Package 'rlang'

March 28, 2019

Version 0.3.3

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 (>= 3.1.0)

Suggests 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` *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

`message`	The message to display.
`.subclass`	Subclass of the condition. This allows your users to selectively handle the conditions signalled by your functions.
`...`	Additional data to be stored in the condition object.
`trace`	A `trace` object created by `trace_back()`.

`call`	Deprecated as of rlang 0.3.0. Storing the full backtrace is now preferred to storing a simple call.
`parent`	A parent condition object created by `abort()`.
`msg, type`	These arguments were renamed to `message` and `.type` and are deprecated as of rlang 0.3.0.

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 (>= 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(err$message) # 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())

}
```

---

are_na *Test for missing values*

Description

`are_na()` checks for missing values in a vector and is equivalent to `base::is.na()`. It is a vectorised predicate, meaning that its output is always the same length as its input. On the other hand, `is_na()` is a scalar predicate and always returns a scalar boolean, `TRUE` or `FALSE`. If its input is not scalar, it returns `FALSE`. Finally, there are typed versions that check for particular missing types.

Usage

```
are_na(x)

is_na(x)

is_lgl_na(x)

is_int_na(x)

is_dbl_na(x)

is_chr_na(x)

is_cpl_na(x)
```

Arguments

x	An object to test

Details

The scalar predicates accept non-vector inputs. They are equivalent to `is_null()` in that respect. In contrast the vectorised predicate `are_na()` requires a vector input since it is defined over vector values.

Examples

```
# are_na() is vectorised and works regardless of the type
are_na(c(1, 2, NA))
are_na(c(1L, NA, 3L))

# is_na() checks for scalar input and works for all types
is_na(NA)
is_na(na_dbl)
is_na(character(0))

# There are typed versions as well:
is_lgl_na(NA)
is_lgl_na(na_dbl)
```

---

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

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

`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`	Soft-deprecated. This argument no longer has any effect. The parent of the data mask is determined from either: • The `env` argument of `eval_tidy()` • 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.
`top`	The last environment of the data mask. If the data mask is only one environment deep, `top` should be the same as `bottom`.

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

rlang 0.3.0

Passing environments to `as_data_mask()` is soft-deprecated. Please build a data mask with `new_data_mask()`.

The `parent` argument no longer has any effect. The parent of the data mask is determined from either:

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

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 soft-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` *Coerce to an environment*

Description

`as_environment()` coerces named vectors (including lists) to an environment. It first checks that `x` is a dictionary (see `is_dictionaryish()`). If supplied an unnamed string, it returns the corresponding package environment (see `pkg_env()`).

Usage

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

Arguments

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

Details

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

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` *Convert to function or closure*

Description

Stable

- `as_function()` transform objects to functions. It fetches functions by name if supplied a string or transforms formulas to function.
- `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).

Usage

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

is_lambda(x)

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

Arguments

`x`	A function or formula. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function with two arguments, `.x` or `.` and `.y`. This allows you to create very compact anonymous functions with up to two inputs. Functions created from formulas have a special class. Use `is_lambda()` to test for it.
`env`	Environment in which to fetch the function in case `x` is a string.

Examples

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

# 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(`~`)
```

`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 `enquo()` 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(foobar))

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

---

as_name *Extract names from symbols*

Description

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

---

as_utf8_character *Coerce to a character vector and attempt encoding conversion*

Description

Unlike specifying the `encoding` argument in `as_string()` and `as_character()`, which is only declarative, these functions actually attempt to convert the encoding of their input. There are two possible cases:

- The string is tagged as UTF-8 or latin1, the only two encodings for which R has specific support. In this case, converting to the same encoding is a no-op, and converting to native always works as expected, as long as the native encoding, the one specified by the `LC_CTYPE` locale (see `mut_utf8_locale()`) has support for all characters occurring in the strings. Unrepresentable characters are serialised as unicode points: "<U+xxxx>".
- The string is not tagged. R assumes that it is encoded in the native encoding. Conversion to native is a no-op, and conversion to UTF-8 should work as long as the string is actually encoded in the locale codeset.

When translating to UTF-8, the strings are parsed for serialised unicode points (e.g. strings looking like "U+xxxx") with `chr_unserialise_unicode()`. This helps to alleviate the effects of character-to-symbol-to-character roundtrips on systems with non-UTF-8 native encoding.

Usage

```
as_utf8_character(x)

as_native_character(x)

as_utf8_string(x)

as_native_string(x)
```

Arguments

x	An object to coerce.

Examples

```
# Let's create a string marked as UTF-8 (which is guaranteed by the
# Unicode escaping in the string):
utf8 <- "caf\uE9"
str_encoding(utf8)
as_bytes(utf8)

# It can then be converted to a native encoding, that is, the
# encoding specified in the current locale:
## Not run:
mut_latin1_locale()
latin1 <- as_native_string(utf8)
str_encoding(latin1)
as_bytes(latin1)

## End(Not run)
```

`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	Expected encoding of a string or character vector. One of UTF-8, latin1, or unknown.

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

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

Examples

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

unbox(boxed)

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

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

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

---

call2 *Create a call*

Description

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

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

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

Usage

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

Arguments

`.fn`	Function to call. Must be a callable object: a string, symbol, call, or a function.
`...`	Arguments to the call either in or out of a list. These dots support tidy dots features.
`.ns`	Namespace with which to prefix `.fn`. 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")
```

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()
```

caller_fn *Get properties of the current or caller frame*

Description

Experimental

- The current frame is the execution context of the function that is currently being evaluated.
- The caller frame is the execution context of the function that called the function currently being evaluated.

See the call stack topic for more information.

Usage

```
caller_fn(n = 1)

current_fn()
```

Arguments

n	The number of generations to go back.

Life cycle

These functions are experimental.

See Also

`caller_env()` and `current_env()`

---

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 soft-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 soft-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 *Inspect a call*

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 *Modify the arguments of a call*

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 `zap()` to remove arguments. These dots support tidy dots 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 `call_standardise()` manually.

Value

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

Life cycle

- Prior to rlang 0.3.0, `NULL` was the sentinel for removing arguments. As of 0.3.0, `zap()` objects remove arguments and `NULL` simply adds an argument set to `NULL`. This breaking change allows the deletion sentinel to be distinct from valid argument values.
- The `.standardise` argument is soft-deprecated as of rlang 0.3.0.
- In rlang 0.2.0, `lang_modify()` was soft-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` *Extract function name or namespaced of a call*

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 soft-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 subsetted functions return NULL:
call_name(quote(foo$bar()))
call_name(quote(foo[[bar]]()))
call_name(quote(foo()()))

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

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

`call_standardise` *Standardise a call*

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 soft-deprecated and renamed to `call_standardise()`. See lifecycle section in `call2()` for more about this change.

catch_cnd *Catch a condition*

Description

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

Usage

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

Arguments

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

Value

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

Examples

```
catch_cnd(10)
catch_cnd(abort("an error"))
catch_cnd(cnd_signal("my_condition", .msg = "a condition"))
```

cnd *Create a condition object*

Description

These constructors make it easy to create subclassed conditions. Conditions are objects that power the error system in R. They can also be used for passing messages to pre-established handlers.

Usage

```
cnd(.subclass, ..., message = "")

error_cnd(.subclass = NULL, ..., message = "", trace = NULL,
  parent = NULL)

warning_cnd(.subclass = NULL, ..., message = "")

message_cnd(.subclass = NULL, ..., message = "")
```

Arguments

.subclass	The condition subclass.
...	Named data fields stored inside the condition object. These dots are evaluated with explicit splicing.
message	A default message to inform the user about the condition when it is signalled.
trace	A trace object created by trace_back().
parent	A parent condition object created by abort().

Details

cnd() creates objects inheriting from condition. Conditions created with error_cnd(), warning_cnd() and message_cnd() inherit from error, warning or message.

Lifecycle

The .type and .msg arguments have been renamed to .subclass and message. They are defunct as of rlang 0.3.0.

See Also

cnd_signal(), with_handlers().

Examples

```
# Create a condition inheriting from the s3 type "foo":
cnd <- cnd("foo")

# Signal the condition to potential handlers. Since this is a bare
# condition the signal has no effect if no handlers are set up:
cnd_signal(cnd)

# When a relevant handler is set up, the signal causes the handler
# to be called:
with_handlers(cnd_signal(cnd), foo = exiting(function(c) "caught!"))

# Handlers can be thrown or executed inplace. See with_handlers()
# documentation for more on this.

# Signalling an error condition aborts the current computation:
err <- error_cnd("foo", message = "I am an error")
try(cnd_signal(err))
```

---

cnd_muffle *Muffle a condition*

Description

Unlike exiting() handlers, calling() handlers must be explicit that they have handled a condition to stop it from propagating to other handlers. Use cnd_muffle() within a calling handler (or as a calling handler, see examples) to prevent any other handlers from being called for that condition.

