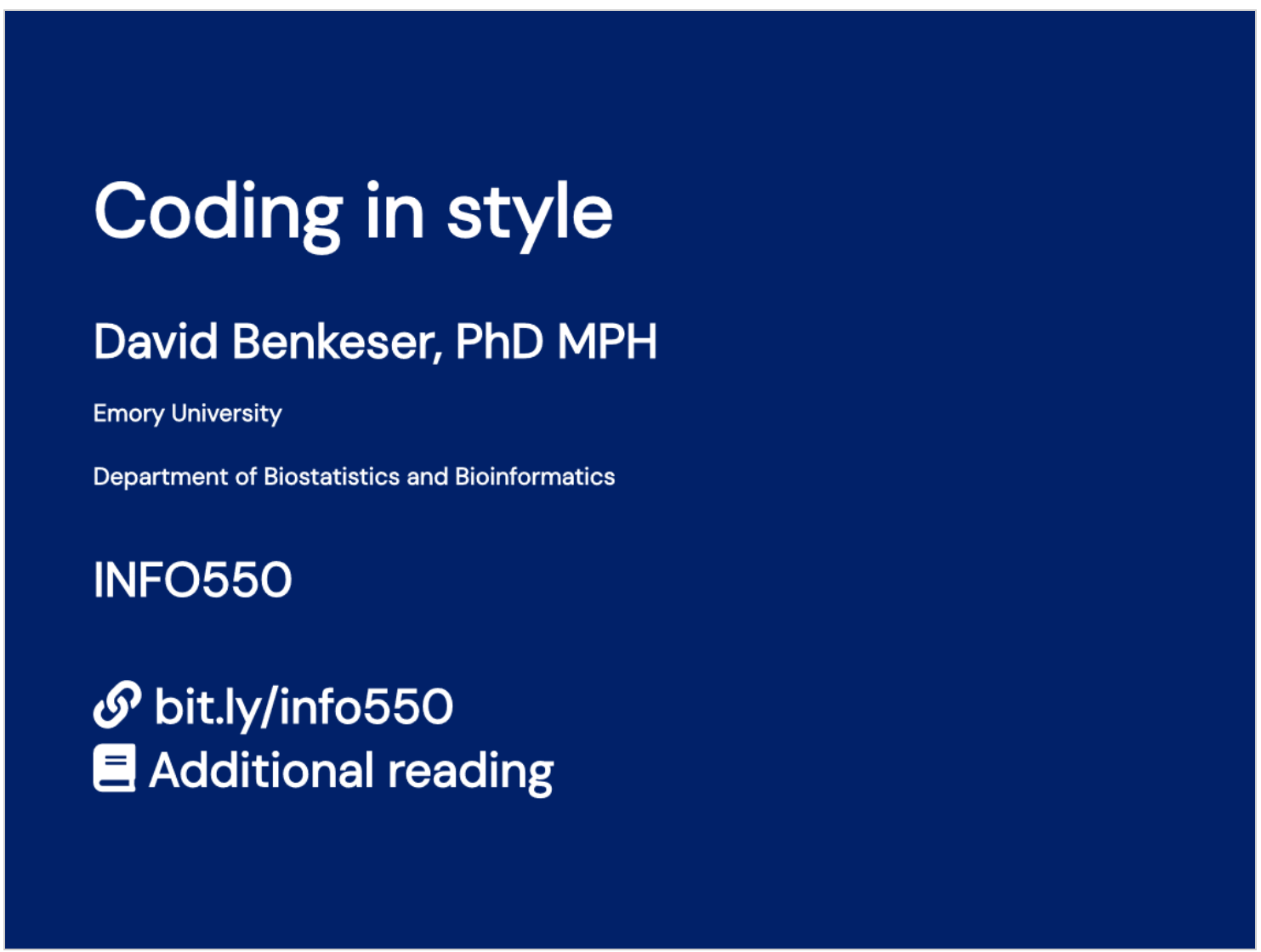
Module 9 Pre-Read - Coding practices

Wednesday, June 5, 2024 03:35

https://benkeser.github.io/info550/lectures/08_codestyle/codestyle.html#6

Coding in style



Saying [researchers] should spend more time thinking about the way they write code would be like telling a novelist that she should spend more time thinking about how best to use Microsoft Word. Sure, there are people who take whole courses in how to change fonts or do mail merge, but anyone moderately clever just opens the thing up and figures out how it works along the way.

