Data masking

Last modified: 2026-11-03 08:35:08 (PST)

General Overview

This chapter covers data masking, a unique process in R in which columns are treated as distinct objects within their dataframe's environment. In research code, data masking most frequently comes up when writing wrapper functions where arguments to indicate column names are supplied as strings. We often do this when we repeat the same code on multiple columns, and want to apply a function to a vector of strings that correspond to column names in a dataframe. For example, we might want to clean multiple columns using the same function or estimate the same model under different feature sets. Here, we try to break down what data masking is, why this error comes up, and common approaches to solve this problem.

What is Data Masking?

Within certain tidyverse operations, columns are called as if they were variables. For example, while running `df |> mutate(X = …)` R recognizes that `X` specifically references a column in `df` without explicitly stating its membership `df |> mutate(df$X = …)` or calling the column name as a string `df |> mutate("X" = …)`.

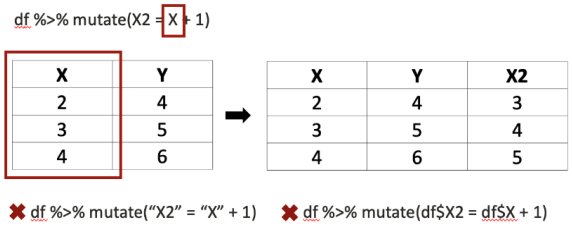


Figure 1: Data masking in tidyverse operations

However, this behavior may introduce errors when we attempt to incorporate variables from the global environment within these tidyverse pipelines. In the example shown in Figure 1, `column_name = "X"` followed by `df |> mutate(X2 = column_name + 1)` would yield an error,

since `column_name` is not a column in `df` and the variable `column_name` is not defined within the environment of `df`

Using tidy evaluation for data masking

In dplyr-based R programming, we make use of tidy evaluation. This allows us to avoid using base R syntax to reference specific columns in a data frame. By leveraging Tidy evaluation-based data masking, we can employ long pipes with several dplyr verbs to manipulate our data using stand-alone variables that store column names as strings.

For example, consider a data frame "df" that contains a column called "heavyrain" that we want to manipulate. Suppose we wanted to convert the values of "heavyrain" into a factor.

Using base R, which does not mask data, heavyrain must have quotes to be treated as a data-variable:

```
df[["outcome"]] = as.factor(df[["heavyrain"]])
```

In a dplyr pipe, heavyrain is being masked using tidy evaluation and will be correctly interpreted as a column because it is recognized as a data-variable:

```
df |> mutate(outcome = as.factor(heavyrain))
```

With modified data masking, heavyrain is a string that is coerced into being recognized as a data-variable:

```
var_name = "heavyrain"
df |> mutate(outcome = as.factor(!!sym(var_name)))
```

While cleaner and often more convenient, the data frame that `var_name` is in is now "masked" and we refer to the vectors in the dataframe (data-variables) as though they were objects of their own (environmental-variables). This is why we can just say the variable's name in the context of a pipe – we treat it as though it's an object defined in our environment. Within normal scripts, this is usually fine, because the data frame is "held on to" in the pipe. However, it can cause some programming hurdles when writing functions that take strings of variable/column names as arguments. In the next section, we briefly describe how to troubleshoot common errors in data masking, as relevant to research code.

Data masking vs. tidy selection

Many {tidyverse} verbs use *two* different flavors of tidy evaluation, and the way you refer to a column stored in a string variable depends on which one you are in.

- **Data-masking** contexts treat column names as values you compute *with*: the {dplyr} verbs `mutate()`, `filter()`, `summarize()`/`summarise()` (the summary expressions), `group_by()`, and `arrange()`, plus `aes()` in {ggplot2}.
- **Tidy-selection** contexts treat column names as columns you *choose*: `select()`, `rename()`, `relocate()`, `pull()`, the `.cols` argument of `across()`, the `.by` argument of `mutate()`/`summarize()`, and the `cols`, `names_from`, `values_from`, and `id_cols` arguments of `pivot_*()` in {tidyr}.

To refer to a column named by a string variable, use the tool that matches the context:

Table 1: Referring to a column named by a string variable

Context	Use	Avoid
Data-masking	`!!sym(var)` or `.data[[var]]`	bare `var` (uses the string value itself, not the column it names)
Tidy-selection	`all_of(var)` / `any_of(var)`	the `.data` pronoun (`.data[[var]]`, `.data$col`)

Use `all_of()` when every selected name must exist (it errors if one is missing) and `any_of()` when names that are absent should simply be dropped.

