# CODING BEST PRACTICES

*THANKS to Chloe Bracis for her slides!!*

# Writing maintainable code

# NAMING CONVENTIONS

- There are 5 naming conventions to choose from:

  **alllowercase**: e.g. adjustcolor

  **period.separated**: e.g. plot.new

  **underscore_separated**: e.g. numeric_version

  **lowerCamelCase**: e.g. addTaskCallback

  **UpperCamelCase**: e.g. SignatureMethod

- Whatever you choose, stick with it and be consistent (naming R files, variables, functions)

# FILE NAMES

- Use a meaningful and descriptive names
- End in .R
- Avoid spaces in the name

- If files need to be run in sequence, prefix them with numbers:
- 01_download_clean_and_explore_buffalos.R
- 02_variogram_and_akde_per_buffalo.R
- 03_SSF_per_buffalo.R
- 04_plots_for_MS_buffalo.R

GOOD:

```
get_Buffalo_Data.R
getBuffaloData.R
```

BAD:

```
foo.R
get Buffalo Data.R
89317240934735.r
```

# FILE NAMES

Do you have a folder like this?

Myscript.R
Myscript2.R
Myscript_old.R
Myscript_old2.R
Myscript_trysp.R
Myscript_.R
Myscript4.R
Myscript_usethisone.R

# VARIABLES & FUNCTION NAMES

- Strive for names that are concise and meaningful (this is not easy!). Spend the time to come up with good variable names.
- Generally, variable names should be nouns and function names should be verbs.
- Long names are ok, you don't have to type them (tab in Rstudio)
- Avoid using names of existing functions and variables!

GOOD:

```
pcaEmbedding  # variable
getPcaEmbedding <- function() # function
```

BAD:

```
pcaembedding  # variable
getpcaembedding <- function() # function
pe  # variable
gpe <- function()  # function
temp1
m3
```

BAD:

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

# COMMENTS

- **Comment incessantly.** Comments should not take the place of clean and clear code, but can help explain why you might have done certain steps or used certain functions. Commenting is just as important for you as it is for any one else reading your code - as they say, your worst collaborator is you from 6 months ago.
- Use comments to document the intention of the code
- Document the why not the what (i.e. intent, behavior, and purpose of code), comments should explain at a higher level of abstraction the programmer’s intention
- Use good names instead of just relying on comments
- Do not use comments that repeat the code
- Don’t assume you will remember anything about the code when you look at it later
- Write comments in English (you never know who you will end up sharing it with)

- If you follow your comment with four dashes (# ----) RStudio will enable code folding until the next instance of this.

# SYNTAX

- **Assignments**
  - use **<-** not **=**

GOOD:

```
x <- 23
y <- 12
```

BAD:

```
x = 23
23 -> x
```

- **Line Length**
  - Strive to limit your code to 80 characters per line
  - (Softwrap option in RStudio)

- **Indentation**
  - Use two spaces to indent code. Never mix tabs and spaces. RStudio can automatically convert the tab character to spaces (see Tools -> Global options -> Code).
  - you can always force the RStudio indent by highlighting code and using `CTRL + I`.

GOOD:

```
if(!exists("x")) {
  x = c(3, 5)
  y = x[2]
}
```

BAD:

```
if(!exists("x")){
x = c(3, 5)
y = x[2]
}
```

# SYNTAX

- **Spacing**
  - Place spaces around all binary operators such as **=**, **+**, **-**, **<-**, etc.
  - Place a space before the **(**, except in a function call
  - Use space after a comma
  - There's a small exception to this rule: **:**, **::** and **:::** don't need spaces around them.
  - The shortcut *Ctrl+Shift+A* will reformat the code, adding spaces for maximum readability.

GOOD

```
average <- mean(feet / 12 + inches, na.rm = TRUE)
```

BAD

```
average<-mean (feet/12+inches,na.rm=TRUE)
```

GOOD

```
x <- 1:10
base::get
```

BAD

```
x <- 1 : 10
base :: get
```

GOOD

```
if (debug) do(x)
diamonds[5, ]
```

BAD

```
if ( debug ) do(x)  # No spaces around debug
x[1,]   # Needs a space after the comma
x[1 ,]  # Space goes after comma not before
```

# SYNTAX

- **Curly Braces**
  - Do not place **{** on its own line
  - Place **}** on its own line unless followed by else
  - Surround else statements with curly braces

GOOD

```
if (a < d) {
    a <- (b + c) * d
} else {
    a <- d
}
```

BAD

```
if (a < d)
{
    a <- (b + c) * d
} else a <- d
```

## miscellaneous

- **Use** library() **NOT** require() **when loading packages in scripts.**

  library will throw an error if it the library is not already installed, but require will silently fail. This means you get a mysterious failure later on when functions within a package are not available and you thought they were.