Usage

```
cnd_muffle(cnd)
```

Arguments

cnd	A condition to muffle.

Mufflable conditions

Most conditions signalled by base R are muffable, although the name of the restart varies. cnd_muffle() will automatically call the correct restart for you. It is compatible with the following conditions:

- `warning` and `message` conditions. In this case `cnd_muffle()` is equivalent to `base::suppressMessages()` and `base::suppressWarnings()`.
- Bare conditions signalled with `signal()` or `cnd_signal()`. Note that conditions signalled with `base::signalCondition()` are not mufflable.
- Interrupts are sometimes signalled with a `resume` restart on recent R versions. When this is the case, you can muffle the interrupt with `cnd_muffle()`. Check if a restart is available with `base::findRestart("resume")`.

If you call `cnd_muffle()` with a condition that is not mufflable you will cause a new error to be signalled.

- Errors are not mufflable since they are signalled in critical situations where execution cannot continue safely.
- Conditions captured with `base::tryCatch()`, `with_handlers()` or `catch_cnd()` are no longer mufflable. Muffling restarts *must* be called from a calling handler.

Examples

```
fn <- function() {
  inform("Beware!", "my_particular_msg")
  inform("On your guard!")
  "foobar"
}

# Let's install a muffling handler for the condition thrown by `fn()`.
# This will suppress all `my_particular_wng` warnings but let other
# types of warnings go through:
with_handlers(fn(),
  my_particular_msg = calling(function(cnd) {
    inform("Dealt with this particular message")
    cnd_muffle(cnd)
  })
)

# Note how execution of `fn()` continued normally after dealing
# with that particular message.

# cnd_muffle() can also be passed to with_handlers() as a calling
# handler:
with_handlers(fn(),
  my_particular_msg = calling(cnd_muffle)
)
```

`cnd_signal` *Signal a condition*

Description

Signal a condition to handlers that have been established on the stack. Conditions signalled with `cnd_signal()` are assumed to be benign. Control flow can resume normally once the condition has been signalled (if no handler jumped somewhere else on the evaluation stack). On the other hand, `cnd_abort()` treats the condition as critical and will jump out of the distressed call frame (see `rst_abort()`), unless a handler can deal with the condition.

Usage

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

Arguments

`cnd`	A condition object (see `cnd()`).
`.cnd, .mufflable`	These arguments are retired. `.cnd` has been renamed to `cnd` and `.mufflable` no longer has any effect as non-critical conditions are always signalled with a muffling restart.

Details

If `.critical` is `FALSE`, this function has no side effects beyond calling handlers. In particular, execution will continue normally after signalling the condition (unless a handler jumped somewhere else via `rst_jump()` or by being `exiting()`). If `.critical` is `TRUE`, the condition is signalled via `base::stop()` and the program will terminate if no handler dealt with the condition by jumping out of the distressed call frame.

`calling()` handlers are called in turn when they decline to handle the condition by returning normally. However, it is sometimes useful for a calling handler to produce a side effect (signalling another condition, displaying a message, logging something, etc), prevent the condition from being passed to other handlers, and resume execution from the place where the condition was signalled. The easiest way to accomplish this is by jumping to a restart point (see `with_restarts()`) established by the signalling function. `cnd_signal()` always installs a muffle restart (see `cnd_muffle()`).

Lifecycle

- Modifying a condition object with `cnd_signal()` is defunct. Consequently the `.msg` and `.call` arguments are retired and defunct as of rlang 0.3.0. In addition `.cnd` is renamed to `cnd` and soft-deprecated.
- The `.mufflable` argument is soft-deprecated and no longer has any effect. Non-critical conditions are always signalled with a muffle restart.
- Creating a condition object with `cnd_signal()` is soft-deprecated. Please use `signal()` instead.

See Also

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

Examples

```
# Creating a condition of type "foo"
cnd <- cnd("foo")

# If no handler capable of dealing with "foo" is established on the
# stack, signalling the condition has no effect:
cnd_signal(cnd)

# To learn more about establishing condition handlers, see
# documentation for with_handlers(), exiting() and calling():
with_handlers(cnd_signal(cnd),
  foo = calling(function(c) cat("side effect!\n"))
)


# By default, cnd_signal() creates a muffling restart which allows
# calling handlers to prevent a condition from being passed on to
# other handlers and to resume execution:
undesirable_handler <- calling(function(c) cat("please don't call me\n"))
muffling_handler <- calling(function(c) {
  cat("muffling foo...\n")
  cnd_muffle(c)
})

with_handlers(foo = undesirable_handler,
  with_handlers(foo = muffling_handler, {
    cnd_signal(cnd("foo"))
    "return value"
  }))
```

---

cnd_type *What type is a condition?*

Description

Use `cnd_type()` to check what type a condition is.

Usage

```
cnd_type(cnd)
```

Arguments

cnd	A condition object.

Value

A string, either `"condition"`, `"message"`, `"warning"`, `"error"` or `"interrupt"`.

Examples

```
cnd_type(catch_cnd(abort("Abort!")))
cnd_type(catch_cnd(interrupt()))
```

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)
```

dots_n *How many arguments are currently forwarded in dots?*

Description

This returns the number of arguments currently forwarded in `...` as an integer.

Usage

```
dots_n(...)
```

Arguments

`...`	Forwarded arguments.

Examples

```
fn <- function(...) dots_n(..., baz)
fn(foo, bar)
```

dots_values *Evaluate dots with preliminary splicing*

Description

This is a tool for advanced users. It captures dots, processes unquoting and splicing operators, and evaluates them. Unlike `dots_list()`, it does not flatten spliced objects, instead they are attributed a `spliced` class (see `splice()`). You can process spliced objects manually, perhaps with a custom predicate (see `flatten_if()`).

Usage

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

Arguments

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

Examples

```
dots <- dots_values(!!! list(1, 2), 3)
dots

# Flatten the objects marked as spliced:
flatten_if(dots, is_spliced)
```

empty_env *Get the empty environment*

Description

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

Usage

```
empty_env()
```

Examples

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

entrace *Add backtrace from error handler*

Description

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.

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

Usage

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

Arguments

`cnd`	When `entrace()` is used as a calling handler, `cnd` is the condition to handle.
`...`	Unused. These dots are for future extensions.
`top`	The first frame environment to be included in the backtrace. This becomes the top of the backtrace tree and represents the oldest call in the backtrace. This is needed in particular when you call `trace_back()` indirectly or from a larger context, for example in tests or inside an RMarkdown document where you don't want all of the knitr evaluation mechanisms to appear in the backtrace.
`bottom`	The last frame environment to be included in the backtrace. This becomes the rightmost leaf of the backtrace tree and represents the youngest call in the backtrace. Set this when you would like to capture a backtrace without the capture context. Can also be an integer that will be passed to `caller_env()`.

See Also

`with_abort()` to promote conditions to rlang errors.

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

`.env`	An environment.
`...`	Pairs of names and expressions, values or functions. Pass `zap()` objects to remove bindings. These dots support tidy dots features.
`.eval_env`	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 <- "fievel"
  quo(paste(who, "mouse"))
})
env_bind_lazy(env, name = !!quo)
env$name

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

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

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

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

---

env_bury *Mask bindings by defining symbols deeper in a scope*

Description

env_bury() is like env_bind() but it creates the bindings in a new child environment. This makes

sure the new bindings have precedence over old ones, without altering existing environments. Unlike `env_bind()`, this function does not have side effects and returns a new environment (or object wrapping that environment).

Usage

```
env_bury(.env, ...)
```

Arguments

`.env`	An environment.
`...`	Pairs of names and expressions, values or functions. Pass `zap()` objects to remove bindings. These dots support tidy dots features.

Value

A copy of `.env` enclosing the new environment containing bindings to `...` arguments.

See Also

`env_bind()`, `env_unbind()`

Examples

```
orig_env <- env(a = 10)
fn <- set_env(function() a, orig_env)

