Specifying Variable Names in R

To Attach, or Not To Attach

Bruce Dudek

2025-04-02

Contents

1 Introduction

One of the things that R novices quickly learn is the cumbersome way of naming variables that are in data.frames. The `dataframe$variable` style of variable naming is slow to write and irritating. There are a few solutions to naming variables with shorter code, and we have leaned on one, the `attach` function quite a bit.

However, the `attach` function has some drawbacks, and best practices in R often recommend against using it.

case	degree_yrs	pubs	cits	salary	gender
1	3	18	50	51876	female
2	6	3	26	54511	female
3	3	2	50	53425	female
4	8	17	34	61863	male
5	9	11	41	52926	female
6	6	6	37	47034	male

The motivation for this short document is to outline those drawbacks, to show one alternative by using the `which` function, and to recommend best practices when using the `lm` or `aov` functions. At times, in my other tutorial documents, I have used `attach`, particularly in exploratory data analysis and graphics functions. But the present document shows the preferred alternative in the next to last section, one that also applies to many other R functions (e.g., `lm` or `aov`).

In addition, one point that is argued for helpfulness of the "tidyverse" as a primary way of doing R programming is the motivation to avoid use of the $ convention. Tools from the **dplyr** package and other tidyverse functions permit this avoidance. A brief section at the end of this document provides an overview.

2 Load the Primary Data Set

Let's work with the "Cohen" data set employed previously. (Cohen regression textbook data set on publications, citations, and salary of faculty members of a college department).

```
cohen <- read.csv("cohen.csv", stringsAsFactors=TRUE)
gt::gt(head(cohen))
```

An illustration of the need for more efficiency in variable naming is found if we try to test the pearson correlation between the salary and pubs variables.

We cannot do that calculation this first way because the global environment in R does not contain objects named "salary" and "pubs".

```
cor.test(pubs,salary)
```

```
Error: object 'pubs' not found
```

3 Illustrate normative variable naming

The inital solution is to fully name each variable with the "dataframe$varname" code structure.

```
cor.test(cohen$pubs,cohen$salary)
```

```
    Pearson's product-moment correlation

data:  cohen$pubs and cohen$salary
t = 4.5459, df = 60, p-value = 2.706e-05
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.2934802 0.6710778
sample estimates:
      cor
0.5061468
```

But this is the inefficient coding style that we are trying to avoid.

3.0.1 Use of the `with` function

A handy tool in R is the `with` function. We can use it to specify the data frame we are working with, and then embed the function that we want to use within that `which` code.

```
with(cohen, cor.test(pubs,salary))
```

```
    Pearson's product-moment correlation

data:  pubs and salary
t = 4.5459, df = 60, p-value = 2.706e-05
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.2934802 0.6710778
sample estimates:
      cor
0.5061468
```

This certainly works and may be advantageous when we have large numbers of variables to name in the desired function. But it is not necessarily the most efficient, and it leads to nested type code structures which may be difficult to follow.

4 Illustrate use of the `attach` function

Our preferred solution, up to now, has been to use the 'attach' function

```
attach(cohen)
```

Now, the cohen object is made available to the global environment in R and individual variable names will be recognized without using the "$" convention or the `with` function.

```
cor.test(pubs,salary)
```

```
    Pearson's product-moment correlation

data:  pubs and salary
t = 4.5459, df = 60, p-value = 2.706e-05
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.2934802 0.6710778
sample estimates:
      cor
0.5061468
```

The difficulty with using attach arises when multiple data frames or objects are attached. If more than one data frame or object has a variable name duplicated (or triplicated, etc), then only the most recently attached data frame will have that variable available. This can lead to confusion, errors, or in some cases use of the wrong variable without recognition of the problem.

In the R ecosystem, it is often strongly recommended not to use `attach`. But if there is no risk of duplicate variable names, then it is possible to use it safely, but the issue is that sometimes the user may not be aware of conflicts.

One method for addressing the potential problem is to `detach` the object after its need has passed.

```
detach(cohen)
```

Now we will see an error since R cannot find the variables since the `dataframe$` specifier is not included. Note that the error refers only to the first variable since the function errored out once it encountered the first error.

```
cor.test(salary,pubs)
```

```
Error: object 'salary' not found
```

Detach works works, but also requires writing more code. And, multiple attach and detach function executions seems unwieldly.

Another problem arises if many data frames have been attached and the memory of the user is poor on which ones had been attached. So, prior to detaching, it might be valuable to generate a list of attached objects.

