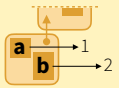
# Tidy evaluation with rlang : : **CHEAT SHEET**

## Vocabulary

**Tidy Evaluation (Tidy Eval)** is not a package, but a framework for doing non-standard evaluation (i.e. delayed evaluation) that makes it easier to program with tidyverse functions.

**pi** — **Symbol** - a name that represents a value or object stored in R. *is_symbol(expr(pi))*




**Environment** - a list-like object that binds symbols (names) to objects stored in memory. Each env contains a link to a second, **parent** env, which creates a chain, or search path, of environments. *is_environment(current_env())*

rlang::**caller_env**(n = 1) Returns calling env of the function it is in.

rlang::**child_env**(.parent, ...) Creates new env as child of .parent. Also **env**.

rlang::**current_env**() Returns execution env of the function it is in.

**1** — **Constant** - a bare value (i.e. an atomic vector of length 1). *is_bare_atomic(1)*

**abs** ( **1** ) — **Call object** - a vector of symbols/constants/calls that begins with a function name, possibly followed by arguments. *is_call(expr(abs(1)))*

**pi** — code
3.14 — result

**Code** - a sequence of symbols/constants/calls that will return a result if evaluated. Code can be:

1. Evaluated immediately (**Standard Eval**)
2. Quoted to use later (**Non-Standard Eval**)

*is_expression(expr(pi))*

**e**
**a + b** — **Expression** - an object that stores quoted code without evaluating it. is_*expression(expr(a + b))*

**q**
**a + b,** — **Quosure**- an object that stores both quoted code (without evaluating it) and the code's environment. *is_quosure(quo(a + b))*

rlang::**quo_get_env**(quo) Return the environment of a quosure.

rlang::**quo_set_env**(quo, expr) Set the environment of a quosure.

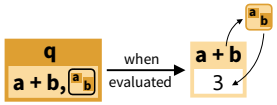
**a + b** rlang::**quo_get_expr**(quo) Return the expression of a quosure.

**Expression Vector** - a list of pieces of quoted code created by base R's *expression* and *parse* functions. Not to be confused with **expression**.

## Quoting Code

Quote code in one of two ways (if in doubt use a quosure):

### QUOSURES




**Quosure**- An expression that has been saved *with an environment* (aka a closure).

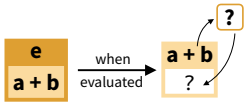
A quosure can be evaluated later in the stored environment to return a predictable result.

rlang::**quo**(expr) Quote contents as a quosure. Also **quos** to quote multiple expressions. *a <- 1; b <- 2; q <- quo(a + b); qs <- quos(a, b)*

rlang::**enquo**(arg) Call from within a function to quote what the user passed to an argument as a quosure. Also **enquos** for multiple args.
*quote_this < - function(x) enquo(x)*
*quote_these < - function(…) enquos(…)*

rlang::**new_quosure**(expr, env = caller_env()) Build a quosure from a quoted expression and an environment.
*new_quosure(expr(a + b), current_env())*


### EXPRESSION




**Quoted Expression** - An expression that has been saved by itself.

A quoted expression can be evaluated later to return a result that will depend on the environment it is evaluated in

rlang::**expr**(expr) Quote contents. Also **exprs** to quote multiple expressions. *a <- 1; b <- 2; e <- expr(a + b); es <- exprs(a, b, a + b)*

rlang::**enexpr**(arg) Call from within a function to quote what the user passed to an argument. Also **enexprs** to quote multiple arguments.
*quote_that < - function(x) enexpr(x)*
*quote_those < - function(…) enexprs(…)*

rlang::**ensym**(x) Call from within a function to quote what the user passed to an argument as a symbol, accepts strings. Also **ensyms**.
*quote_name < - function(name) ensym(name)*
*quote_names < - function(…) ensyms(…)*