This manual began with a growing sense that our own version of this self-taught seat-of-the-pants approach to computing was hitting its limits.

Gentzkow and Shapiro

Code and Data for the Social Sciences: A Practitioner's Guide

Principles for scientific coding

In this order:

1. Code that works.
2. Code that is reproducible.
3. Code that is readable.
4. Code that is generalizable.
5. Code that is efficient.

A **minimal standard** for scientific computing is 1–3.

Advice for beginner coders

Test code before relying on it.

It's OK to **copy/paste code** from Stack Overflow, but make sure you **understand how it works**.

- Run line by line and see what each does.
- Change the code and see if it behaves as expected.

Stakes for copy/paste can be high!

- Incorrect analyses.
- Expensive (inadvertent) cloud computing.

Advice for intermediate coders

Premature optimization is the root of all evil.

Getting code correct AND readable is most important.

- Make your code more efficient later.
- After a paper is submitted for review?

Remember: you don't get bonus points for code that "looks impressive".

Think before you code

```
# Averaging over drtmle option:
#   Instantiate empty lists
#   Note: length = 1 if n_SL = 1 or "drtmle" not in avg_over):
#     nuisance_drtmle
#     nuisance_aiptw_c
#     ic_drtmle
#     QnMod, gnMod, QrnMod, grnMod
#     drtmle -- eventually avg over to get final point estimates
#     aiptw_c -- eventually avg over
#     tmle -- eventually avg over
#     aiptw -- eventually avg over
#     gcomp -- eventually avg over
#     validRows -- should this be added to the output?
#   Wrap everything into a big for loop and add indexes to the
#   objects above
#   Add code below for loop to do aggregation

# For introducing cross-validated standard errors:
#   if se_cv = "full", but cvFolds = 1
#   need to modify make_validRows recognize this situation?
#   Or could modify estimateQ/g directly and have the functions
#   call themselves if se_cv = "full" and cvFolds == 1
#   or probably better to add an if statement that would call the
#   estimateQ again outside. Yes, that's probably the way to go
```

Don't repeat yourself

Don't repeat yourself (DRY) is a fundamental concept in programming.

- *Ruthlessly eliminate duplication*, Wilson et al

For example, variables `score1=1`, `score2=2`, `score3=3` → `score=list(1,2,3)`.

If you write the same code more than once, it should be a function.

- Break large tasks into smaller calls to functions.
- Give functions (everything, really) meaningful names.
  - self-documenting code
  - use tab-completion

Functions in R

If you don't know how functions work in R, learn about them now.

- Software carpentry: Creating R Functions
- R for Data Science: Functions
- DataCamp: A Tutorial on Using Functions in R!

Generalize... some

Write code **a bit more general** than your data or specific task.

- Don't assume particular dimensions.
- Don't forget about missing values (even if *your* data have none).

But **don't try to handle every case.**

- Try to anticipate what you might be asked for, but don't prepare for every possibility.
- If you think of **extreme cases** where code will break, **document them**.

Use **function arguments** to handle different cases.

- Don't assume particular file names.
- Don't assume particular tuning parameters.
- Don't assume particular regression formulas.

No magic numbers

There are **many decision points** in an analysis. Give them a name!

- How many **bootstrap samples**?
  - `nboot = 1e3`
- What **tolerance for convergence** of an algorithm?
  - `tol_covergence = 1e-3`
- What **threshold** for excluding missing data?
  - `tol_missingness = 3`

Even better, include them as an **argument to a function** (with default values and documentation, as needed)!

