13: Summary + Extras

bit.ly/SISBID3

We'll briefly summarize all that we discussed, and further touch on how to share your research data and code. Then we'll give pointers to the many things that we didn't have time to talk about.

The most important tool is the mindset, when starting, that the end product will be reproducible.

– Keith Baggerly

So true. Desire for reproducibility is step one.

Steps toward reproducible research

- Slow down
- Organize; document
- Everything with code
- Scripts → RMarkdown
- Code → functions → packages
- Version control with Git
- Automation with Make
- Choose a license
- Share your work with others

Moving from "standard practice" to "fully reproducible" is hard. There are a lot of tools to learn and a lot of workflow changes to make. Don't try to change everything all at once. Focus on improving one aspect at a time, ideally jointly with your friends and colleagues. Your goal should be to have each project be a bit better organized than the previous.

Make with R Markdown

To use Make with R Markdown, you need to tell your operating system where it can find pandoc. RStudio includes pandoc, but you need to add the relevant directory to your `PATH`.

Mac:

```
/Applications/RStudio.app/Contents/MacOS/pandoc
```

Windows:

```
"c:\Program Files\RStudio\bin\pandoc"
```


~/.bash_profile

```
export PATH=$PATH:/Applications/RStudio.app/Contents/MacOS/pandoc

noclobber=1    # prevent overwriting of files
IGNOREEOF=1    # disable Ctrl-D as a way to exit
HISTCONTROL=ignoredups

alias cl='clear;cd'
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
alias ls='ls -GF'
alias 'l.'='ls -d .[a-zA-Z]*'
alias ll='ls -lh'
alias md='mkdir'
alias rd='rmdir'
alias rmb='rm .*~ *~ *.bak *.bk!'

alias Rb='R CMD build --force --resave-data'
alias Ri='R CMD INSTALL --library=/Users/kbroman/Rlibs'
alias Rc='R CMD check --library=/Users/kbroman/Rlibs'
alias Rcc='R CMD check --as-cran --library=/Users/kbroman/Rlibs'
```

Use the `.bash_profile` file to define various variables and aliases to make your life easier.

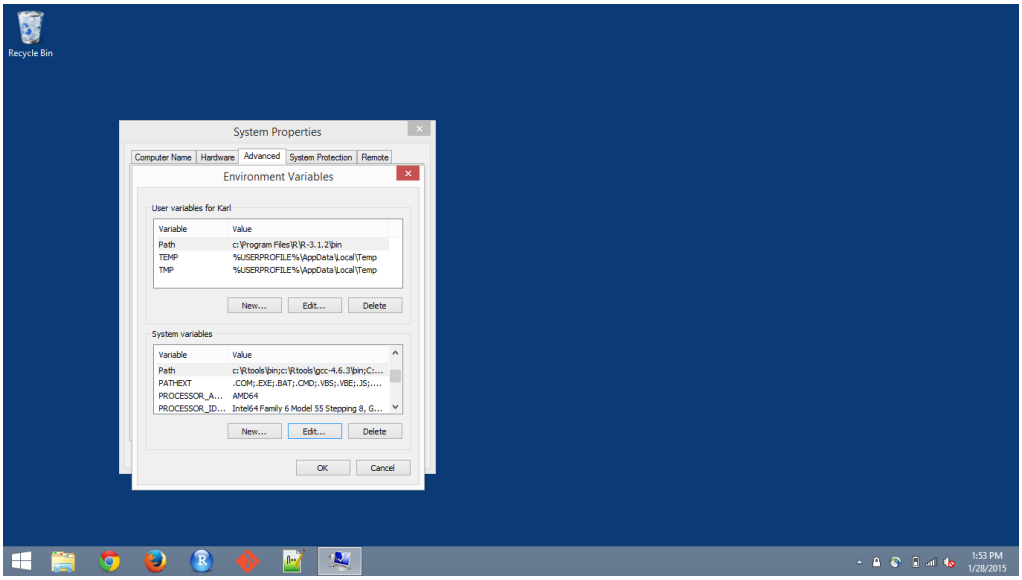
The most important variable is `PATH`: it defines the set of directories where the shell will look for executable programs. If "`.`" isn't part of your `PATH`, you'll need to type something like `./myscript.py` to execute a script in your working directory. So put "`.`" in your `PATH`.

My `.bash_profile` file sources a `.bashrc` file; I don't quite understand when one is used versus the other. Google "`.bashrc vs .bash_profile`." There are links to my `.bash_profile` and `.bashrc` files on the resources page at the course web site; some of it might just be total crap.

If you're using Windows and Git Bash, the `.bash_profile` file will be in your Documents folder (I think).