- **take advantage of RStudio's autocomplete features** (and/or copy/paste).

  Typing mistakes are often a reason that code doesn't work (e.g. a lowercase letter that really should have been uppercase), and using auto-complete or copy/paste reduces the prevalence of these mistakes. To use auto-complete, start typing something and hit the Tab button on your keyboard. You should see options start to pop up.

# REFERENCES

(order of appearance does not reflect any kind of order of importance)


- R Style Guide: https://jef.works/R-style-guide/
- Google's R Style Guide: https://google.github.io/styleguide/Rguide.xml#comments
- Advanced R, style guide: http://adv-r.had.co.nz/Style.html
- Efficient R programming, Coding style: https://csgillespie.github.io/efficientR/coding-style.html
- R Code – Best practices: https://www.r-bloggers.com/r-code-best-practices/
- Beyond Basic R - Introduction and Best Practices: https://owi.usgs.gov/blog/intro-best-practices/
- Best Practices for Writing R Code: https://swcarpentry.github.io/r-novice-inflammation/06-best-practices-R/

- A Guide to Reproducible Code in Ecology and Evolution (British Ecological Society.): https://www.britishecologicalsociety.org/wp-content/uploads/2017/12/guide-to-reproducible-code.pdf

# Organizing code

# Structuring a script

- **Keep track of who wrote your code and its intended purpose**
  - Starting code with annotated description of what the code does will save you or someone else a lot of time and effort when trying to understand what a particular script does.

```
# This is code to replicate the analyses and figures from my 2014 Science
# paper. Code developed by Sarah Supp, Tracy Teal, and Jon Borelli
```

- **Be explicit about the requirements and dependencies of your code**
  - Loading all of the packages that will be necessary to run your code.
  - Another way you can be explicit about the requirements of your code and improve it's reproducibility is to limit the "hard-coding" of the input and output files for your script. If your code will read in data from a file, define a variable early in your code that stores the path to that file.

This is preferable to:

```
input_file <- "data/data.csv"
output_file <- "data/results.csv"

# read input
input_data <- read.csv(input_file)
# get number of samples in data
sample_number <- nrow(input_data)
# generate results
results <- some_other_function(input_file, sample_number)
# write results
write.table(results, output_file)
```

to this:

```
# check
input_data <- read.csv("data/data.csv")
# get number of samples in data
sample_number <- nrow(input_data)
# generate results
results <- some_other_function("data/data.csv",
sample_number)
# write results
write.table("data/results.csv", output_file)
```

# Organizing code

**Write functions**

- DRY: Don’t repeat yourself,
  - make a function
- Each function should do one thing
- Related functions in a file
  - source(“moveFunctions.R”)
- Build up a library of functions

**Write scripts**

- Reproducible code (scientific method)
- Everything in a script
- Save the results of a function call as an object, rather than typing in the numeric value
- Set.seed, to make code reproducible

# Organizing code

**Use projects**

- Feature of Rstudio
- Organize related files together (R code, data, output)
- One project per ‘analysis’
- Organizing advice: https://nicercode.github.io/blog/2013-04-05-projects/
-
- **Don’t save your workspace**
  - Rerun script or save output as .csv or .rdata
- Be careful what you have in memory
- Don’t use the same name for everything
  - Especially temp!!
  -
- If possible, keep track of `sessionInfo()` somewhere in your project folder.
  - it captures all of the packages used in the current project. If a newer version of a package changes the way a function behaves, you can always go back and reinstall the version that worked (Note: At least on CRAN, all older versions of packages are permanently archived).
- `renv`: library for R dependency management https://rstudio.github.io/renv/articles/renv.html

# Version control

- Develop your code using version control and frequent updates! E.g. GitHub, GitLab
- What can you do with source control?
  - Compare latest to previous version (what exactly have I changed today)
  - Similarly, it's easy to undo back to the last version (oops, my latest change broke something)
  - Go back or compare to any previous version
  - Keep track of what version given to someone or published
  - Share code easily

- Here some instructions:
- https://www.r-bloggers.com/rstudio-and-github/
- https://masterr.org/da/on-git-github-and-gitlab/
- https://r-bio.github.io/intro-git-rstudio/
- https://www.britishecologicalsociety.org/wp-content/uploads/2017/12/guide-to-reproducible-code.pdf