- `get_bootstrap_ci <- function(..., nboot = 1e3)`
- `my_algorithm <- function(..., tol_convergence = 1e-3)`
- `rm_missing_data <- function(..., tol_missingness = 3)`

Other guidelines

- **Indent**!
  - 2 or 4 spaces (join the debate)
  - Tabs can get nasty across systems. In my experience, Windows is pretty dumb about this.
- Use **white space**!
  - After commas, operators
- Use `{}` and `()` to **avoid ambiguity**.
- Keep **lines short**!
  - Rule of thumb is 72 or 80 characters.
  - Most text editors have settings to help with this.

Other guidelines

```r
# move values above/below quantiles to those quantiles
winsorize<-function(vec,q=0.006){
lohi<-quantile(vec,c(q,1-q),na.rm=TRUE)
if(diff(lohi)<0)lohi<-rev(lohi)
vec[!is.na(vec)&vec<lohi[1]]<-lohi[1]
vec[!is.na(vec)&vec>lohi[2]]<-lohi[2]
vec}
```

```r
# move values above/below quantiles to those quantiles
winsorize <- function(vec, q = 0.006){
  lohi <- quantile(vec, c(q, 1 - q), na.rm = TRUE)
  if(diff(lohi) < 0){
    lohi <- rev(lohi)
  }
  vec[ !is.na(vec) & (vec < lohi[1]) ] <- lohi[1]
  vec[ !is.na(vec) & (vec > lohi[2]) ] <- lohi[2]
  return(vec)
}
```

Naming objects

Give objects informative names:

- Names should be:
  1. most importantly, descriptive
  2. as concise as possible while still being descriptive
- Avoid `tmp1`, `tmp2`, ...
  - ...as `tmp`ting as it may be.
- Functions as verbs, objects as nouns

Naming objects

Have **consistency** in your naming systems:

- E.g., `markers` vs. `mnames`; `nft` vs. `n_ft`

Commit to a case:

- `camelCase` vs. `pothole_case`

Don't confuse yourself:

- `total` vs. `totals`
- `result` vs. `rslt`
- `X` vs. `x`
- `z` vs. `zz`

Underscores should be preferred as separator when coding in `R`.

Self-documenting code

Many of these recommendations are to **make code self-documenting**.

Comments should be used mostly for **why**, not **what**.

- The **what** should be inherent from the code itself.

This is really **hard to do**.

- Sometimes its harder than its worth, but it is worth trying!

Self-documenting code

Here's an example of bad code:

```
tmp1 <- 9.81
tmp2 <- 5
tmp3 <- 0.5 * tmp1 * tmp2^2
```

Here's an example of documented bad code:

```
# gravitational constant
tmp1 <- 9.81
# time object is falling
tmp2 <- 5
# displacement of the object
tmp3 <- 0.5 * tmp1 * tmp2^2
```

Self-documenting code

Here's self-documenting code:

```
gravitational_force <- 9.81
time_in_seconds <- 5
displacement <- 1/2 * gravitational_force * time_in_seconds^2
```

Now let's add a comment explaining the why:

```
# compute displacement of falling object with Newton's equation
gravitational_force <- 9.81
time_in_seconds <- 5
displacement <- 1/2 * gravitational_force * time_in_seconds^2
```

Self-documenting code

Even better, make it a function!

```
# for falling objects based on Newton's equation
compute_displacement <- function(time_in_seconds){
  gravitational_force <- 9.81
  time_in_seconds <- 5
  displacement <- 1/2 * gravitational_force * time_in_seconds^2
  return(displacement)
}
```

Rules for writing comments:

1. *Only comment code that's hard to understand.*
2. *Try not to write code that's hard to understand.*

Recap

Why spend a lecture on coding style if everyone knows how to code?

- Clear code is more likely to be **correct**.
- Clear code is **easier to use**.
- Clear code is **easier to revisit** six months from now.
- Software based on clear code is **easier to maintain**.
- Clear code is **easier to extend**.