## Parsing and Deparsing


**Parse** - Convert a string to a saved expression.

rlang::**parse_expr**(x) Convert a string to an expression. Also **parse_exprs**, **sym**, **parse_quo**, **parse_quos**. e<-*parse_expr("a+b")*

**Deparse** - Convert a saved expression to a string.

rlang::**expr_text**(expr, width = 60L, nlines = Inf) Convert expr to a string. Also **quo_name**. *expr_text(e)*

## Building Calls

rlang::**call2**(.fn, ..., .ns = NULL) Create a call from a function and a list of args. Use **exec** to create and then evaluate the call. (See back page for **!!!**) *args <- list(x = 4, base = 2)*

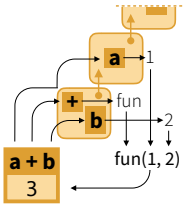
**log** (x = **4** , base = **2** )

**2**

*call2("log", x = 4, base = 2)*
*call2("log", !!!args)*

*exec("log", x = 4, base = 2)*
*exec("log", !!!args)*

## Evaluation



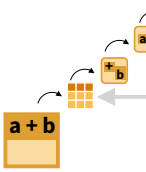

To evaluate an expression, R :

1. Looks up the symbols in the expression in the active environment (or a supplied one), followed by the environment's parents
2. Executes the calls in the expression

***The result of an expression depends on which environment it is evaluated in.***

### QUOTED EXPRESSION

rlang::**eval_bare**(expr, env = parent.frame()) Evaluate expr in env. *eval_bare(e, env =.GlobalEnv)*




*a <- 1; b <- 2*
*p <- quo(**.data$**a + **!!**b)*
*mask <- tibble(a = 5, b = 6)*
*eval_tidy(p, data = mask)*

### QUOSURES (and quoted exprs)

rlang::**eval_tidy**(expr, data = NULL, env = caller_env()) Evaluate expr in env, using data as a **data mask**. Will evaluate quosures in their stored environment. *eval_tidy(q)*

**Data Mask** - If data is non-NULL, eval_tidy inserts data into the search path before env, matching symbols to names in data.

Use the pronoun **.data$** to force a symbol to be matched in data, and **!!** (see back) to force a symbol to be matched in the environments.

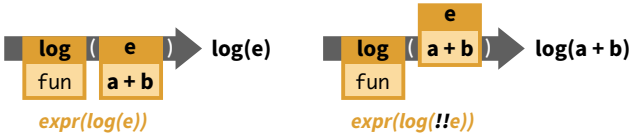
# Quasiquotation (!!, !!!, :=)

### QUOTATION

Storing an expression without evaluating it.
*e <- expr(a + b)*

### QUASIQUOTATION

Quoting *some* parts of an expression while evaluating and then inserting the results of others (**unquoting** others).
*e <- expr(a + b)*

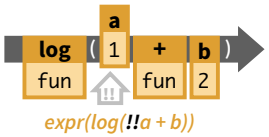



rlang provides **!!**, **!!!**, and **:=** for doing quasiquotation.

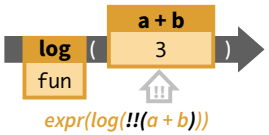
**!!**, **!!!**, and **:=** are not functions but syntax (symbols recognized by the functions they are passed to). Compare this to how

. is used by magrittr::**%>%()**
. is used by stats::**lm()**
.x is used by purrr::**map()**, and so on.

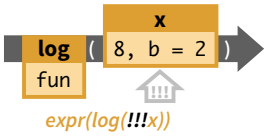
**!!**, **!!!**, and **:=** are only recognized by some rlang functions and functions that use those functions (such as tidyverse functions).