# Performance

# Memory issues

- R can run into memory issues. A good practice when running long lines of computationally intensive code is to remove temporary objects after they have served their purpose. However, sometimes, R will not clean up unused memory for a while after you delete objects. You can force R to tidy up its memory by using **gc()**.

```
# Sample dataset of 1000 rows
interim_object <- data.frame(rep(1:100, 10),
                             rep(101:200, 10),
                             rep(201:300, 10))
object.size(interim_object) # Reports the memory size allocated to the object
rm("interim_object") # Removes only the object itself and not necessarily the memory allotted to it
gc() # Force R to release memory it is no longer using
ls() # Lists all the objects in your current workspace
rm(list = ls()) # If you want to delete all the objects in the workspace and start with a clean slate
```

# Testing code

- Generally always test any new piece of code, specially functions on a small subset of your data, just to verify it does what you wanted to do, without having to wait for ever or crashing the computer or R

- Proper testing is done with testthat, for details see:
- R packages, testing: http://r-pkgs.had.co.nz/tests.html
- testthat: Get Started with Testingby Hadley by Wickham: https://journal.r-project.org/archive/2011/RJ-2011-002/RJ-2011-002.pdf
- Example of unit testing R code with testthat: https://www.johndcook.com/blog/2013/06/12/example-of-unit-testing-r-code-with-testthat/

# Code performance

- Take advantage of multiple processors, parallelize your code
- Quick Intro to Parallel Computing in R: https://nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html
- How-to go parallel in R – basics + tips: https://www.r-bloggers.com/how-to-go-parallel-in-r-basics-tips/
- An introduction to multidplyr: https://github.com/hadley/multidplyr/blob/master/vignettes/multidplyr.md
- Running code in parallel: https://jstaf.github.io/hpc-r/parallel/
- A brief foray into parallel processing with R: https://beckmw.wordpress.com/2014/01/21/a-brief-foray-into-parallel-processing-with-r/

## Still too slow...

- Simulations: consider using a language other than R (Python, Java, ...)
- **Optimize only if time saved greatly outweighs time to implement**
- Intuition is often wrong, optimize only after Profiling*

- *However, there are a few common issues specific to R
  - It's faster to vectorize than loop (e.g., lappy)
  - Calculating with matrices is faster than with data.frames
  - Pre-allocating storage is faster then appending many times

# Profiling R code

- Test which operation within your code is taking the longest
- Find more info at: Profiling R code: https://www.r-bloggers.com/profiling-r-code/

```
Rprof()
data <- array(rnorm(200 * 200 * 100),
              dim = c(200, 200, 100))
dataSd <- apply(data, 3, sd)
Rprof(NULL)
summaryRprof()
```

```
$by.self
                self.time self.pct total.time total.pct
"rnorm"              0.22    68.75       0.22     68.75
"array"              0.04    12.50       0.26     81.25
"aperm.default"      0.04    12.50       0.04     12.50
"var"                0.02     6.25       0.02      6.25

$by.total
                total.time total.pct self.time self.pct
"array"               0.26     81.25      0.04    12.50
"rnorm"               0.22     68.75      0.22    68.75
"apply"               0.10     31.25      0.00     0.00
"aperm.default"       0.04     12.50      0.04    12.50
"aperm"               0.04     12.50      0.00     0.00
"var"                 0.02      6.25      0.02     6.25
"FUN"                 0.02      6.25      0.00     0.00

$sample.interval
[1] 0.02

$sampling.time
[1] 0.32
```

# microbenchmark

- Check how long each function is taking

```
x <- rnorm(10000)
f1 <- mean
f2 <- function(x) {mean(x)}
f3 <- function(x) {sum(x) / length(x)}
f4 <- function(x) {sum(x) / 1000}
f5 <- function(x) {sum(x * .001)}
microbenchmark(f1(x), f2(x), f3(x), f4(x), f5(x), times=1000)
```

```
Unit: microseconds
  expr    min      lq     mean  median      uq      max neval
 f1(x) 19.960 20.3275 21.34490 20.6495 21.1385   45.295  1000
 f2(x) 20.467 20.8020 22.92714 21.1585 21.6140 1068.481  1000
 f3(x)  9.491  9.6910 11.52805  9.8125  9.9660 1204.806  1000
 f4(x)  9.421  9.5930 11.45184  9.6955  9.8510 1119.461  1000
 f5(x) 16.011 16.9445 23.86270 17.3685 18.2975 4120.151  1000
```

# Making code more efficient

- Advanced R, performance: http://adv-r.had.co.nz/Performance.html
- Advanced R, optimizing code: http://adv-r.had.co.nz/Profiling.html
- Efficient R programming, Efficient optimisation: https://csgillespie.github.io/efficientR/performance.html
- Strategies to Speedup R Code: https://www.r-bloggers.com/strategies-to-speedup-r-code/