# fn() currently sees `a` as the value `10`:
fn()

# env_bury() will bury the current scope of fn() behind a new
# environment:
fn <- env_bury(fn, a = 1000)
fn()

# Even though the symbol `a` is still defined deeper in the scope:
orig_env$a
```

---

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

Usage

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

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 *Does an environment have or see bindings?*

Description

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

Usage

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

Arguments

env	An environment.
nms	A character vector 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` *Does environment inherit from another environment?*

Description

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

Usage

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

Arguments

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

`env_lock` *Lock an environment*

Description

Experimental

Locked environments cannot be modified. An important example is namespace environments which are locked by R when loaded in a session. Once an environment is locked it normally cannot be unlocked.

Note that only the environment as a container is locked, not the individual bindings. You can't remove or add a binding but you can still modify the values of existing bindings. See `env_binding_lock()` for locking individual bindings.

Usage

```
env_lock(env)

env_is_locked(env)
```

Arguments

`env`	An environment.

Value

The old value of `env_is_locked()` invisibly.

See Also

env_binding_lock()

Examples

```
# New environments are unlocked by default:
env <- env(a = 1)
env_is_locked(env)

# Use env_lock() to lock them:
env_lock(env)
env_is_locked(env)

# Now that `env` is locked, it is no longer possible to remove or
# add bindings. If run, the following would fail:
# env_unbind(env, "a")
# env_bind(env, b = 2)

# Note that even though the environment as a container is locked,
# the individual bindings are still unlocked and can be modified:
env$a <- 10
```

---

env_name *Label of an environment*

Description

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

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

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

Usage

```
env_name(env)

env_label(env)
```

Arguments

env	An environment.

Examples

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

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

---

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

Description

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

Usage

```
env_names(env)

env_length(env)
```

Arguments

`env`	An environment.

Value

A character vector of object names.

Names of symbols and objects

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

Encoding

There are deep encoding issues when you convert a string to symbol and vice versa. Symbols are *always* in the native encoding (see `set_chr_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 unicode points unserialised.

Examples

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

env_parent *Get parent environments*

Description

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

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

Usage

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

env_tail(env = caller_env(), last = global_env(), 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 soft-deprecated, 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`.

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` *Pretty-print an environment*

Description

This prints:

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

Usage

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

Arguments

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

`env_unbind` *Remove bindings from an environment*

Description

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

Usage

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

Arguments

`env`	An environment.
`nms`	A character vector 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 enviroment of a frame on the call stack.

Usage

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

Arguments

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

Details

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

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

- `return()` calls are evaluated in frame environments that might be burried deep in the call stack. This causes a long return that unwinds multiple frames (triggering the `on.exit()` event for each frame). By contrast `eval()` only returns from the `eval()` call, one level up.

- `on.exit()`, `parent.frame()`, `sys.call()`, and generally all the stack inspection functions `sys.xxx()` are evaluated in the correct frame environment. This is similar to how this type of calls can be evaluated deep in the call stack because of lazy evaluation, when you force an argument that has been passed around several times.

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

See Also

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

Examples

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

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

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

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

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

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

```
fn(base::eval)
```

`eval_tidy` *Evaluate an expression with quosures and pronoun support*

Description

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 `as_data_mask()` or `new_data_mask()`. Objects in `data` have priority over those in `env`. See the section about data masking.
`env`	The environment in which to evaluate `expr`. This environment is not applicable for quosures because they have their own environments.

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

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

---

exiting *Create an exiting or in place handler*

Description

There are two types of condition handlers: exiting handlers, which are thrown to the place where they have been established (e.g., `with_handlers()`'s evaluation frame), and local handlers, which are executed in place (e.g., where the condition has been signalled). `exiting()` and `calling()` create handlers suitable for `with_handlers()`.

Usage

```
exiting(handler)

calling(handler)
```

Arguments

handler	A handler function that takes a condition as argument. This is passed to `as_function()` and can thus be a formula describing a lambda function.

Details

A subtle point in the R language is that conditions are not thrown, handlers are. `base::tryCatch()` and `with_handlers()` actually catch handlers rather than conditions. When a critical condition is signalled with `base::stop()` or `abort()`, R inspects the handler stack and looks for a handler that can deal with the condition. If it finds an exiting handler, it throws it to the function that

established it (with_handlers()). That is, it interrupts the normal course of evaluation and jumps to with_handlers() evaluation frame (see ctxt_stack()), and only then and there the handler is called. On the other hand, if R finds a calling handler, it executes it locally. The calling handler can choose to handle the condition by jumping out of the frame (see rst_jump() or return_from()). If it returns locally, it declines to handle the condition which is passed to the next relevant handler on the stack. If no handler is found or is able to deal with the critical condition (by jumping out of the frame), R will then jump out of the faulty evaluation frame to top-level, via the abort restart (see rst_abort()).

Life cycle

exiting() is in the questioning stage because with_handlers() now treats handlers as exiting by default.

See Also

with_handlers() for examples, restarting() for another kind of calling handler.

Examples

```
# You can supply a function taking a condition as argument:
hnd <- exiting(function(c) cat("handled foo\n"))
with_handlers(signal("A foobar condition occurred", "foo"), foo = hnd)

# Or a lambda-formula where "." is bound to the condition:
with_handlers(foo = calling(~cat("hello", .$attr, "\n")), {
  signal("A foobar condition occurred", "foo", attr = "there")
  "foo"
})
```

---

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

exprs	A list of expressions.
width	Soft-deprecated. Maximum width of names.
printer	Soft-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.
quos	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

`x`	A function, raw expression, or formula to interpolate.
`env`	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` *Turn an expression to a label*

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` *Print an expression*

Description

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

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

Usage

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

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

Arguments

`x`	An object or expression to print.
`width`	The width of the deparsed or printed expression. Defaults to the global option `width`.

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 *Get or set function body*

Description

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

Usage

```
fn_body(fn = caller_fn())

fn_body(fn) <- value
```

Arguments

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

Examples

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

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

---

fn_env *Return the closure environment of a function*

Description

Closure environments define the scope of functions (see `env()`). When a function call is evaluated, R creates an evaluation frame (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 lookep up in the calling frame if not supplied.
value	New formals or formals names for fn.

Details

Unlike formals(), these helpers also work with primitive functions. See is_function() for a discussion of primitive and closure functions.

Note that the argument names are taken from the closures that are created when passing the primitive to as_closure(). For instance, while the arguments of the primitive operator + are labelled e1 and e2, fn_fmls_names() will return .x and .y. Note that for many primitives the base R argument names are purely placeholders since they don't perform regular argument matching. E.g. this returns 5 instead of -5:

```
`-`(e2 = 10, 5)
```

To regularise the semantics of primitive functions, it is usually a good idea to coerce them to a closure first:

```
minus <- as_closure(`-`)
minus(.y = 10, 5)
```

See Also

call_args() and call_args_names()

Examples

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

# Works with primitive functions:
fn_fmls(base::switch)

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

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

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

---

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

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

Details

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

Life cycle

- Using `get_env()` without supplying `env` is soft-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 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.

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_length *How long is an object?*

Description

This is a function for the common task of testing the length of an object. It checks the length of an object in a non-generic way: `base::length()` methods are ignored.

Usage

```
has_length(x, n = NULL)
```

Arguments

x	A R object.
n	A specific length to test x with. If NULL, has_length() returns TRUE if x has length greater than zero, and FALSE otherwise.

Examples

```
has_length(list())
has_length(list(), 0)

has_length(letters)
has_length(letters, 20)
has_length(letters, 26)
```

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.

Usage

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

Arguments

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

Details

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

Value

A logical vector of the same length as name

Examples

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

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

Description

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

Usage

```
inherits_any(x, class)

inherits_all(x, class)

inherits_only(x, class)
```

Arguments

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

Examples

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

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

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

# The order of classes must match as well:
inherits_all(obj, c("baz", "foo"))

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

`is_call` *Is object a call?*

Description

This function tests if `x` is a call. This is a pattern-matching predicate that returns `FALSE` if `name` and `n` are supplied and the call does not match these properties. `is_unary_call()` and `is_binary_call()` hardcode `n` to 1 and 2.

Usage

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

Arguments

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

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_callable` *Is an object callable?*

Description

A callable object is an object that can appear in the function position of a call (as opposed to argument position). This includes symbolic objects that evaluate to a function or literal functions embedded in the call.

Usage

```
is_callable(x)
```

Arguments

x	An object to test.

Details

Note that strings may look like callable objects because expressions of the form `"list"()` are valid R code. However, that's only because the R parser transforms strings to symbols. It is not legal to manually set language heads to strings.

Examples

```
# Symbolic objects and functions are callable:
is_callable(quote(foo))
is_callable(base::identity)