Before generating the list, lets attach the cohen data frame again.

```
attach(cohen)
```

Now generate the list of attached objects.

```
intersect(search(), objects())
```

```
[1] "cohen"
```

But there is one downside to this as well. If "cohen" had been attached more than once, like this:

```
attach(cohen) # now the second time
```

```
The following objects are masked from cohen (pos = 3):

    case, cits, degree_yrs, gender, pubs, salary
```

```
attach(cohen) # now the third time
```

```
The following objects are masked from cohen (pos = 3):

    case, cits, degree_yrs, gender, pubs, salary

The following objects are masked from cohen (pos = 4):

    case, cits, degree_yrs, gender, pubs, salary
```

then we will have the same object attached three times (but notice in the RStudio "environment" pane that it is visible only once). Using the above code again, the impication is that only one is attached:

```
intersect(search(), objects())
```

```
[1] "cohen"
```

But if we detach,

```
detach(cohen) # remove last instance
```

and then ask for the list again:

```
intersect(search(), objects())
```

```
[1] "cohen"
```

we see the object still attached. We would have to execute `detach` three total times to remove the object. There are other ways of getting around this by writing a more extensive loop to remove all instances, but this hardly seems worth the effort for the benefit of using attach.

```
detach(cohen) #remove second instance
detach(cohen) #remove first instance
```

And ask again; this time it finds none….

```
intersect(search(), objects())
```

```
character(0)
```

5 Using a "data" argument with R functions

Many R functions, such as `lm`, `t.test`, and `aov`, permit use of a "data" argument that precludes the need for use of `with` or `attach`. This is convenient, plus it is a good style of coding because it makes clear exactly which data frame is being used in the function. Given these strengths, this approach is strongly recommended and is used in many accompanying tutorial docs for the 510/511 class.

```
fit1 <- lm(salary ~ pubs + cits, data=cohen)
summary(fit1)
```

```
Call:
lm(formula = salary ~ pubs + cits, data = cohen)

Residuals:
     Min       1Q   Median       3Q      Max
-17133.1  -5218.3   -341.3   5324.1  17670.3

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 40492.97    2505.39  16.162  < 2e-16 ***
pubs          251.75      72.92   3.452  0.00103 **
cits          242.30      59.47   4.074  0.00014 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7519 on 59 degrees of freedom
Multiple R-squared:  0.4195,    Adjusted R-squared:  0.3998
F-statistic: 21.32 on 2 and 59 DF,  p-value: 1.076e-07
```

It also works with the `t.test` function (and many others):

```
t.test(salary~gender, data=cohen)
```

```
    Welch Two Sample t-test

data:  salary by gender
t = -1.6714, df = 58.511, p-value = 0.09999
alternative hypothesis: true difference in means between group female and group male is not e
```

```
95 percent confidence interval:
 -8678.3306   779.6836
sample estimates:
mean in group female   mean in group male
            52650.00             56599.32
```

6 The "Tidyverse" approach

In the culture of the tidyverse, it is argued that the $ syntax is awkward and unnecessary - there is a bit of snobbery associated with this attitude to many base R functions among tidyverse adherents. An alternative approach using tools from the **dplyr** package and using pipes is argued to be "better". Perhaps…….

I have taken this wording and code directly from the `dplyr` programming page:

https://dplyr.tidyverse.org/articles/programming.html

"Data masking makes data manipulation faster because it requires less typing. In most (but not all) base R functions you need to refer to variables with $, leading to code that repeats the name of the data frame many times:"

This next code chunk extracts lines of data from the starwars data frame - found in the **dplyr** package - where species is human and homeworld is Naboo. It uses the base R bracketing/subsetting notation (you can't see the "species" variable in the rendered table listing because the tibble table is truncated). Notice that the cases where the homeworld and species variable don't fit the specification are returned as missing (NA).

```
library(dplyr)
starwars[starwars$homeworld == "Naboo" & starwars$species == "Human", ,]
```

```
# A tibble: 13 x 14
   name     height  mass hair_color skin_color eye_color birth_year sex   gender
   <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
 1 Palpati~    170    75 grey       pale       yellow            82 male  mascu~
 2 <NA>         NA    NA <NA>       <NA>       <NA>              NA <NA>  <NA>
 3 <NA>         NA    NA <NA>       <NA>       <NA>              NA <NA>  <NA>
 4 Padmé A~    185    45 brown      light      brown             46 fema~ femin~
 5 Ric Olié    183    NA brown      fair       blue              NA male  mascu~
 6 Quarsh ~    183    NA black      dark       brown             62 male  mascu~
 7 <NA>         NA    NA <NA>       <NA>       <NA>              NA <NA>  <NA>
 8 <NA>         NA    NA <NA>       <NA>       <NA>              NA <NA>  <NA>
 9 Dormé       165    NA brown      light      brown             NA fema~ femin~
10 <NA>         NA    NA <NA>       <NA>       <NA>              NA <NA>  <NA>
```

```
11 <NA>         NA    NA <NA>       <NA>       <NA>              NA <NA>  <NA>
12 <NA>         NA    NA <NA>       <NA>       <NA>              NA <NA>  <NA>
13 <NA>         NA    NA <NA>       <NA>       <NA>              NA <NA>  <NA>
# i 5 more variables: homeworld <chr>, species <chr>, films <list>,
#   vehicles <list>, starships <list>
```

"The **dplyr** equivalent of this code is more concise because data masking allows you to need to type starwars once."