The `.data` pronoun does not belong in tidy selection

A common (and **silent**) mistake is reaching for the `.data` pronoun inside a tidy-selection context:

```
# Soft-deprecated: .data used in selection contexts
df |> select(.data$Subject, .data[[var]])
df |> summarize(.by = c(.data$group), med = median(value))
df |> tidyr::pivot_wider(names_from = .data$Parameter)
```

The `.data` pronoun is a *data-masking* tool, not a selection tool. Using it in a selection context was deprecated in {tidyselect} 1.2.0 (release notes), but it is a **soft** deprecation, signaled through {lifecycle}. Per the {lifecycle} documentation, a soft-deprecated feature warns only users who trigger it from the global environment and developers who use it directly (for

example when running {testthat} tests); there is *no warning when the deprecated feature is called indirectly by another package*. So in installed-package code it usually emits *no warning at all*, and it slips past interactive use, CI, and code review unnoticed. Rewrite these with `all_of()`/`any_of()` and plain strings:

```
# Idiomatic tidy selection
df |> select(all_of(c("Subject", var)))
df |> summarize(.by = all_of("group"), med = median(value))
df |> tidyr::pivot_wider(names_from = "Parameter")
```

To make this class of soft deprecation fail loudly, set `options(lifecycle_verbosity = "error")` in your tests (for example in `tests/testthat/setup.R`). This option promotes deprecation signals to errors regardless of the soft-versus-warning distinction ({lifecycle} reference). Plain `options(warn = 2)` will *not* catch it, because `warn = 2` only converts warnings into errors once a warning is actually signaled, and for a soft deprecation in installed-package code no warning is signaled in the first place.

Technical Overview

This section covers the R functions and tools that we often use in the context of data masking, focusing on the bang bang operator (`!!`) with symbol coercion (`sym()`) and the Walrus operator (`:=`).

The combined use of `!!` and `sym()` allows us to use strings, rather than data-variables, to reference column names within dplyr. Together, `!!sym("column_name")` forces dplyr to recognize "column_name" as a data-variable prior to evaluating the rest of the expression, enabling the ability to perform calculations on the column while referring to it as a string. `sym()` is a function that turns strings into symbols. In the context of a dplyr pipe, these symbols are interpreted as data-variables. The `!!` (bang bang) operator tells dplyr to evaluate the sym() expression first, e.g. to unquote its expression (e.g. "column_name") and evaluate it as a pre-existing object, first. This is helpful because often we use `sym("column_name")` within a larger expression, and dplyr might evaluate other elements of the expression first without !!, causing errors.

When we want to create a new column (via mutate or summarize), the Walrus operator (`:=`) allows us to specify the new column's name using a string. For example, while `df |> mutate("new_column" = values)` would yield an error, `df |> mutate("new_column" := values)` will correctly create a new column called "new_column". If we want to use a variable representing a string, we can use !! to force the variable to be evaluated before using `:=` to assign the value of the new column.

```
col_name = "new_column"
df |> mutate(!!col_name := values)
```

Example

Suppose we want to write a function "generate_descriptive_table" to summarize how the prevalence of "outcome" varies under different levels of a "risk_factor" in a data frame "df"

We can start by writing the function shell:

```
generate_descriptive_table <- function (df, outcome, rf) {
outcome_dist_by_rf <- ….
return(outcome_dist_by_rf)
}
```

Next, we can filter the data frame for only rows in which "rf" and "outcome" are not missing. We can use !! and sym() within filter to evaluate the strings stored in "rf" and "outcome". Note that defining `!!sym(outcome)` or `!!sym(rf)` in variables *outside of the dplyr pipeline* will *not* work.

```
generate_descriptive_table <- function(df, outcome, rf) {
  outcome_dist_by_rf <- df |>
    filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
    ….
  return(outcome_dist_by_rf)
}
```

Similarly, we use !! and sym() in group_by to evaluate column name, stored as a string in the argument "rf"

```
generate_descriptive_table <- function(df, outcome, rf) {
  outcome_dist_by_rf <- df |>
    filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
    group_by(!!sym(rf)) |>
    ….
  return(outcome_dist_by_rf)
}
```

Finally, we can use the walrus operator, !! and sym() with "summarize" to create a new column that takes the mean of the column referenced in "rf". We also use "glue" or "paste" to give the new column an informative name that includes the "outcome" it describes.

```r
generate_descriptive_table <- function(df, outcome, rf) {
  outcome_dist_by_rf <- df |>
    filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
    group_by(!!sym(rf)) |>
    summarize(!!(glue::glue("{outcome}_prev")) := mean(!!sym(outcome)))
  return(outcome_dist_by_rf)
}
```

OR

```r
generate_descriptive_table <- function(df, outcome, rf) {
  outcome_dist_by_rf <- df |>
    filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
    group_by(!!sym(rf)) |>
    summarize(!!(paste0(outcome, "_prev")) := mean(!!sym(outcome)))
  return(outcome_dist_by_rf)
}
```

OR

```r
generate_descriptive_table <- function(df, outcome, rf) {
  new_column_name = paste0(outcome, "_prev")
  outcome_dist_by_rf <- df |>
    filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
    group_by(!!sym(rf)) |>
    summarize(!!(new_column_name) := mean(!!sym(outcome)))
  return(outcome_dist_by_rf)
}
```