# node_poke_car() lets you modify calls without any checking:
lang <- quote(foo(10))
node_poke_car(lang, current_env())

# Use is_callable() to check an input object is safe to put as CAR:
obj <- base::identity

if (is_callable(obj)) {
  lang <- node_poke_car(lang, obj)
} else {
  abort("`obj` must be callable")
}

eval_bare(lang)
```

---

is_condition *Is object a condition?*

Description

Is object a condition?

Usage

```
is_condition(x)
```

Arguments

x	An object to test.

---

is_copyable *Is an object copyable?*

Description

When an object is modified, R generally copies it (sometimes lazily) to enforce value semantics. However, some internal types are uncopyable. If you try to copy them, either with <- or by argument passing, you actually create references to the original object rather than actual copies. Modifying these references can thus have far reaching side effects.

Usage

```
is_copyable(x)
```

Arguments

x	An object to test.

Examples

```
# Let's add attributes with structure() to uncopyable types. Since
# they are not copied, the attributes are changed in place:
env <- env()
structure(env, foo = "bar")
env

# These objects that can only be changed with side effect are not
# copyable:
is_copyable(env)

structure(base::list, foo = "bar")
str(base::list)
```

---

is_empty *Is object an empty vector or NULL?*

Description

Is object an empty vector or NULL?

Usage

```
is_empty(x)
```

Arguments

x	object to test

Examples

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

---

is_environment *Is object an environment?*

Description

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

Usage

```
is_environment(x)

is_bare_environment(x)
```

Arguments

x	object to test

`is_expression` *Is an object an expression?*

Description

`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 *Is object a formula?*

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` *Is object a function?*

Description

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

Usage

```
is_function(x)

is_closure(x)

is_primitive(x)

is_primitive_eager(x)

is_primitive_lazy(x)
```

Arguments

x	Object to be tested.

Details

Closures are functions written in R, named after the way their arguments are scoped within nested environments (see https://en.wikipedia.org/wiki/Closure_(computer_programming)). The root environment of the closure is called the closure environment. When closures are evaluated, a new environment called the evaluation frame is created with the closure environment as parent. This is where the body of the closure is evaluated. These closure frames appear on the evaluation stack (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. See `fn_fmls()` for a function that returns a representation of formal arguments for primitive functions. 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 *Is a package installed in the library?*

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 *Is a vector integer-like?*

Description

These predicates check whether R considers a number vector to be integer-like, according to its own tolerance check (which is in fact delegated to the C library). This function is not adapted to data analysis, see the help for base::is.integer() for examples of how to check for whole numbers.

Things to consider when checking for integer-like doubles:

- This check can be expensive because the whole double vector has to be traversed and checked.
- Large double values may be integerish but may still not be coercible to integer. This is because integers in R only support values up to 2^31 - 1 while numbers stored as double can be much larger.

Usage

```
is_integerish(x, n = NULL, finite = NULL)

is_bare_integerish(x, n = NULL, finite = NULL)

is_scalar_integerish(x, finite = NULL)
```

Arguments

x	Object to be tested.
n	Expected length of a vector.
finite	Whether all values of the vector are finite. The non-finite values are NA, Inf, -Inf and NaN. Setting this to something other than NULL can be expensive because the whole vector needs to be traversed and checked.

See Also

is_bare_numeric() for testing whether an object is a base numeric type (a bare double or integer vector).

Examples

```
is_integerish(10L)
is_integerish(10.0)
is_integerish(10.0, n = 2)
is_integerish(10.000001)
is_integerish(TRUE)
```

is_interactive *Is R running interactively?*

Description

Like base::interactive(), is_interactive() returns TRUE when the function runs interactively and FALSE when it runs in batch mode. It also checks:

- Whether knitr or an RStudio notebook is in progress.
- The rlang_interactive global option. If set to a single TRUE or FALSE, is_interactive() returns that value instead. This escape hatch is useful in unit tests or to manually turn on interactive features in RMarkdown outputs.

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 is_interactive().
frame	The environment of a running function which defines the scope of the temporary options. When the function returns, the options are reset to their original values.
expr	An expression to evaluate with interactivity set to value.

is_named *Is object named?*

Description

is_named() 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 *Is an object a namespace environment?*

Description

Is an object a namespace environment?

Usage

```
is_namespace(x)
```

Arguments

x	An object to test.

`is_reference` *Is an object referencing another?*

Description

There are typically two situations where two symbols may refer to the same object.

- R objects usually have copy-on-write semantics. This is an optimisation that ensures that objects are only copied if needed. When you copy a vector, no memory is actually copied until you modify either the original object or the copy is modified.

  Note that the copy-on-write optimisation is an implementation detail that is not guaranteed by the specification of the R language.

- Assigning an uncopyable object (like an environment) creates a reference. These objects are never copied even if you modify one of the references.

Usage

```
is_reference(x, y)
```

Arguments

x, y	R objects.

Examples

```
# Reassigning an uncopyable object such as an environment creates a
# reference:
env <- env()
ref <- env
is_reference(ref, env)

# Due to copy-on-write optimisation, a copied vector can
# temporarily reference the original vector:
vec <- 1:10
copy <- vec
is_reference(copy, vec)

# Once you modify on of them, the copy is triggered in the
# background and the objects cease to reference each other:
vec[[1]] <- 100
is_reference(copy, vec)
```

is_stack *Is object a stack?*

Description

Soft-deprecated

Usage

```
is_stack(x)

is_eval_stack(x)

is_call_stack(x)
```

Arguments

x	An object to test

is_symbol *Is object a symbol?*

Description

Is object a symbol?

Usage

```
is_symbol(x, name = NULL)
```

Arguments

x	An object to test.
name	An optional name or vector of names that the symbol should match.

is_true *Is object identical to TRUE or FALSE?*

Description

These functions bypass R's automatic conversion rules and check that `x` is literally `TRUE` or `FALSE`.

Usage

```
is_true(x)

is_false(x)
```

Arguments

x	object to test

Examples

```
is_true(TRUE)
is_true(1)

is_false(FALSE)
is_false(0)
```

---

lang_head *Return the head or tail of a call*

Description

Soft-deprecated

As of rlang 0.2.0 these functions are retired (soft-deprecated for now) because they are low level accessors that are rarely needed for end users.

Usage

```
lang_head(lang)

lang_tail(lang)
```

Arguments

lang	A call.

---

last_error *Last* abort() *error*

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()
```

`lifecycle` *Life cycle of the rlang package*

Description

Maturing

The rlang package is currently maturing. Unless otherwise stated, this applies to all its exported functions. Maturing functions are susceptible to API changes. Only use these in packages if you're prepared to make changes as the package evolves. See sections below for a list of functions marked as stable.

The documentation pages of retired functions contain life cycle sections that explain the reasons for their retirements.

Stable functions

Stable

- `eval_tidy()`
- `!!`, `!!!`
- `enquo()`, `quo()`, `quos()`
- `enexpr()`, `expr()`, `exprs()`
- `sym()`, `syms()`
- `new_quosure()`, `is_quosure()`
- `missing_arg()`, `is_missing()`
- `quo_get_expr()`, `quo_set_expr()`
- `quo_get_env()`, `quo_set_env()`
- `eval_bare()`
- `set_names()`, `names2()`
- `as_function()`, `new_function()`

Experimental functions

Experimental

These functions are not yet part of the rlang API. Expect breaking changes.

- `with_env()`, `locally()`
- `env_poke()`
- `env_bind_fns()`, `env_bind_exprs()`
- `pkg_env()`, `pkg_env_name()`
- `scoped_env()`, `scoped_names()`, `scoped_envs()`, `is_scoped()`
- `ns_env()`, `ns_imports_env()`, `ns_env_name()`
- `is_pairlist()`, `as_pairlist()`, `is_node()`, `is_node_list()`
- `is_definition()`, `new_definition()`, `is_formulaish()`, `dots_definitions()`
- `scoped_options()`, `with_options()`, `push_options()`, `peek_options()`, `peek_option()`
- `as_bytes()`, `chr_unserialise_unicode()`, `set_chr_encoding()`, `chr_encoding()`, `set_str_encoding()`, `str_encoding()`
- `mut_utf8_locale()`, `mut_latin1_locale()`, `mut_mbcs_locale()`
- `caller_fn()`, `current_fn()`

Questioning stage

Questioning

In the questioning stage as of rlang 0.3.0

- child_env()
- type_of(), switch_type(), coerce_type()
- switch_class(), coerce_class()
- lang_type_of(), switch_lang(), coerce_lang()
- flatten(), squash(), and their atomic vector variants
- modify() and prepend()
- as_logical(), as_character(), etc.
- with_restarts(), rst_list(), rst_exists(), rst_jump(), rst_maybe_jump(), rst_abort(). It is not clear yet whether we want to recommend restarts as a style of programming in R.
- return_from() and return_to().
- expr_label(), expr_name(), and expr_text().

