Accelerated TA session 8: Style and review

Accelerated Coding Lab

2022-09-08

I. Intro to Coding Style[1]

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)
```

[1]The great content is copy and paste 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 `:`.

```r
# 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 `<-`.

```r
# 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 `}`.

```r
# 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() %>%
  gather(measure, value, -Species) %>%
  arrange(value)

# Bad
iris %>% group_by(Species) %>% summarize_all(mean) %>%
ungroup %>% gather(measure, value, -Species) %>%
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 %>%
    gather(measure, value, -Species) %>%
    arrange(-value)
  ```

- Variable name and assignment on the same line:

  ```
  iris_long <- iris %>%
    gather(measure, value, -Species) %>%
    arrange(-value)
  ```

Review

Doing math with vectors

T-tests are used to determine if two sample means are equal. The formula for a t-score is:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

where x_i is the mean of the first or second set of data, s_i is the sample standard deviation of the first or second set of data, and n_i is the sample size of the ith set of data.

We'll first create two data sets of random numbers following a normal distribution:

```r
set.seed(1)
data_1 <- rnorm(1000, 3)
data_2 <- rnorm(100, 2)
```

1. What built-in functions do you need to calculate the variables in the formula for each `data_`*i*?
2. Calculate the t-score using the formula above?

What did you get for the t-score? Hint: You should have gotten 9.243, if not, double check your code!

Remark: As a rule of thumb, t-scores close to 0 imply that the means are not statistically distinguishable, and large t-scores (e.g. t > 3) imply the data have different means. You'll learn more in stats 1!

Data manipulation with `[` and `dplyr`

Using `storms` which comes with `dplyr`. Do the following in base R and dplyr.

1. What category storms have a non-zero value for `hurricane_force_diameter`? (Once you subset the data you can use `distinct()` (tidyverse) or `unique()` base R to find the answer.)
2. Find all data from storms named "Ana".
3. What is the `maximum category` for storms named "Ana"? Have we ever had a hurricane named "Ana"?
4. Collect the columns that relate to the time and location of the storms.
5. Get the columns that measure the "force diameter" of tropical storms and hurricanes.
6. Create a column that is called `ratio` that is the ratio of pressure to wind.
7. What is the mean and sd of `ratio` for category 5 storms?
8. What is the mean and sd of `ratio` for category 1 storms?
9. What is the first year `tropicalstorm_force_diameter` is not `NA`?

`ifelse or case_when`

1. Answer the question: What is the first year `tropicalstorm_force_diameter` is not `NA` using sorting and no filtering. You'll notice that when we sort `NA` goes to the end of the line / bottom of the data. This motivates creating an indicator column that is 1 if the data is missing and 0 otherwise.
2. Add a column to the data called `season` that takes names "winter", "spring", "summer" or "fall" depending on the month of the year. (You can pick the cut offs as you see fit.)
3. *challenge* using `case_when` in `mutate` make your season indicator depend on the month and day of the year. (E.g. Winter is roughly December 21st to March 20th.)