Style Guide

Thanks to Professor Gerard

Learning Objectives

- Coding Style
- tidyverse style guide
- Google style guide
- Bioconductor Style Guide

Style Guides

- Each organization has a style guide on how code should be formatted that you should adhere to.

- When everyone on a project uses a consistent style, it makes code easier to read and understand, and it makes collaboration easier and faster.

- There are lots of style guides (see links in the Learning Objectives). This document contains the style guide for our class.

- This style guide is obviously opinionated, and others have their own thoughts (which is perfectly fine!). The important thing is consistency among collaborators.

- We will mostly follow the tidyverse style guide. Below I place some points of emphasis and note differences.

- I expect you to follow this style guide in all homeworks and assignments.

File Names

- File and folder names should only have

    1. Letters
    2. Numbers
    3. Underscores (`_`).
    4. Possibly dashes `-`. But these are discouraged.

- In particular, never use spaces or periods in a file name.
- Capital letters are discouraged. You should work almost entirely with lower-case letters.
- Always begin a file name with a lower-case letter.
- Exceptions to this are:

    - Hidden files/folders begin with a period `.`
    - Standard/Required files, such as `NAMESPACE`, `README.md`, etc…

- R scripts should end in `.R` (not `.r`).
- R markdown files should end in `.Rmd`.

Syntax

Names

- Only use lower-case snake_case.

    - Good

```
        red_apple
```

```
- Bad
```

```
        Red_apple
        red.apple
        redApple
        RedApple
```

- Variables should be nouns and functions should be verbs

- Never use single letters as variables/functions
    - Good:

```
num_sim <- 10
```

```
- Bad
```

```
simulate <- 10 ## verb
x <- 10 ## single letter
```

- **Exceptions**: Some letters are standard. Such as `n` for the sample size in `rnorm()`, `runif()`, etc...

Commas

- Always put a space after a comma, not before (like English).
    - Good:

```
mat[1, ]
```

- Bad:

```{r, eval = FALSE}
mat[1 ,]
mat[1 , ]
mat[1,]
```

Parentheses

- Don't put a space in or around parentheses for functions.
    - Good:

```
mean(x)
```

```
- Bad:
```

```
mean (x)
mean(x )
```

- Put spaces around parentheses for `if` statements, and `for` and `while` loops.
  - Good:

```
if (x) {

}
```

```
- Bad:
```

```
if(x){

}
```

- Put a space only after `()` for function creations.
  - Good:

```
sim <- function(x) {
}
```

```
- Bad:
```

```
sim <- function (x) {
}

sim <- function(x){
}
```

Curley Braces

- Whenever you use curly braces `{}`, the opening brace should be the last character on a line, and the closing brace should be the first character on a line.
  - Good:

```
if (condition) {
  dostuff()
}
```

- Bad

```
if (condition)
{
  dostuff()
}

if (condition) {
  dostuff() }
```

`if-else`

- `else` statements should be on the same line as a closing brace.
  - Good:

```
if (condition) {

} else if (condition2) {

} else {

}
```

- Only use `ifelse()` where vectorization is important. If `condition` should be length 1, then use full `if-else` statements.
- In a `if-then` statement, use `||` or `&&`, not `|` or `&`, since the latter two vectorize operations.

Infix Characters

- An **infix** operator is one where arguments on both sides of it are used in a function. The alternative is **prefix** notation. Compare

```
5 + 10 ## infix notation
```

```
[1] 15
```

```r
`+`(5, 10) ## prefix notation
```

```
[1] 15
```

- Put spaces around all infix characters `==`, `+`, `-`, `*`, `/`, `^`, `|>`, etc…
  - Good:

```r
x + 10
```

```
- Bad:
```

```r
x+10
x+ 10
x +10
```

- Exceptions: `::`, `:::`, `$`, `@`, `[`, `[[`, unary `-`, unary `+`, `:`, and `?`.
  - E.g. do `ggplot2::qplot()` or `-1`, not `ggplot2 :: qplot()` and `- 1`

Code Length

- No lines should be greater than 80 characters.
- To get a vertical line displaying the code length, in R studio go to "Tools > Global Options… > Code > Display". Make sure "Show margin" is checked with "80" in the text box.
- If a function call/definition is too long, break up arguments on new line.

```r
this <- is_a_very_long_function_call(
  that = "is",
  broken = "up",
  into = "many",
  indented = "lines",
  that = "are",
  easier = "to",
  read = NULL
)
```

Other things

- Always use `<-` for assignment, not `=`.
- Always use `"` for strings, not `'`.
- Always use `TRUE` or `FALSE`, not `T` or `F`
  - `T` and `F` are aliases for `TRUE` and `FALSE`, and so may be overwritten by the user, which is scary.
- Don't include non-ASCII characters in your code.
  - ASCII characters are lower case letters (`a` through `z`), upper case letters (`A` through `Z`), digits (`0` through `9`), and common punctuation.
  - Including non-ASCII characters will give you a CRAN note.
  - Non-ASCII characters usually show up when you copy and paste from the web. E.g. the following look normal but are non-ASCII (and are all different):
    * En Dash: "–"
    * Em Dash: "—"
    * Horizontal Bar: " "
    * En Quad: " "
    * Em Quad: " "
    * En Space: " "
    * Em Space: " "
  - If you accidentally include such characters, you can find them with

```r
          tools::showNonASCIIfile()
```

Functions

Function Argument Length

- If you have a lot of arguments, indent the arguments on new lines.

```r
run_me <- function(this,
                   is,
                   a = "lot",
                   of = "arguments",
                   that = "are longer than 80 characters") {
}
```

Function Length

- You should break up your functions into discrete tasks.
  - Reduces duplicating code, so less prone to bugs.
  - Allows you to think more modularly about tasks, which makes code easier to reason about.
  - Makes it easier to combine code in new ways.
- To force you to do this, make all functions be less than 50 lines. This is what Bioconductor does.

Explicit returns

- In R, the last value evaluated in a function will be implicitly returned. I think this is bad practice since it makes it harder to reason about what R is returning. So always include a `return()` statement. **Never** do

```
add_two <- function(x, y) {
  x + y
}
```

```
**Always** do
```

```
add_two <- function(x, y) {
  return(x + y)
}
```

Importing

- **Never** use the `@import` tag in a package to bring all of a package's exported functions into the `NAMESPACE`. This creates too much risk for name collision.
- In a package, never import functions, always type the package where the function came from. This makes it easier to reason about namespaces. **Never** do

```
#' @importFrom ggplot2 qplot
plot_red <- function(x, y) {
  qplot(x, y, color = I("red"))
}
```

```
**Always** do
```

```
plot_red <- function(x, y) {
  ggplot2::qplot(x, y, color = I("red"))
}
```

- **Exceptions**:
  - You will have to import infix functions (surrounded by percent signs). Such as ::: {.cell layout-align="center"}

    ```
    #' @importFrom magrittr %>%
    #' @importFrom foreach %dopar%
    ```

    :::
  - There is a small performance penalty for using `::` (about 5 µs). So import a function if you are iterating it ∼ million times, and each iteration takes on the order of 1 ns.

Order of Arguments

- Always place arguments with defaults after arguments without defaults.
- Good:

```
function(arg1, arg2, arg3 = NULL) {

}
```

- Bad:

```
function(arg1, arg3 = NULL, arg2) {

}
```

`lintr`

- The `lintr` package will check many coding issues. Try running the following in the top directory of your package.

```
lintr::lint_package()
```