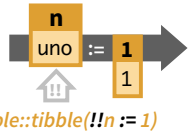

**!!** Unquotes the symbol or call that follows. Pronounced "unquote" or "bang-bang." *a <- 1; b <- 2*
*expr(log(!!a + b))*




Combine **!!** with **()** to unquote a longer expression.
*a <- 1; b <- 2*
*expr(log(!!(a + b)))*




**!!!** Unquotes a vector or list and splices the results as arguments into the surrounding call. Pronounced "unquote splice" or "bang-bang-bang."
*x <- list(8, b = 2)*
*expr(log(!!!x))*




**:=** Replaces an = to allow unquoting within the name that appears on the left hand side of the =. Use with **!!**
*n <- expr(uno)*
*tibble::tibble(!!n := 1)*

# Programming Recipes

**Quoting function**- A function that quotes any of its arguments internally for delayed evaluation in a chosen environment. You must take **special steps to program safely** with a quoting function.

**How to spot a quoting function?**
A function quotes an argument if the argument returns an error when run on its own.

Many tidyverse functions are quoting functions: e.g. **filter**, **select**, **mutate**, **summarise**, etc.

dplyr::filter(cars, **speed = = 25**)

| speed | dist |
|---|---|
| 1 | 25 85 |

**speed == 25**
Error!

### PROGRAM WITH A QUOTING FUNCTION

```
data_mean <- function(data, var) {
  require(dplyr)
  var <- rlang::enquo(var)                    1
  data %>%
    summarise(mean = mean(!!var))             2
}
```

1. Capture user argument that will be quoted with rlang::**enquo**.
2. Unquote the user argument into the quoting function with **!!**.

### PASS MULTIPLE ARGUMENTS TO A QUOTING FUNCTION

```
group_mean <- function(data, var, …) {
  require(dplyr)
  var <- rlang::enquo(var)
  group_vars <- rlang::enquos(…)              1
  data %>%
    group_by(!!!group_vars) %>%               2
    summarise(mean = mean(!!var))
}
```

1. Capture user arguments that will be quoted with rlang::**enquos**.
2. Unquote splice the user arguments into the quoting function with **!!!**.

### PASS TO ARGUMENT NAMES OF A QUOTING FUNCTION

```
named_mean <- function(data, var) {
  require(dplyr)
  var <- rlang::ensym(var)                    1
  data %>%
    summarise(!!name := mean(!!var))          2
}
```

1. Capture user argument that will be quoted with rlang::**ensym**.
2. Unquote the name into the quoting function with **!!** and **:=**.

### MODIFY USER ARGUMENTS

```
my_do <- function(f, v, df) {
  f <- rlang::enquo(f)                        1
  v <- rlang::enquo(v)
  todo <- rlang::quo((!!f)(!!v))              2
  rlang::eval_tidy(todo, df)                  3
}
```

1. Capture user arguments with rlang::**enquo**.
2. **Unquote** user arguments into a new expression or quosure to use
3. **Evaluate** the new expression/ quosure instead of the original argument

### APPLY AN ARGUMENT TO A DATA FRAME

```
subset2 <- function(df, rows) {
  rows <- rlang::enquo(rows)                  1
  vals <- rlang::eval_tidy(rows, data = df)
  df[vals, , drop = FALSE]                    2
}
```

1. Capture user argument with rlang::**enquo**.
2. Evaluate the argument with rlang::**eval_tidy**. Pass the data frame to **data** to use as a data mask.
3. **Suggest** in your documentation that your users use the **.data** and **.env** pronouns.

### PASS CRAN CHECK

```
#' @importFrom rlang .data                    1
mutate_y <- function(df) {
  dplyr::mutate(df, y = .data$a +1)           2
}
```

Quoted arguments in tidyverse functions can trigger an **R CMD check** NOTE about undefined global variables. To avoid this:

1. Import rlang::**.data** to your package, perhaps with the roxygen2 tag **@importFrom rlang .data**
2. Use the **.data$** pronoun in front of variable names in tidyverse functions

### WRITE A FUNCTION THAT RECOGNIZES QUASIQUOTATION (!!,!!!,:=)

1. Capture the quasiquotation-aware argument with rlang::**enquo**.
2. Evaluate the arg with rlang::**eval_tidy**.

```
add1 <- function(x) {
  q <- rlang::enquo(x)                        1
  rlang::eval_tidy(q) + 1                     2
}
```