In the questioning stage as of rlang 0.2.0

- UQ(), UQS()
- dots_splice(), splice()

Soft-deprecated functions and arguments

Soft-deprecated

Soft-deprecated as of rlang 0.3.0

- get_env(): The env argument no longer has a default and must be supplied
- cnd_signal(): The .mufflable argument no longer has any effect
- invoke(): Use the simpler exec() instead.
- set_attrs(), mut_attrs()
- cnd_signal(): .cnd => cnd
- is_frame(), global_frame(), current_frame(), ctxt_frame(), call_frame(), frame_position(), caller_frame()
- ctxt_depth(), call_depth(), ctxt_stack(), call_stack(), stack_trim()
- Passing a function or formula to env_depth(), env_poke_parent(), env_parent<-, env_tail(), set_env(), env_clone(), env_inherits(), env_bind(), scoped_bindings(), with_bindings(), env_poke(), env_has(), env_get(), env_names(), env_bind_exprs() and env_bind_fns(). This internal genericity was causing confusion (see issue #427). You should now extract the environment separately before calling these functions.
- env_bind_exprs() => env_bind_lazy()
- env_bind_fns() => env_bind_active()
- scoped_names() => base::search()
- is_scoped() => is_attached()
- scoped_env() => search_env()
- scoped_envs() => search_envs()

- The `width` and `printer` arguments of `exprs_auto_name()` and `quos_auto_name()` no longer have any effect. For the same reason, passing a width as `.named` argument of dots collectors like `quos()` is soft-deprecated.
- `call_modify()`: `.standardise` and `.env` arguments.
- `new_logical_along()`, `new_integer_along()`, `new_double_along()`, `new_complex_along()`, `new_character_along()`, `new_raw_along()`, `new_list_along()`.
- `as.character()` on quosures.
- Assigning non-quosure objects to quosure lists.
- Supplying a named `!!!` call.

Soft-deprecated as of rlang 0.2.0:

- `overscope_clean()`
- `overscope_eval_next()` => `eval_tidy()`
- `lang_head()`, `lang_tail()`
- `quo_expr()` => `quo_squash()`
- `parse_quosure()` => `parse_quo()`
- `parse_quosures()` => `parse_quos()`
- `as_overscope()` => `as_data_mask()`
- `new_overscope()` => `new_data_mask()`
- `lang()` => `call2()`
- `new_language()` => `new_call()`
- `is_lang()` => `is_call()`
- `is_unary_lang()` => Use the `n` argument of `is_call()`
- `is_binary_lang()` => Use the `n` argument of `is_call()`
- `quo_is_lang()` => `quo_is_call()`
- `is_expr()` => `is_expression()`
- `lang_modify()` => `call_modify()`
- `lang_standardise()` => `call_standardise()`
- `lang_fn()` => `call_fn()`
- `lang_name()` => `call_name()`
- `lang_args()` => `call_args()`
- `lang_args_names()` => `call_args_names()`

Deprecated functions and arguments

Deprecated

Deprecated as of rlang 0.3.0

- `as_data_mask()`: `parent` argument
- `new_data_mask()`: `parent` argument
- `env_tail()`: `sentinel` => `last`
- `abort()`, `warn()`, `inform()`: `msg`, `type` => `.msg`, `.type`
- `abort()`, `warn()`, `inform()`, `cnd()`, `error_cnd()`, `warning_cnd()`, `message_cnd()`: `call` argument.
- `length()` and `names()` on tidy eval `.data` pronouns.
- `is_character()`, `is_string()`, and variants: The `encoding` argument.

Defunct functions and arguments

Defunct

Defunct as of rlang 0.3.0:

- UQE()
- eval_tidy_()
- is_quosureish(), as_quosureish()
- as_dictionary() => as_data_pronoun()
- cnd_signal(): .msg and .call.
- cnd(), error_cnd(), warning_cnd() and message_cnd(): .msg => message.

Archived

Archived

These functions were entirely removed from the package. You will find them in the commit history and previous releases.

Archived in rlang 0.3.0:

- cnd_inform(), cnd_warn() and cnd_abort()
- new_cnd() => cnd()
- cnd_message() => message_cnd()
- cnd_warning() => warning_cnd()
- cnd_error() => error_cnd()
- rst_muffle() => cnd_muffle()
- inplace() => calling(). The muffle argument of inplace() has not been implemented in calling() and is now defunct.

---

missing *Missing values*

Description

Missing values are represented in R with the general symbol NA. They can be inserted in almost all data containers: all atomic vectors except raw vectors can contain missing values. To achieve this, R automatically converts the general NA symbol to a typed missing value appropriate for the target vector. The objects provided here are aliases for those typed NA objects.

Usage

```
na_lgl

na_int

na_dbl

na_chr

na_cpl
```

Format

An object of class `logical` of length 1.

Details

Typed missing values are necessary because R needs sentinel values of the same type (i.e. the same machine representation of the data) as the containers into which they are inserted. The official typed missing values are `NA_integer_`, `NA_real_`, `NA_character_` and `NA_complex_`. The missing value for logical vectors is simply the default `NA`. The aliases provided in rlang are consistently named and thus simpler to remember. Also, `na_lgl` is provided as an alias to `NA` that makes intent clearer.

Since `na_lgl` is the default `NA`, expressions such as `c(NA, NA)` yield logical vectors as no data is available to give a clue of the target type. In the same way, since lists and environments can contain any types, expressions like `list(NA)` store a logical `NA`.

Examples

```
typeof(NA)
typeof(na_lgl)
typeof(na_int)

# Note that while the base R missing symbols cannot be overwritten,
# that's not the case for rlang's aliases:
na_dbl <- NA
typeof(na_dbl)
```

`missing_arg` *Generate or handle a missing argument*

Description

These functions help using the missing argument as a regular R object.

- `missing_arg()` generates a missing argument.
- `is_missing()` is like `base::missing()` but also supports testing for missing arguments contained in other objects like lists.
- `maybe_missing()` is useful to pass down an input that might be missing to another function, potentially substituting by a default value. It avoids triggering an "argument is missing" error.

Usage

```
missing_arg()

is_missing(x)

maybe_missing(x, default = missing_arg())
```

Arguments

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

Other ways to reify the missing argument

- `base::quote(expr = )` is the canonical way to create a missing argument object.
- `expr()` called without argument creates a missing argument.
- `quo()` called without argument creates an empty quosure, i.e. a quosure containing the missing argument object.

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 formals lists. While C functions are always allowed to return the missing argument and pass it to other C functions, this is not the case on the R side. If you're implementing your meta-programming algorithm in R, use `maybe_missing()` when an input might be the missing argument object.

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(l[[1]])

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

`names2` *Get names of a vector*

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-vector *Create vectors matching a given length*

Description

These functions construct vectors of a given length, with attributes specified via dots. Except for `new_list()` and `new_bytes()`, the empty vectors are filled with typed missing values. This is in contrast to the base function `base::vector()` which creates zero-filled vectors.

Usage

```
new_logical(n, names = NULL)

new_integer(n, names = NULL)

new_double(n, names = NULL)

new_character(n, names = NULL)

new_complex(n, names = NULL)

new_raw(n, names = NULL)

new_list(n, names = NULL)
```

Arguments

`n`	The vector length.
`names`	Names for the new vector.

See Also

rep_along

Examples

```
new_list(10)
new_logical(10)
```

new-vector-along-retired *Create vectors matching the length of a given vector*

Description

These functions are soft-deprecated as of rlang 0.3.0 because they are longer to type than the equivalent `rep_along()` or `rep_named()` calls without added clarity.

Usage

```
new_logical_along(x, names = base::names(x))

new_integer_along(x, names = base::names(x))

new_double_along(x, names = base::names(x))

new_character_along(x, names = base::names(x))

new_complex_along(x, names = base::names(x))

new_raw_along(x, names = base::names(x))

new_list_along(x, names = base::names(x))
```

Arguments

x	A vector.
names	Names for the new vector.

---

new_formula *Create a formula*

Description

Create a formula

Usage

```
new_formula(lhs, rhs, env = caller_env())
```

Arguments

lhs, rhs	A call, name, or atomic vector.
env	An environment.