Important note: use of aliases within your code will create reproducibility issues; another user will need those same aliases. Consider testing your code on a more basic account.

`PATH` in Windows



With Git Bash, you can have a `~/.bash_profile` file that adds stuff to your `PATH`, just as in Mac OS X and Linux.

But things will also be added to the `PATH` variable via the Path system variable and/or a Path user variable. You can get to these via the "Control panel," but it's a bit cumbersome.

The simplest way to get to the relevant dialog box seems to be to click Win-w (the little windows key and the w key) and searching for "path".

Challenges

- Daily maintenance
  - README*s* up to date?
  - Documentation matches code?
- Cleaning up the junk
  - Move defunct stuff into an `Old/` subdirectory?
- Start over from the beginning, nicely?

The organization of a project is dependent on your worst day with it. You keep everything carefully arranged for months and then one day you need to rush, rush, rush to get a manuscript out the door, and you leave a big mess.

And a common problem is that you don't really know what you're doing until it's all done. So if you can, spend a week at the end making a separate, clean version of the whole thing.

Sharing your work

- Why share?
  - Funding agency or journal requirement
  - Increased visibility
  - So that others can build on your work
- When?
  - Continuously and instantaneously
  - When you submit a paper
  - When your paper appears
- Risks?

bit.ly/rr_sharing_slides

There a lot of advantages to making your work public, and you may be required to make it public.

Some scientists put all of the work in the open as they're doing it. Others wait until they submit a manuscript; others until the manuscript actually appears. Still others want to delay releasing data yet further. The earlier the better, I think.

Are there risks? Many worry about being scooped: having someone find something cool in your data before you got a chance to find it yourself. I think the risk of this is far outweighed by the advantages of having data and code in the open. You may also worry about people finding problems in your analyses. But if there are problems, wouldn't you rather know about them? You can't build upon it if it's not right.

The biggest risk is that you'll be ignored. Sharing more, sooner, and in a more convenient form will encourage broader visibility.

I'm not worried about being scooped,
I'm worried about being ignored.

– Magnus Nordborg

I'm not entirely sure that this is the original source of the idea.

- Share more
- Share sooner
- Share in a way that it's easy for others to learn from and build upon

It's easy for me to say, but I think the benefits, to the community, of open science far outweigh the risks to individual investigators. Sort of like vaccines.

What to share?

- For sure
  - Primary dataset
  - Metadata
  - Data cleaning scripts
  - Analysis scripts
- It could help
  - Very-raw data
  - Processed/clean data
  - Intermediate results
- No
  - Confidential data (e.g. HIPAA data)
  - Passwords, private keys

It can be tricky to define what is the "primary dataset". How raw of data should you share? Where would someone want to start, and how can you make it easiest for them to dig into your analyses or make use of the data for other purposes, such as for a meta-analysis study? In some cases, it can be valuable to share many of the intermediate results, so that folks can jump into the bit that they care about without having to first run a bunch of complex and time-consuming analyses to get there.

You definitely want to make sure that you're not including any confidential information, particularly regarding patient-level data, but also private keys for data APIs which you might be including in your scripts.

Where to share?

- Domain-specific repository
  - Genbank, dbGaP, etc.
  - See `re3data.org`
- Figshare, Dryad, Zenodo
- Institutional repository
- GitHub, BitBucket

Share code at GitHub or BitBucket. But how about data? Consider domain-specific repositories like dbGaP, then general data repositories like Zenodo, and then look to see whether your institution has a data repository.

Another option is as supplemental material for a publication, at the Journal's website, but this is often the most cumbersome for users.

Resources

- R Markdown
  - `rmarkdown.rstudio.com`
- R Packages
  - Releasing to CRAN: `r-pkgs.had.co.nz/release.html`
  - Leek group: `github.com/jtleek/rpackages`
  - When to trust an R package: `bit.ly/trust_r_pkg`
- Make
  - `kbroman.org/minimal_make`
  - remake R package, `github.com/richfitz/remake`
- Git
  - Git branches: `nicercode.github.io/git/branches.html`
  - Hadley on Git/GitHub: `r-pkgs.had.co.nz/git.html`
  - Git subtrees: `bit.ly/git_subtree`
- Also see `bit.ly/sisbid3_resources`

Links to some resources related to the course material.

R CMD BATCH

Why is it cool?

- Scripting lets you run everything in the background
- Increases the odds you've got *everything* reproducible

Why didn't we cover it?

- A bit geek-heavy

Where would we point you?

