), 13, 14, 90

backtrace, 107
bare-type-predicates, 17, 95, 105
base environment, 80
base::.Internal(), 65
base::alist(), 88
base::as.call(), 19
base::as.list(), 81
base::as.name(), 14, 16
base::as.symbol(), 14, 16
base::assign(), 33
base::body(), 52
base::call(), 19
base::delayedAssign(), 33
base::eval(), 8, 43, 45
base::formals(), 62
base::I(), 18
base::ifelse(), 78
base::inherits(), 58
base::interactive(), 67
base::is.integer(), 66
base::makeActiveBinding(), 33
base::match.arg(), 6
base::match.call(), 23, 26
base::message(), 4, 27
base::missing(), 71
base::parse(), 80
base::quote(), 74
base::stop(), 4, 27
base::structure(), 18
base::suppressMessages(), 5
base::suppressWarnings(), 5
base::typeof(), 20, 105
base::warning(), 4, 27
base_env(), 62
box, 18
boxed, 29
bquote(), 88
bytes (vector-construction), 106

c(), 106
call, 59
call2, 19
call2(), 21, 22, 24–26, 59, 87
call_args, 21
call_args(), 53
call_args_names (call_args), 21
call_args_names(), 53
call_fn, 22
call_fn(), 25

- call_inspect, 23
- call_modify, 23
- call_modify(), 110
- call_name, 25
- call_name(), 22
- call_ns (call_name), 25
- call_standardise, 26
- call_standardise(), 23
- callable, 19
- caller frame, 6
- caller_env, 20
- caller_env(), 30, 103
- caller_frame(), 21
- calling, 29
- calling (with_handlers), 108
- calling handler, 107
- calling(), 108
- catch_cnd, 27
- child_env (env), 30
- chr (vector-construction), 106
- closure, 12
- cnd(), 28
- cnd_entrace (entrace), 29
- cnd_entrace(), 30
- cnd_message(), 4
- cnd_signal, 27
- cnd_signal(), 28, 108
- cnd_type(), 27
- constant objects, 85
- constructed calls, 53
- cpl (vector-construction), 106
- ctxt_stack(), 52, 56, 64, 108
- current_env (caller_env), 20
- current_env(), 56
- current_frame(), 21

- dbl (vector-construction), 106
- definition, 63
- documentation topic, 81
- done, 28
- dots_list (tidy-dots), 100

- empty environment, 31, 36, 85, 86
- empty_env, 29
- empty_env(), 11, 40
- enexpr (quotation), 86
- enexprs (quotation), 86
- enquo (quotation), 86
- enquo(), 85, 86
- enquos (quotation), 86
- ensym (quotation), 86
- ensyms (quotation), 86
- entrace, 29

- env, 30
- env(), 34, 37, 40, 52
- env_bind, 33
- env_bind(), 31, 42, 110
- env_bind_active (env_bind), 33
- env_bind_lazy (env_bind), 33
- env_clone, 36
- env_depth, 36
- env_get, 37
- env_get_list (env_get), 37
- env_has, 38
- env_has(), 31, 57
- env_inherits, 38
- env_label (env_name), 39
- env_length (env_names), 39
- env_name, 39
- env_name(), 40
- env_names, 39
- env_parent, 40
- env_parents (env_parent), 40
- env_poke_parent (get_env), 56
- env_print, 42
- env_tail (env_parent), 40
- env_unbind, 42
- environment, 87
- eval_bare, 43
- eval_tidy, 45
- eval_tidy(), 8, 9, 44, 88
- exec, 47
- exiting, 107
- expr (quotation), 86
- expr_deparse (expr_print), 51
- expr_interp, 49
- expr_label, 50
- expr_label(), 55, 91
- expr_name (expr_label), 50
- expr_name(), 48
- expr_print, 51
- expr_print(), 81
- expr_text (expr_label), 50
- expr_text(), 55
- expression, 15, 80, 90
- expression(), 61
- expressions, 87
- exprs (quotation), 86
- exprs(), 33, 74, 101
- exprs_auto_name, 48

- f_env (f_rhs), 54
- f_env<- (f_rhs), 54
- f_label (f_text), 55
- f_label(), 91
- f_lhs (f_rhs), 54