Value

A formula object.

See Also

new_quosure()

Examples

```
new_formula(quote(a), quote(b))
new_formula(NULL, quote(b))
```

new_function *Create a function*

Description

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 of default arguments. Note that if you want arguments that don't have defaults, you'll need to use the special function alist, e.g. `alist(a = , b = 1)`
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(x) x + 3
g <- new_function(alist(x = ), quote(x + 3))

# The components of the functions are identical
identical(formals(f), formals(g))
identical(body(f), body(g))
identical(environment(f), environment(g))

# But the functions are not identical because f has src code reference
identical(f, g)

attr(f, "srcref") <- NULL
# Now they are:
stopifnot(identical(f, g))
```

new_quosures *Create a list of quosures*

Description

This small S3 class provides methods for `[` and `c()` and ensures the following invariants:

- The list only contains quosures.
- It is always named, possibly with a vector of empty strings.

`new_quosures()` takes a list of quosures and adds the `quosures` class and a vector of empty names if needed. `as_quosures()` calls `as_quosure()` on all elements before creating the `quosures` object.

Usage

```
new_quosures(x)

as_quosures(x, env, named = FALSE)

is_quosures(x)
```

Arguments

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

---

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
```

Arguments

x	Object
name	Attribute name

Examples

```
# Unlike `@`, this operator extracts attributes for any kind of
# objects:
factor(1:3) %@% "levels"
mtcars %@% class

mtcars %@% class <- NULL
mtcars

# It also works on S4 objects:
.Person <- setClass("Person", slots = c(name = "character", species = "character"))
fievel <- .Person(name = "Fievel", species = "mouse")
fievel %@% name
```

op-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, y`	`y` for elements of `x` that are NA; otherwise, `x`.

See Also

op-null-default

Examples

```
c("a", "b", NA, "c") %|% "default"
```

op-null-default *Default value for* `NULL`

Description

This infix function makes it easy to replace `NULL`s 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
```

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 is one of the strings contain 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))
```

---

prim_name *Name of a primitive function*

Description

Name of a primitive function

Usage

```
prim_name(prim)
```

Arguments

prim	A primitive function such as base::c().

---

quasiquotation *Quasiquotation of an expression*

Description

Quasiquotation is the mechanism that makes it possible to program flexibly with tidy evaluation grammars like dplyr. It is enabled in all tidyeval quoting functions, the most fundamental of which are quo() and expr().

Quasiquotation 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().

Use qq_show() to experiment with quasiquotation 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 quasiquotation 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 quasiquotation. 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

```
# Quasiquotation 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:
quos(mouse1 := bernard, mouse2 = bianca)

# The `:=` is mainly useful to unquote names. Unlike `=` it
# supports `!!` on its LHS:
var <- "unquote me!"
quos(!!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, is_call() returns FALSE. If any other string, is_call() checks that x is namespaced within ns. Can be a character vector of namespaces, in which case the call has to match at least one of them, otherwise is_call() returns FALSE.
expr	A new expression for the quosure.
env	A new environment for the quosure.

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.
- `is_quosureish()` is deprecated as of rlang 0.2.0. This function assumed that quosures are formulas which is currently true but might not be in the future.

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 wether an object is a quosure:
is_quosure(quo)

# If it is a quosure, you can use the specialised type predicates
# to check what is inside it:
quo_is_symbol(quo)
quo_is_call(quo)
quo_is_null(quo)

# quo_is_missing() checks for a special kind of quosure, the one
# that contains the missing argument:
quo()
quo_is_missing(quo())

fn <- function(arg) enquo(arg)
fn()
quo_is_missing(fn())
```

quotation *Quotation*

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

`expr`	An expression.
`arg`	A symbol representing an argument. The expression supplied to that argument will be captured instead of being evaluated.
`...`	For `enexprs()`, `ensyms()` and `enquos()`, names of arguments to capture without evaluation (including `...`). For `exprs()` and `quos()`, the expressions to capture unevaluated (including expressions contained in `...`).
`.named`	Whether to ensure all dots are named. Unnamed elements are processed with `quo_name()` to build a default name. See also `quos_auto_name()`.
`.ignore_empty`	Whether to ignore empty arguments. Can be one of `"trailing"`, `"none"`, `"all"`. If `"trailing"`, only the last argument is ignored if it is empty. Note that `"trailing"` applies only to arguments passed in `...`, not to named arguments. On the other hand, `"all"` also applies to named arguments.
`.unquote_names`	Whether to treat `:=` as `=`. Unlike `=`, the `:=` syntax supports `!!` unquoting on the LHS.
`.homonyms`	How to treat arguments with the same name. The default, `"keep"`, preserves these arguments. Set `.homonyms` to `"first"` to only keep the first occurrences, to `"last"` to keep the last occurrences, and to `"error"` to raise an informative error and indicate what arguments have duplicated names.
`.check_assign`	Whether to check for `<-` calls passed in dots. When `TRUE` and a `<-` call is detected, a warning is issued to advise users to use `=` if they meant to match a function parameter, or wrap the `<-` 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 <- enquo(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 *Squash a quosure*

Description

quo_squash() flattens all nested quosures within an expression. For example it transforms ^foo(^bar(), ^baz) to the bare expression foo(bar(), baz).

This operation is safe if the squashed quosure is used for labelling or printing (see 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 quo_squash() in order to make a non-tidyeval API compatible with quosures. In that case, getting rid of the nested quosures is likely to cause subtle bugs and it is good practice to warn the user about it.

Life cycle

This function replaces quo_expr() which was soft-deprecated in rlang 0.2.0. quo_expr() was a misnomer because it implied that it was a mere expression acccessor 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 *Create vectors matching the length of a given vector*

Description

These functions take the idea of `seq_along()` and apply it to repeating values.

Usage

```
rep_along(along, x)

rep_named(names, x)
```

Arguments

along	Vector whose length determine how many times x is repeated.
x	Values to repeat.
names	Names for the new vector. The length of names determines how many times x is repeated.

See Also

new-vector

Examples

```
x <- 0:5
rep_along(x, 1:2)
rep_along(x, 1)

# Create fresh vectors by repeating missing values:
rep_along(x, na_int)
rep_along(x, na_chr)

# rep_named() repeats a value along a names vectors
rep_named(c("foo", "bar"), list(letters))
```

restarting *Create a restarting handler*

Description

This constructor automates the common task of creating an `calling()` handler that invokes a restart.

Usage

```
restarting(.restart, ..., .fields = NULL)
```

Arguments

`.restart`	The name of a restart.
`...`	Additional arguments passed on the restart function. These arguments are evaluated only once and immediately, when creating the restarting handler. Furthermore, they support tidy dots features.
`.fields`	A character vector specifying the fields of the condition that should be passed as arguments to the restart. If named, the names (except empty names `""`) are used as argument names for calling the restart function. Otherwise the the fields themselves are used as argument names.

Details

Jumping to a restart point from a calling handler has two effects. First, the control flow jumps to wherever the restart was established, and the restart function is called (with `...`, or `.fields` as arguments). Execution resumes from the `with_restarts()` call. Secondly, the transfer of the control flow out of the function that signalled the condition means that the handler has dealt with the condition. Thus the condition will not be passed on to other potential handlers established on the stack.

See Also

`calling()` and `exiting()`.

Examples

```
# This is a restart that takes a data frame and names as arguments
rst_bar <- function(df, nms) {
  stats::setNames(df, nms)
}

# This restart is simpler and does not take arguments
rst_baz <- function() "baz"

# Signalling a condition parameterised with a data frame
fn <- function() {
  with_restarts(signal("A foobar condition occurred", "foo", foo_field = mtcars),
    rst_bar = rst_bar,
    rst_baz = rst_baz
  )
}

# Creating a restarting handler that passes arguments `nms` and
# `df`, the latter taken from a data field of the condition object
restart_bar <- restarting("rst_bar",
  nms = LETTERS[1:11], .fields = c(df = "foo_field")
)

# The restarting handlers jumps to `rst_bar` when `foo` is signalled:
with_handlers(fn(), foo = restart_bar)

# The restarting() constructor is especially nice to use with
# restarts that do not need arguments:
with_handlers(fn(), foo = restarting("rst_baz"))
```

`return_from` *Jump to or from a frame*

Description

Questioning

While `base::return()` can only return from the current local frame, these two functions will return from any frame on the current evaluation stack, between the global and the currently active context. They provide a way of performing arbitrary non-local jumps out of the function currently under evaluation.