In the next code chunk, the use of the pipe operator (%>%) from the **magrittr** package permits reading the code in this manner:

- Start with the starwars data frame.
- Filter that data frame so that the returned object contains only rows (cases) of data where the homeworld and species variables match the indicated values.

```
starwars %>% dplyr::filter(homeworld == "Naboo", species == "Human")
```

```
# A tibble: 5 x 14
  name      height  mass hair_color skin_color eye_color birth_year sex   gender
  <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
1 Palpatine    170    75 grey       pale       yellow            82 male  mascu~
2 Padmé Am~    185    45 brown      light      brown             46 fema~ femin~
3 Ric Olié     183    NA brown      fair       blue              NA male  mascu~
4 Quarsh P~    183    NA black      dark       brown             62 male  mascu~
5 Dormé        165    NA brown      light      brown             NA fema~ femin~
# i 5 more variables: homeworld <chr>, species <chr>, films <list>,
#   vehicles <list>, starships <list>
```

This left to right reading is argued to be "easier". This example also illustrates the tidyverse capability to avoid the R bracketing indexing to do subsetting. But the main point is that the $ syntax is avoided.

We could also use this code style in regression, but it is limiting to only provide the summary with the tidy function (CI's?, etc). Bottom line is that the tidyverse approach involves quite a different strategy and mindset to R programming and requires its own learning curve. I won't take this document into an argument about "which is better".

Here, the output of the `lm` function is passed to the `tidy` function from the broom package. The cohen data set is specified as the data frame to be used via the pipes operator. And then, inside `lm`, there is not data argument and the single period tells `lm` to use the indicated data frame specified in the pipe. Note that the use of the tidy function here was unnecessary - it was used just to produce nicely formatted output for the rendered markdown doc.

```
cohen <- read.csv("cohen.csv")
library(broom)
cohen %>%
  do(broom::tidy(lm(salary ~ pubs + cits, .)))
```

```
# A tibble: 3 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)   40493.    2505.      16.2  2.44e-23
2 pubs            252.      72.9      3.45 1.03e- 3
3 cits            242.      59.5      4.07 1.40e- 4
```

7 Conclusions

Whenever possible, the most efficient method of avoiding the "dataframe$varname" specification is to use the "data" argument to specify a data frame within a function. For those functions that don't permit a "data" argument, the `with` function is a good alternative.

Use `attach` only when you are certain that it will be the only data frame used in that code file.

Tidyverse methods (pipes and **dplyr** tools) can be useful when the code has more complexity than the step by step approach to R that we have been emphasizing. Is it "easier to learn" and "less complicated"? Not sure.

8 Documentation for Reproducibility

R software products such as this markdown document should be simple to reproduce, if the code is available. But it is also important to document the exact versions of the R installation, the OS, and the R packages in place when the document is created.

```
sessionInfo()
```

```
R version 4.4.2 (2024-10-31 ucrt)
Platform: x86_64-w64-mingw32/x64
Running under: Windows 11 x64 (build 22631)

Matrix products: default
```

```
locale:
[1] LC_COLLATE=English_United States.utf8
[2] LC_CTYPE=English_United States.utf8
[3] LC_MONETARY=English_United States.utf8
[4] LC_NUMERIC=C
[5] LC_TIME=English_United States.utf8

time zone: America/New_York
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] broom_1.0.7 dplyr_1.1.4

loaded via a namespace (and not attached):
 [1] vctrs_0.6.5       cli_3.6.3         gt_0.11.1         knitr_1.49
 [5] rlang_1.1.4       xfun_0.50         purrr_1.0.2       generics_0.1.3
 [9] jsonlite_1.8.9    glue_1.8.0        backports_1.5.0   htmltools_0.5.8.1
[13] rmarkdown_2.29    evaluate_1.0.3    tibble_3.2.1      fastmap_1.2.0
[17] yaml_2.3.10       lifecycle_1.0.4   compiler_4.4.2    pkgconfig_2.0.3
[21] tidyr_1.3.1       rstudioapi_0.17.1 digest_0.6.37     R6_2.5.1
[25] tidyselect_1.2.1  utf8_1.2.4        pillar_1.10.1     magrittr_2.0.3
[29] tools_4.4.2       withr_3.0.2       xml2_1.3.6
```