- f_lhs<- (f_rhs), 54
- f_name (f_text), 55
- f_rhs, 54
- f_rhs<- (f_rhs), 54
- f_text, 55
- fancy bindings, 42
- fn_body, 52
- fn_body<- (fn_body), 52
- fn_env, 52
- fn_env<- (fn_env), 52
- fn_fmlls, 53
- fn_fmlls(), 21, 62, 65
- fn_fmlls<- (fn_fmlls), 53
- fn_fmlls_names (fn_fmlls), 53
- fn_fmlls_names(), 21
- fn_fmlls_names<- (fn_fmlls), 53
- fn_fmlls_syms (fn_fmlls), 53
- formals(), 65
- formula, 63

- get_env, 56
- get_env(), 42, 56, 84
- get_expr (set_expr), 98
- get_expr(), 84, 98
- global environment, 80

- has_name, 57
- have_name (is_named), 68

- imports environments, 39
- inform (abort), 3
- inform(), 27, 28
- inherits_all (inherits_any), 58
- inherits_all(), 18
- inherits_any, 58
- inherits_only (inherits_any), 58
- int (vector-construction), 106
- interrupt (abort), 3
- interrupt(), 27
- invoking a restart, 108
- is_atomic (type-predicates), 104
- is_atomic(), 18
- is_bare_atomic (bare-type-predicates), 17
- is_bare_bytes (bare-type-predicates), 17
- is_bare_character (bare-type-predicates), 17
- is_bare_double (bare-type-predicates), 17
- is_bare_environment (is_environment), 61
- is_bare_formula (is_formula), 63
- is_bare_integer (bare-type-predicates), 17
- is_bare_integerish (is_integerish), 66
- is_bare_list (bare-type-predicates), 17
- is_bare_logical (bare-type-predicates), 17
- is_bare_numeric (bare-type-predicates), 17
- is_bare_numeric(), 67
- is_bare_raw (bare-type-predicates), 17
- is_bare_string (bare-type-predicates), 17
- is_bare_vector (bare-type-predicates), 17
- is_bool (scalar-type-predicates), 94
- is_box (box), 18
- is_bytes (type-predicates), 104
- is_call, 59
- is_call(), 62
- is_character (type-predicates), 104
- is_closure (is_function), 64
- is_dictionaryish (is_named), 68
- is_done_box (done), 28
- is_double (type-predicates), 104
- is_empty, 60
- is_environment, 61
- is_expression, 61
- is_expression(), 40, 60
- is_false (is_true), 70
- is_formula, 63
- is_function, 64
- is_function(), 12
- is_installed, 66
- is_integer (type-predicates), 104
- is_integerish, 66
- is_interactive, 67
- is_lambda (as_function), 12
- is_list (type-predicates), 104
- is_logical (type-predicates), 104
- is_missing (missing_arg), 71
- is_named, 68
- is_namespace, 69
- is_null (type-predicates), 104
- is_primitive (is_function), 64
- is_primitive_eager (is_function), 64
- is_primitive_lazy (is_function), 64
- is_quosure (quosure), 84
- is_quosure(), 15
- is_quosures (new_quosures), 75
- is_raw (type-predicates), 104
- is_scalar_atomic (scalar-type-predicates), 94
- is_scalar_bytes (scalar-type-predicates), 94

- is_scalar_character
 - (scalar-type-predicates), 94
- is_scalar_double
 - (scalar-type-predicates), 94
- is_scalar_integer
 - (scalar-type-predicates), 94
- is_scalar_integerish(is_integerish), 66
- is_scalar_list
 - (scalar-type-predicates), 94
- is_scalar_logical
 - (scalar-type-predicates), 94
- is_scalar_raw(scalar-type-predicates), 94
- is_scalar_vector
 - (scalar-type-predicates), 94
- is_string(scalar-type-predicates), 94
- is_symbol, 69
- is_symbolic(is_expression), 61
- is_syntactic_literal(is_expression), 61
- is_true, 70
- is_vector(type-predicates), 104
- is_weakref, 70
- is_weakref(), 76, 110
- is_zap(zap), 110

- label, 42
- lapply(), 47
- last_error, 71
- last_error(), 4, 93
- last_trace(last_error), 71
- lgl(vector-construction), 106
- list2(tidy-dots), 100
- ll(vector-construction), 106
- locked, 42

- maybe_missing(missing_arg), 71
- methods::as(), 81
- missing argument, 86
- missing_arg, 71
- missing_arg(), 101, 110