- Stackoverflow discussion: `bit.ly/so_rscript`
- Codecademy Learn the Command Line lesson: `bit.ly/learn_command_line`

Command-line-based programming, such as with `R CMD BATCH`, is really useful for making a project fully reproducible.

Command-line-based programming gives you more complete control of your workflows.

Coding conventions

Why are they cool?

- They help you keep things consistent between team members
- They make code easier to read, and more likely to be used

Why didn't we cover them?

- Not enough time

Where would we point you?

- Hadley's recommendations `adv-r.had.co.nz/Style.html`
- Google's recommendations `google.github.io/styleguide/Rguide.xml`

Coding conventions help to keep things consistent and make code easier to read.

Code review

Why is it cool?

- Helps to find bugs and clean up confusing bits
- Potentially a test of the reproducibility of your work

Why didn't we cover it?

- Not enough time

Where would we point you?

- Software Carpentry blog post, `bit.ly/swc_codereview`
- Titus Brown's blog post, `http://bit.ly/titus_codereview`

Working together with another person and reviewing each other's code can lead to better code and better projects.

Software testing

Why is it cool?

- Explicit tests help you to avoid bugs, and to find bugs sooner

Why didn't we cover it?

- Not enough time

Where would we point you?

- testthat package, `github.com/hadley/testthat`
- Testing R Code book

Hadley wrote, “It’s not that we don’t test our code, it’s that we don’t store our tests so they can be re-run automatically.”

Including explicit tests and running them repeatedly will help you to avoid bugs or at least to find bugs sooner.

Continuous integration (eg Travis)

Why is it cool?

- Automatically build and run tests when you push to GitHub

Why didn’t we cover it?

- Not enough time

Where would we point you?

- Julia Silge blog post, `juliasilge.com/blog/beginners-guide-to-travis`
- Hadley’s R packages book, `r-pkgs.had.co.nz/check.html#travis`

Travis makes it easy to run packages tests regularly.

Capturing dependencies

Why is it cool?

- Ensure that your carefully constructed reproducible project doesn't fail due to a change in one of the packages you use

Why didn't we cover it?

- Not enough time

Where would we point you?

- packrat package, `github.com/rstudio/packrat`
- checkpoint package, `github.com/RevolutionAnalytics/checkpoint`

Saving copies of the exact versions of packages that you use can help ensure that your reproducible project will continue to be reproducible.

Containers (e.g. docker)

Why are they cool?

- Capture your entire environment, so your project is *for sure* fully reproducible.

Why didn't we cover them?

- A bit technical

Where would we point you?

- Rocker: Docker for R
- R Docker tutorial

Docker containers allow you to encapsulate your entire environment (including the operating system and all libraries) for full reproducibility.

R Markdown templates

Why are they cool?

- More complete control over the appearance of your document

Why didn't we cover them?

- A bit technical

Where would we point you?

- R Markdown documentation

Document templates give you full control over the appearance of your R Markdown document output.

knitr Bootstrap

Why is it cool?

- Allows for generation of slicker reports

Why didn't we cover it?

- A bit technical

Where would we point you?

- `github.com/jimhester/knitrBootstrap`

bootstrap-styled reports

GitHub pages

Why are they cool?

- Webpages built entirely in Markdown, providing nicer interfaces to your content

Why didn't we cover them?

- Tangential to *reproducible research*?

Where would we point you?

- `pages.github.com`
- `kbroman.org/simple_site`
- `bookdown.org/yihuit/blogdown`

Slick web pages from a github repository

Bookdown

Why is it cool?

- Write a book (or book-like object) entirely in R Markdown

Why didn't we cover it?

- Not enough time

Where would we point you?

- `bookdown.org/yihui/bookdown`

Surprisingly easy way to create an e-book or website.

Shiny!

Why is it cool?

- Interactive pictures have pizzazz.

Why didn't we cover it?

- Tangential to *reproducible research*?

Where would we point you?

- `shiny.rstudio.com`
- `shiny.rstudio.com/tutorial`

Feedback we'd like from you (1)

What motivated us to teach this course?
What would we see as a positive outcome?

- Given this motivation, are we doing things right?
- What motivated you to take this course?
- Were there specific sessions you found really useful/really useless?
- Points you'd like us to expand on?
- Were there points you were hoping we'd cover that we didn't?

Feedback we'd like from you (2)

- Do you have examples/anecdotes you think we might be able to use that you'd be willing to share?
- Were there ways we could've used time more effectively?
- Were there ways we could've used our TAs more effectively?
- Can you see things you learned in this course changing how you do things day to day?
  - Why or why not?
  - Can we ask you again in 6 months?
  - Can we ask you again in a year?
- Could you write this down now? (anonymous is fine)