

# API design

<https://design.tidyverse.org/> + <https://tidydesign.substack.com/>

**September 2023**



# Your turn

What function do you have the most trouble remembering how to use? What do you think makes it so hard?

# Goals

1. Starting building your API muscles
2. Some concrete ideas to work on later
3. Give me feedback 😊

# Today we'll focus on the function spec

`args(read.csv)`

```
#> function (file, header = TRUE, sep = ",", quote = "\"",  
#>   dec = ".", fill = TRUE, comment.char = "", ... )
```

`args(rep)`

```
#> function (x, ... )
```

`args(sample)`

```
#> function (x, size, replace = FALSE, prob = NULL)
```

`args(grepl)`

```
#> function (pattern, x, ignore.case = FALSE, perl = FALSE,  
#>   fixed = FALSE, useBytes = FALSE)
```

# The function spec is important because

1. It's shown in the docs
2. It's shown in tooltips
3. It's stored in your brain?

# The seven principles of scannable function specs

1. Make arguments explicit
2. Put the most important arguments first
3. Required arguments shouldn't have defaults
4. Put ... after required arguments
5. Keep defaults short and sweet
6. Enumerate possible options
7. Reduce clutter with an options object

Make arguments explicit

```
# The previous worst offender
data.frame(x = "a")
options(stringsAsFactors = TRUE)
data.frame(x = "a")

as.POSIXct("2020-01-01 09:00")
factor(c("cherry", "honeydew")) # Czech
```



# What does as.POSIXct look like?

```
as.POSIXct ← function(x, tz = "", ... ) {  
  UseMethod("as.POSIXct")  
}
```

We could fix to a specific timezone...

```
as.POSIXct ← function(x, tz = "UTC", ... ) {  
  UseMethod("as.POSIXct")  
}
```

Or remove the default

```
as.POSIXct ← function(x, tz, ... ) {  
  UseMethod("as.POSIXct")  
}
```

It would be more helpful to make the default more explicit

```
as.POSIXct ← function(x, tz = Sys.timezone(), ... ) {  
  UseMethod("as.POSIXct")  
}
```

Or even print it when not supplied

```
as.POSIXct ← function(x, tz = Sys.timezone()) {  
  if (missing(tz)) {  
    message("Using `tz = \"\", tz, \"\\\"`")  
  }  
  base::as.POSIXct(x, tz = tz)  
}  
as.POSIXct("2020-01-01 09:00")
```

Why this construction?

# Should this be a warning?

# Your turn

What's the downside of assuming a specific locale? Think about functions like `str_sort()` and `dplyr::arrange()`. How do they sort by default? How would you want them to sort? How might other people want them to sort?

Put the most important  
arguments first

# Rules of thumb

1. If the output is a transformation of an input (e.g. `log()`, `stringr::str_replace()`, `dplyr::left_join()`) then that argument the most important.
2. Other arguments that determine the type or shape of the output are typically very important.
3. Optional arguments (i.e. arguments with a default) are the least important, and should come last.



```
grep(pattern, x, ...)
```

```
gsub(pattern, replacement, x, ...)
```

```
lm(formula, data, ...)
```

```
gather(data, key = "key", value = "value", ...)
```



The variables to pivot (required)

# Your turn

Why do the order of data and aesthetics arguments vary between `ggplot2()` and (e.g.) `geom_point()`?

Are there other tidyverse functions that you think violate this principle?

Required arguments  
shouldn't have defaults

# Mostly historical issues

```
sample(x, size, replace = FALSE, prob = NULL)
```

```
diag(x = 1, nrow, ncol, names = TRUE)
```

```
lm(formula, data, subset, weights, na.action, ...)
```

```
# But some people have continued to make mistakes
```

```
ggplot2::geom_abline(..., slope, intercept)
```

# Your turn

There's one pattern that we occasionally use that's an exception to this rule. Can you figure it out by looking at `dplyr::slice_head()`, `rvest::html_element()`, `forcats::fct_other()`, and `modelr::seq_range()`?