- names2, 73
- new_box(box), 18
- new_box(), 7
- new_data_mask(as_data_mask), 7
- new_data_mask(), 45, 46
- new_environment(env), 30
- new_formula, 74
- new_function, 74
- new_quosure(as_quosure), 15
- new_quosure(), 74, 86
- new_quosures, 75
- new_weakref, 76

- new_weakref(), 110
- node_cdr(), 62

- op-get-attr, 77
- op-na-default, 78
- op-null-default, 78, 78

- package environments, 39
- pairlist2, 79
- pairlist2(), 74
- parse_expr, 79
- parse_expr(), 61, 62
- parse_exprs(parse_expr), 79
- parse_quo(parse_expr), 79
- parse_quos(parse_expr), 79
- peek_option(scoped_options), 96
- peek_options(scoped_options), 96
- pkg_env(), 11
- Pronouns, 45
- push_options(scoped_options), 96

- qq_show(quasiquotation), 81
- quasiquotation, 12, 46, 81, 89
- quo(quotation), 86
- quo(), 15, 49, 85, 86
- quo_get_env(quosure), 84
- quo_get_env(), 56
- quo_get_expr(quosure), 84
- quo_get_expr(), 46, 98
- quo_is_call(quosure), 84
- quo_is_missing(quosure), 84
- quo_is_null(quosure), 84
- quo_is_symbol(quosure), 84
- quo_is_symbolic(quosure), 84
- quo_label, 90
- quo_label(), 92
- quo_name(quo_label), 90
- quo_name(), 48, 88, 92
- quo_set_env(quosure), 84
- quo_set_env(), 56
- quo_set_expr(quosure), 84
- quo_set_expr(), 98
- quo_squash, 92
- quo_text(quo_label), 90
- quos(quotation), 86
- quos(), 86
- quos_auto_name(exprs_auto_name), 48
- quos_auto_name(), 75, 88
- quosure, 14, 84, 88
- Quosures, 45
- quosures, 8, 51, 80
- quotation, 86
- quote(), 88

- quoted expression, [84](#)
- quoting, [19](#)
- rep_along, [92](#)
- rep_named(rep_along), [92](#)
- return_from(), [108](#)
- rlang backtrace, [29](#)
- rlang::last_error(), [93](#)
- rlang_backtrace_on_error, [4](#), [29](#), [93](#)
- scalar-type-predicates, [18](#), [94](#), [105](#)
- scoped_bindings, [95](#)
- scoped_interactive(is_interactive), [67](#)
- scoped_options, [96](#)
- search_envs(), [29](#)
- seq2, [97](#)
- seq2_along(seq2), [97](#)
- seq_along(), [92](#)
- set_env(get_env), [56](#)
- set_env(), [56](#)
- set_expr, [98](#)
- set_expr(), [98](#)
- set_names, [99](#)
- signal(abort), [3](#)
- signal(), [28](#)
- squashed, [13](#)
- stats::setNames(), [99](#)
- sym, [100](#)
- sym(), [59](#), [87](#)
- symbolic, [19](#)
- symbolic expressions, [85](#)
- symbols, [14](#), [16](#)
- syms(sym), [100](#)
- syms(), [87](#)
- tidy dots, [19](#), [20](#), [23](#), [31](#), [33](#), [79](#), [106](#), [108](#)
- tidy-dots, [47](#), [100](#)
- tidyeval-data, [102](#)
- trace_back, [103](#)
- trace_back(), [4](#)
- trace_length(trace_back), [103](#)
- tryCatch(), [107](#)
- type-predicates, [18](#), [95](#), [104](#)
- unbox(box), [18](#)
- uncopyable, [31](#), [56](#)
- unquote, [100](#)
- unquote-splice, [100](#)
- unquoted, [103](#)
- unquoting operators, [89](#)
- UQ (quasiquote), [81](#)
- UQS (quasiquote), [81](#)
- vector-coercion, [106](#)
- vector-construction, [106](#)
- warn(abort), [3](#)
- warn(), [27](#), [28](#)
- with_abort, [107](#)
- with_abort(), [5](#), [29](#), [30](#)
- with_bindings(scoped_bindings), [95](#)
- with_handlers, [108](#)
- with_handlers(), [28](#), [108](#)
- with_interactive(is_interactive), [67](#)
- with_options(scoped_options), [96](#)
- wref_key, [109](#)
- wref_key(), [76](#)
- wref_value(wref_key), [109](#)
- wref_value(), [76](#)
- zap, [110](#)
- zap(), [23](#), [33](#)