

# Coding Style

Harris Coding Camp

Summer 2023

## I. Intro to Coding Style<sup>1</sup>

### Object names

“There are only two hard things in Computer Science: cache invalidation and naming things.”

— Phil Karlton

Variable and function names should use only lowercase letters, numbers, and `_`. Use underscores (`_`) (so called snake case) to separate words within a name.

```
# Good
day_one
day_1
```

```
# Bad
DayOne
dayone
```

Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful (this is not easy!).

```
# Good
day_one
```

```
# Bad
first_day_of_the_month
djm1
```

Where possible, avoid re-using names of common functions and variables. This will cause confusion for the readers of your code.

```
# Bad
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

---

<sup>1</sup>The great content is more or less copy-and-pasted from <https://style.tidyverse.org/syntax.html>

## Spacing

### Commas

Always put a space after a comma, never before, just like in regular English.

```
# Good
x[, 1]

# Bad
x[,1]
x[ ,1]
x[ , 1]
```

### Parentheses

Do not put spaces inside or outside parentheses for regular function calls.

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

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

Place a space after () used for function arguments:

```
# Good
function(x) {}

# Bad
function (x) {}
function(x){}
```

### Infix operators

Most infix operators (==, +, -, <-, etc.) should always be surrounded by spaces:

```
# Good
height <- (feet * 12) + inches
mean(x, na.rm = TRUE)

# Bad
height<-(feet*12)+inches
mean(x, na.rm=TRUE)
```

There are a few exceptions, which should never be surrounded by spaces:

- The operators with [high precedence][syntax]: ::, :::, \$, @, [, [[, ^, unary -, unary +, and :.

```

# Good
sqrt(x^2 + y^2)
df$z
x <- 1:10

# Bad
sqrt(x ^ 2 + y ^ 2)
df $ z
x <- 1 : 10

```

## Extra spaces

Adding extra spaces is ok if it improves alignment of = or <-.

```

# Good
list(
  total = a + b + c,
  mean  = (a + b + c) / n
)

# Also fine
list(
  total = a + b + c,
  mean  = (a + b + c) / n
)

```

Do not add extra spaces to places where space is not usually allowed.

## Control flow

### Code blocks

Curly braces, {}, define the most important hierarchy of R code. To make this hierarchy easy to see:

- { should be the last character on the line. Related code (e.g., an if clause, a function declaration, a trailing comma, ...) must be on the same line as the opening brace.
- The contents should be indented by two spaces.
- } should be the first character on the line.
- If used, else should be on the same line as }.

```

# Good
if (y < 0 && debug) {
  message("y is negative")
}

if (y == 0) {
  if (x > 0) {
    log(x)
  } else {

```

```

    message("x is negative or zero")
}
} else {
    y^x
}

# Bad
if (y < 0 && debug) {
    message("Y is negative")
}

if (y == 0)
{
    if (x > 0) {
        log(x)
    } else {
        message("x is negative or zero")
    }
} else { y ^ x }

```

## Long lines

Strive to limit your code to 80 characters per line. This fits comfortably on a printed page with a reasonably sized font. If you find yourself running out of room, this is a good indication that you should encapsulate some of the work in a separate function.

If a function call is too long to fit on a single line, use one line each for the function name, each argument, and the closing `)`. This makes the code easier to read and to change later.

```

# Good
do_something_very_complicated(
    something = "that",
    requires = many,
    arguments = "some of which may be long"
)

# Bad
do_something_very_complicated("that", requires, many, arguments,
                              "some of which may be long"
)

```

You may also place several arguments on the same line if they are closely related to each other, e.g., strings in calls to `paste()` or `stop()`. When building strings, where possible match one line of code to one line of output.

```

# Good
paste0(
    "Requirement: ", requires, "\n",
    "Result: ", result, "\n"
)

# Bad
paste0(

```

```
"Requirement: ", requires,  
"\n", "Result: ",  
result, "\n")
```

## Semicolons

Don't put ; at the end of a line, and don't use ; to put multiple commands on one line.

## Assignment

Use <-, not =, for assignment.

```
# Good; most people use it  
x <- 5  
  
# Okay, but not recommended  
x = 5
```

## Data

### Logical vectors

Prefer TRUE and FALSE over T and F.

### Quotes

Use " and not ' for quoting text. The only exception is when the text already contains double quotes and no single quotes.

```
# Good  
"Text"  
'Text with "quotes"'  
'<a href="http://style.tidyverse.org">A link</a>'  
  
# Bad  
'Text'  
'Text with "double" and \'single\' quotes'
```

## Comments

Each line of a comment should begin with the comment symbol and a single space: #

## Function calls

### Named arguments

A function's arguments typically fall into two broad categories: one supplies the **data** to compute on; the other controls the **details** of computation. When you call a function, you typically omit the names of data

arguments, because they are used so commonly. If you override the default value of an argument, use the full name:

```
# Good
mean(1:10, na.rm = TRUE)

# Bad
mean(x = 1:10, , FALSE)
mean(, TRUE, x = c(1:10, NA))
```

## Pipes

### Introduction

Use `|>` to emphasise a sequence of actions, rather than the object that the actions are being performed on. Avoid using the pipe when:

- You need to manipulate more than one object at a time. Reserve pipes for a sequence of steps applied to one primary object.
- There are meaningful intermediate objects that could be given informative names.

### Whitespace

`|>` should always have a space before it, and should usually be followed by a new line. After the first step, each line should be indented by two spaces. This structure makes it easier to add new steps (or rearrange existing steps) and harder to overlook a step.

```
# Good
iris |>
  group_by(Species) |>
  summarize_if(is.numeric, mean) |>
  ungroup() |>
  pivot_longer(-Species, names_to = "measure", values_to = "value") |>
  arrange(value)

# Bad
iris |> group_by(Species) |> summarize_all(mean) %>%
ungroup |> pivot_longer(-Species, names_to = "measure", values_to = "value") |>
arrange(value)
```

### Long lines

If the arguments to a function don't all fit on one line, put each argument on its own line and indent:

```
iris |>
  group_by(Species) |>
  summarise(
    Sepal.Length = mean(Sepal.Length),
```

```
Sepal.Width = mean(Sepal.Width),  
Species = n_distinct(Species)  
)
```

## Short pipes

A one-step pipe can stay on one line, but unless you plan to expand it later on, you should consider rewriting it to a regular function call.

```
# Good  
iris |> arrange(Species)  
  
iris |>  
  arrange(Species)  
  
arrange(iris, Species)
```

## Assignment

- Variable name and assignment on separate lines:

```
iris_long <-  
  iris |>  
  pivot_longer(-Species, names_to = "measure", values_to = "value") |>  
  arrange(-value)
```

- Variable name and assignment on the same line:

```
iris_long <- iris |>  
  pivot_longer(-Species, names_to = "measure", values_to = "value") |>  
  arrange(-value)
```