Usage

```
return_from(frame, value = NULL)

return_to(frame, value = NULL)
```

Arguments

`frame`	An environment, a frame object, or any object with an `get_env()` method. The environment should be an evaluation environment currently on the stack.
`value`	The return value.

Details

`return_from()` will jump out of `frame`. `return_to()` is a bit trickier. It will jump out of the frame located just before `frame` in the evaluation stack, so that control flow ends up in `frame`, at the location where the previous frame was called from.

These functions should only be used rarely. These sort of non-local gotos can be hard to reason about in casual code, though they can sometimes be useful. Also, consider to use the condition system to perform non-local jumps.

Life cycle

The support for `frame` object is soft-deprecated. Please pass simple environments to `return_from()` and `return_to()`.

These functions are in the questioning lifecycle because we are considering simpler alternatives.

Examples

```
# Passing fn() evaluation frame to g():
fn <- function() {
  val <- g(current_env())
  cat("g returned:", val, "\n")
  "normal return"
}
g <- function(env) h(env)

# Here we return from fn() with a new return value:
h <- function(env) return_from(env, "early return")
fn()
```

```
# Here we return to fn(). The call stack unwinds until the last frame
# called by fn(), which is g() in that case.
h <- function(env) return_to(env, "early return")
fn()
```

`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::enframe)` 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::enframe
# )
# stop("foo")
```

`rst_abort` *Jump to the abort restart*

Description

Questioning

The abort restart is the only restart that is established at top level. It is used by R as a top-level target, most notably when an error is issued (see `abort()`) that no handler is able to deal with (see `with_handlers()`).

Usage

```
rst_abort()
```

Life cycle

All the restart functions are in the questioning stage. It is not clear yet whether we want to recommend restarts as a style of programming in R.

See Also

`rst_jump()`, `abort()`

Examples

```
# The `abort` restart is a bit special in that it is always
# registered in a R session. You will always find it on the restart
# stack because it is established at top level:
rst_list()

# You can use the `above` restart to jump to top level without
# signalling an error:
## Not run:
fn <- function() {
  cat("aborting...\n")
  rst_abort()
  cat("This is never called\n")
}
{
  fn()
  cat("This is never called\n")
}

## End(Not run)

# The `above` restart is the target that R uses to jump to top
# level when critical errors are signalled:
## Not run:
{
  abort("error")
  cat("This is never called\n")
}
```

```
## End(Not run)

# If another `abort` restart is specified, errors are signalled as
# usual but then control flow resumes with from the new restart:
## Not run:
out <- NULL
{
  out <- with_restarts(abort("error"), abort = function() "restart!")
  cat("This is called\n")
}
cat("`out` has now become:", out, "\n")

## End(Not run)
```

`rst_list` *Restarts utilities*

Description

Questioning

Restarts are named jumping points established by `with_restarts()`. `rst_list()` returns the names of all restarts currently established. `rst_exists()` checks if a given restart is established. `rst_jump()` stops execution of the current function and jumps to a restart point. If the restart does not exist, an error is thrown. `rst_maybe_jump()` first checks that a restart exists before jumping.

Usage

```
rst_list()

rst_exists(.restart)

rst_jump(.restart, ...)

rst_maybe_jump(.restart, ...)
```

Arguments

`.restart`	The name of a restart.
`...`	Arguments passed on to the restart function. These dots support tidy dots features.

Life cycle

All the restart functions are in the questioning stage. It is not clear yet whether we want to recommend restarts as a style of programming in R.

See Also

`with_restarts()`

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.

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, encoding = NULL)

is_scalar_bytes(x)
```

Arguments

x	object to be tested.
encoding	Expected encoding of a string or character vector. One of UTF-8, latin1, or unknown.

See Also

type-predicates, bare-type-predicates

scoped_bindings *Temporarily change bindings of an environment*

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

`...`	Pairs of names and values. These dots support splicing (with value semantics) and name unquoting.
`.env`	An environment.
`.frame`	The frame environment that determines the scope of the temporary bindings. When that frame is popped from the call stack, bindings are switched back to their original values.
`.expr`	An expression to evaluate with temporary bindings.

Value

`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 rebinded while executing `expr`
with_bindings(paste(foo, bar), foo = "rebinded")
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

`...`	For `scoped_options()` and `push_options()`, named values defining new option values. For `peek_options()`, strings or character vectors of option names.
`.frame`	The environment of a stack frame which defines the scope of the temporary options. When the frame returns, the options are set back to their original values.
`.expr`	An expression to evaluate with temporary options.
`name`	An option name as string.

Value

For `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

from	The starting point of the sequence.
to	The end point.
x	A vector whose length is the end point.

Value

An integer vector containing a strictly increasing sequence.

Examples

```
seq2(2, 10)
seq2(10, 2)
seq(10, 2)

seq2_along(10, letters)
```

---

set_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

x	An expression, closure, or one-sided formula. In addition, `set_expr()` accept frames.
value	An updated expression.
default	A default expression to return when `x` is not an expression wrapper. Defaults to `x` 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 ommitted a vector is named with itself
set_names(letters[1:5])

# Alternatively you can supply a function
set_names(1:10, ~ letters[seq_along(.)])
set_names(head(mtcars), toupper)

# `...` is passed to the function:
set_names(head(mtcars), paste0, "_foo")
```

`string` *Create a string*

Description

These base-type constructors allow more control over the creation of strings in R. They take character vectors or string-like objects (integerish or raw vectors), and optionally set the encoding. The string version checks that the input contains a scalar string.

Usage

```
string(x, encoding = NULL)
```

Arguments

`x`	A character vector or a vector or list of string-like objects.
`encoding`	If non-null, passed to `set_chr_encoding()` to add an encoding mark. This is only declarative, no encoding conversion is performed.

See Also

`set_chr_encoding()` for more information about encodings in R.

Examples

```
# As everywhere in R, you can specify a string with Unicode
# escapes. The characters corresponding to Unicode codepoints will
# be encoded in UTF-8, and the string will be marked as UTF-8
# automatically:
cafe <- string("caf\uE9")
str_encoding(cafe)
as_bytes(cafe)

# In addition, string() provides useful conversions to let
# programmers control how the string is represented in memory. For
# encodings other than UTF-8, you'll need to supply the bytes in
# hexadecimal form. If it is a latin1 encoding, you can mark the
# string explicitly:
```

```
cafe_latin1 <- string(c(0x63, 0x61, 0x66, 0xE9), "latin1")
str_encoding(cafe_latin1)
as_bytes(cafe_latin1)
```

---

sym *Create a symbol or list of symbols*

Description

These functions take strings as input and turn them into symbols. Contrarily to `as.name()`, they convert the strings to the native encoding beforehand. This is necessary because symbols remove silently the encoding mark of strings (see `set_str_encoding()`).

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 *Collect dots tidily*

Description

`list2()` is equivalent to `list(...)` but provides tidy dots semantics:

- 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 `:=`.

We call quasiquotation support in dots **tidy dots** semantics and functions taking dots with `list2()` tidy dots functions. Quasiquotation is an alternative to `do.call()` idioms and gives the users of your functions an uniform syntax to supply a variable number of arguments or a variable name.

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

Details

Note that while all tidy eval quoting functions have tidy dots semantics, not all tidy dots functions are quoting functions. `list2()` is for standard functions, not quoting functions.

Value

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

Life cycle

One difference of `dots_list()` with `list2()` is that it always allocates a vector of names even if no names were supplied. In this case, the names are all empty `""`. This is for consistency with `enquos()` and `enexprs()` but can be quite costly when long lists are spliced in the results. For this reason we plan to parameterise this behaviour with a `.named` argument and possibly change the default. `list2()` does not have this issue.

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 })
```

`tidyeval-data` *Data pronoun for tidy evaluation*

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` *Capture a backtrace*

Description

A backtrace captures the sequence of calls that lead to the current function, sometimes called the call stack. Because of lazy evaluation, the call stack in R is actually a tree, which the `summary()` method of this object will reveal.

Usage

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

Arguments

top	The first frame environment to be included in the backtrace. This becomes the top of the backtrace tree and represents the oldest call in the backtrace. This is needed in particular when you call `trace_back()` indirectly or from a larger context, for example in tests or inside an RMarkdown document where you don't want all of the knitr evaluation mechanisms to appear in the backtrace.
bottom	The last frame environment to be included in the backtrace. This becomes the rightmost leaf of the backtrace tree and represents the youngest call in the backtrace. Set this when you would like to capture a backtrace without the capture context. Can also be an integer that will be passed to `caller_env()`.