Put ... after required  
arguments

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

```
# Allows code like this
```

```
x ← c(1, 2, 10, NA)
```

```
mean(x, , TRUE)
```

```
mean(x, n = TRUE, t = 0.1)
```

```
mean(x, ..., trim = 0, na.rm = FALSE)
```

```
# Forces you to write
```

```
x ← c(1, 2, 10, NA)
```

```
mean(x, na.rm = TRUE)
```

```
mean(x, na.rm = TRUE, trim = 0.1)
```



```
# And this means you can add new arguments and  
# its guaranteed not to affect existing code  
mean(x, ..., trim = 0, na.rm = FALSE, new_arg = "xyz")
```

Keep defaults short and  
sweet

# Reshape has a very long default computation

```
reshape ← function(  
  ... ,  
  split = if (sep == "") {  
    list(regex = "[A-Za-z][0-9]", include = TRUE)  
  } else {  
    list(regex = sep, include = FALSE, fixed = TRUE)  
  }  
) {}
```

A simple fix is to use a default value of NULL

```
reshape ← function(  
  ... ,  
  split = NULL  
) {  
  if (is.null(split)) {  
    if (sep == "") {  
      split ← list(regex = "[A-Za-z][0-9]", include = TRUE)  
    } else {  
      split ← list(regex = sep, include = FALSE, fixed = TRUE)  
    }  
  }  
}
```

By it's slightly better to use a helper function

```
split_default ← function(sep = ".") {  
  if (sep == "") {  
    list(regex = "[A-Za-z][0-9]", include = TRUE)  
  } else {  
    list(regex = sep, include = FALSE, fixed = TRUE)  
  }  
}
```

```
reshape ← function(..., sep = ".", split = NULL) {  
  split ← split %||% split_default(sep)  
  ...  
}
```

# What is `%||%` ?

```
NULL %||% 1
```

```
#> 1
```

```
NULL %||% NULL %||% 2 %||% 3
```

```
#> 2
```

```
# get from rlang or write yourself
```

```
`%||%` ← function(a, b) {  
  if (is.null(a)) b else a  
}
```

Enumerate possible  
options

# What do these arguments have in common?

```
difftime(units = )
```

```
format(justify = )
```

```
trimws(which = )
```

```
rank(ties.method = )
```

```
dplyr::mutate(.keep = )
```

```
# And which value is used by default?
```



# The implementation of rank() looks something like this

```
rank ← function(x,  
                ties.method = c("average", "first", "last", "random", "max", "min")  
                ) {  
  
  ties.method ← match.arg(ties.method)  
  
  switch(ties.method,  
    average = ,  
    min = ,  
    max = .Internal(rank(x, length(x), ties.method)),  
    first = sort.list(sort.list(x)),  
    last = sort.list(rev.default(sort.list(x, decreasing = TRUE))),  
    random = sort.list(order(x, stats::runif(length(x)))))  
  )  
}
```

`match.arg()` makes this work

```
rank(x, ties.method = "middle")
```

```
#> Error in `match.arg()`:
```

```
#> ! 'arg' should be one of “average”, “first”,
```

```
#>    “last”, “random”, “max”, “min”
```

```
# But this works
```

```
rank(x, ties.method = "r")
```

# We prefer `rlang::arg_match()`

```
rank2 ← function(x, ties.method = c("average", "first", "last", "random",  
  "max", "min")) {  
  ties.method ← rlang::arg_match(ties.method)  
  rank(x, ties.method = ties.method)  
}
```

```
rank2(x, ties.method = "r")
```

```
#> Error in `rank2()`:
```

```
#> ! `ties.method` must be one of "average", "first", "last", "random",
```

```
#>   "max", or "min", not "r".
```

```
#> i Did you mean "random"?
```

# Because it also has some nice suggestion logic

```
rank2(x, ties.method = "avarage")  
#> Error in `rank2()`:  
#> ! `ties.method` must be one of "average", "first", "last", "random",  
#>   "max", or "min", not "avarage".  
#> i Did you mean "average"?
```

Reduce clutter with an  
options object

Instead of this

```
my_fun ← function(x, y,  
                  opt1 = 1,  
                  opt2 = 2,  
                  opt3 = 3,  
                  opt4 = 4,  
                  ) {
```

```
  ...
```

```
}
```

# Try this

```
my_fun ← function(x, y, options = my_fun_opts()) {  
  ...  
}
```

```
my_fun_opts ← function(opt1 = 1, opt2 = 2, ... ) {  
  list(  
    opt1 = opt1,  
    opt2 = opt2,  
    ...  
  )  
}
```

Practice



# Analyse these functions

```
tidy::pivot_longer()
```

```
dplyr::inner_join()
```

```
ggplot2::geom_point()
```