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)
```

`type-predicates` *Type predicates*

Description

These type predicates aim to make type testing in R more consistent. They are wrappers around `base::typeof()`, so operate at a level beneath S3/S4 etc.

Usage

```
is_list(x, n = NULL)

is_atomic(x, n = NULL)

is_vector(x, n = NULL)

is_integer(x, n = NULL)

is_double(x, n = NULL, finite = NULL)

is_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

`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.
`encoding`	Expected encoding of a string or character vector. One of `UTF-8`, `latin1`, or `unknown`.

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

`vector-construction` *Create vectors*

Description

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(..., .encoding = NULL)

bytes(...)
```

Arguments

`...`	Components of the new vector. Bare lists and explicitly spliced lists are spliced.
`.encoding`	If non-null, passed to `set_chr_encoding()` to add an encoding mark. This is only declarative, no encoding conversion is performed.

Life cycle

- 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

```
# For cleaner backtraces:
options(rlang_trace_top_env = current_env())

# with_abort() automatically casts simple errors thrown by stop()
# to rlang errors:
fn <- function() stop("Base error")
try(with_abort(fn()))
last_error()

# with_abort() is handy for rethrowing low level errors. The
# backtraces are then segmented between the low level and high
# level contexts.
low_level1 <- function() low_level2()
low_level2 <- function() stop("Low level error")

high_level <- function() {
  with_handlers(
    with_abort(low_level1()),
    error = ~ abort("High level error", parent = .x)
  )
}

try(high_level())
last_error()
summary(last_error())

# Reset to default
options(rlang_trace_top_env = NULL)
```

---

with_env *Evaluate an expression within a given environment*

Description

These functions evaluate `expr` within a given environment (`env` for `with_env()`, or the child of the current environment for `locally`). They rely on `eval_bare()` which features a lighter evaluation mechanism than base R `base::eval()`, and which also has some subtle implications when evaluting stack sensitive functions (see help for `eval_bare()`).

Usage

```
with_env(env, expr)

locally(expr)
```

Arguments

env	An environment within which to evaluate expr. Can be an object with a get_env() method.
expr	An expression to evaluate.

Details

locally() is equivalent to the base function base::local() but it produces a much cleaner evaluation stack, and has stack-consistent semantics. It is thus more suited for experimenting with the R language.

Life cycle

These functions are experimental. Expect API changes.

Examples

```
# with_env() is handy to create formulas with a given environment:
env <- child_env("rlang")
f <- with_env(env, ~new_formula())
identical(f_env(f), env)

# Or functions with a given enclosure:
fn <- with_env(env, function() NULL)
identical(get_env(fn), env)


# Unlike eval() it doesn't create duplicates on the evaluation
# stack. You can thus use it e.g. to create non-local returns:
fn <- function() {
  g(current_env())
  "normal return"
}
g <- function(env) {
  with_env(env, return("early return"))
}
fn()


# Since env is passed to as_environment(), it can be any object with an
# as_environment() method. For strings, the pkg_env() is returned:
with_env("base", ~mtcars)

# This can be handy to put dictionaries in scope:
with_env(mtcars, cyl)
```

---

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 `cnd_signal()` and `abort()` for two common signal functions). They come in two types: exiting handlers, which jump out of the signalling context and are transferred to `with_handlers()` before being executed. And calling handlers, which are executed within the signal functions.

Usage

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

Arguments

`.expr`	An expression to execute in a context where new handlers are established. The underscored version takes a quoted expression or a quoted formula.
`...`	Named handlers. These should be functions of one argument. These handlers are treated as exiting by default. Use `calling()` to specify a calling handler. These dots support tidy dots features and are passed to `as_function()` to enable the formula shortcut for lambda functions.

Details

An exiting handler is taking charge of the condition. No other handler on the stack gets a chance to handle the condition. The handler is executed and `with_handlers()` returns the return value of that handler. On the other hand, in place handlers do not necessarily take charge. If they return normally, they decline to handle the condition, and R looks for other handlers established on the evaluation stack. Only by jumping to an earlier call frame can a calling handler take charge of the condition and stop the signalling process. Sometimes, a muffling restart has been established for the purpose of jumping out of the signalling function but not out of the context where the condition was signalled, which allows execution to resume normally. See `cnd_muffle()` and the `mufflable` argument of `cnd_signal()`.

Exiting handlers are established first by `with_handlers()`, and in place handlers are installed in second place. The latter handlers thus take precedence over the former.

See Also

`exiting()`, `calling()`.

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 = exiting(handler))

# Handlers are exiting by default so you can omit the adjective:
with_handlers(fn(), foo = handler)

# In place 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 an in place 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))
```

`with_restarts` *Establish a restart point on the stack*

Description

Questioning

Restart points are named functions that are established with `with_restarts()`. Once established, you can interrupt the normal execution of R code, jump to the restart, and resume execution from there. Each restart is established along with a restart function that is executed after the jump and that provides a return value from the establishing point (i.e., a return value for `with_restarts()`).

Usage

```
with_restarts(.expr, ...)
```

Arguments

`.expr`	An expression to execute with new restarts established on the stack. This argument is passed by expression and supports unquoting. It is evaluated in a context where restarts are established.
`...`	Named restart functions. The name is taken as the restart name and the function is executed after the jump. These dots support tidy dots features.

Details

Restarts are not the only way of jumping to a previous call frame (see `return_from()` or `return_to()`). However, they have the advantage of being callable by name once established.

Life cycle

All the restart functions are in the questioning stage. It is not clear yet whether we want to recommend restarts as a style of programming in R.

See Also

`return_from()` and `return_to()` for a more flexible way of performing a non-local jump to an arbitrary call frame.

Examples

```
# Restarts are not the only way to jump to a previous frame, but
# they have the advantage of being callable by name:
fn <- function() with_restarts(g(), my_restart = function() "returned")
g <- function() h()
h <- function() { rst_jump("my_restart"); "not returned" }
fn()

# Whereas a non-local return requires to manually pass the calling
# frame to the return function:
fn <- function() g(current_env())
g <- function(env) h(env)
h <- function(env) { return_from(env, "returned"); "not returned" }
fn()


# rst_maybe_jump() checks that a restart exists before trying to jump:
fn <- function() {
  g()
  cat("will this be called?\n")
}
g <- function() {
  rst_maybe_jump("my_restart")
  cat("will this be called?\n")
}

# Here no restart are on the stack:
fn()

# If a restart point called `my_restart` was established on the
# stack before calling fn(), the control flow will jump there:
rst <- function() {
  cat("restarting...\n")
  "return value"
}
with_restarts(fn(), my_restart = rst)


# Restarts are particularly useful to provide alternative default
# values when the normal output cannot be computed:
```

```
fn <- function(valid_input) {
  if (valid_input) {
    return("normal value")
  }

  # We decide to return the empty string "" as default value. An
  # altenative strategy would be to signal an error. In any case,
  # we want to provide a way for the caller to get a different
  # output. For this purpose, we provide two restart functions that
  # returns alternative defaults:
  restarts <- list(
    rst_empty_chr = function() character(0),
    rst_null = function() NULL
  )

  with_restarts(splice(restarts), .expr = {

    # Signal a typed condition to let the caller know that we are
    # about to return an empty string as default value:
    cnd_signal("default_empty_string")

    # If no jump to with_restarts, return default value:
    ""
  })
}

# Normal value for valid input:
fn(TRUE)

# Default value for bad input:
fn(FALSE)

# Change the default value if you need an empty character vector by
# defining a calling handler that jumps to the restart. It has to
# be calling because exiting handlers jump to the place where they
# are established before being executed, and the restart is not
# defined anymore at that point:
rst_handler <- calling(function(c) rst_jump("rst_empty_chr"))
with_handlers(fn(FALSE), default_empty_string = rst_handler)

# You can use restarting() to create restarting handlers easily:
with_handlers(fn(FALSE), default_empty_string = restarting("rst_null"))
```

---

zap *Create zap objects*

Description

`zap()` creates a sentinel object that indicates that an object should be removed. For instance, named zaps instruct `env_bind()` and `call_modify()` to remove those objects from the environment or the call.

The advantage of zap objects is that they unambiguously signal the intent of removing an object. Sentinels like `NULL` or `missing_arg()` are ambiguous because they represent valid R objects.

Usage

```
zap()

is_zap(x)
```

Arguments

x	An object to test.

Examples

```
# Create one zap object:
zap()

# Create a list of zaps:
rep(list(zap()), 3)
rep_named(c("foo", "bar"), list(zap()))
```

Index