# Responsible scientific computing

**github.com/mbjoseph/responsible-computing**

Max Joseph

30 August, 2016

## Science is the pursuit of pure truth, and the systematizing of it

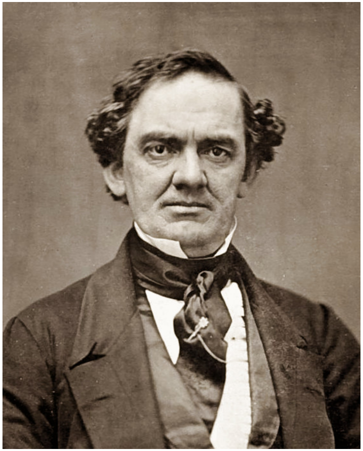
**Figure 1:** P.T. Barnum

## If science is the pursuit of truth...

then we should probably try to avoid being wrong

## But mistakes are nearly inevitible

Modern science demands competence in:

- programming
- data management
- data processing
- workflow management

**But, academic training is still catching up**

## A realistic goal:

Try hard to avoid mistakes. Try harder to identify your mistakes and ensure that when you find one:

- you know exactly what it was, and
- you know exactly when and how it is fixed

## Best practices for scientific computing

Wilson G, et al. (2014) Best Practices for Scientific Computing. PLoS Biol 12(1)

## Today: best and worst practices (BP & WP) in scientific computing

**TLDR**: best practices exist for modern science, but they require a bit of effort. It is our responsibility as scientists to adopt best practices to minimize errors and maintain reproducibility.

## BP1: Let the computer do the work

- analyze your data programmatically
- automate repetitive tasks (at least)

## Automating repetitive tasks

Example tasks:

- data QA/QC (write a function that checks the data as it is collected or entered)
- data processing (e.g., computing summary statistics for many data files)
- producing figures and tables for publication
- updating numeric values in a manuscript every time the analysis or data changes

## BP1.1: let the computer manage your workflow

**GNU Make as a workflow manager**

`https://github.com/weecology/data-sharing-paper`

`https://bsmith89.github.io/make-bml/`

`http://kbroman.org/minimal_make/`

`https://www.gnu.org/software/make/manual/make.pdf`

## WP1: do the work yourself

- manually saving figures
- processing, visualizing, or analysing data in Excel
- manually running scripts from the terminal
- manually executing lines of code in scripts in some specific order
- editing figures in Adobe Illustrator, Inkscape, etc.

# Manual edit revision horror

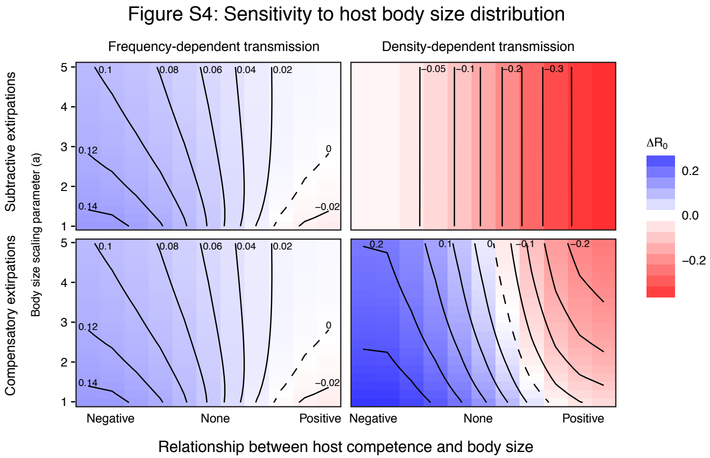

**Figure 2:** A plot that I thought warranted a once-over in Illustrator

## Making the computer do the work takes work

Setting up workflows and programmatic data processing/analysis is time consuming, but:

- it's a good bet that you **will** repeat the analysis (at least a few times, possibly many times)
- your work is easier to reproduce if the sequence of opertations is programmed
- your workflow becomes self-documenting (you don't have to spend as much time documenting how to run your code)

## BP2: put some effort into programming well

- write reusable functions that can be tested
- include condition checks in your functions to verify proper input
- follow a code style guide
- use good names
- write DRY code

## BP2.1: Use good names

```
There are only two hard things in Computer Science:
cache invalidation and naming things.
- Phil Karlton
```

- avoid one or two character names when a more descriptive name can be used (e.g., `yearly_precip` vs. `y`)
- use verbs for function names `get_location()` vs. `location()`
- don't overwrite special names

```
t <- 1:10
T <- FALSE
mean <- mean(x)
matrix <- matrix(1:4, nrow = 2)
```

## WP2: don't bother investing in writing good code, you're a scientist not a software developer!

- write extremely long scripts with copy and pasted code (copypasta)
- don't worry about condition checking, you're the only one who will ever see/use this code
- combine camelCase with pothole_case for extra_hardTo_read_code
- use short names like `x`, `y`, and `t` because they're faster to type

## WP2 show and tell: code behind my first big paper in a high IF journal

- Copypasta
- Rotten documentation, long function, unclear hard-coded indexing, conditional statement glut
- The “Master” script

## BP2.1: don't be overly clever

You're a scientist, not a software developer!

- keep it simple
- don't over-engineer your code
- don't write code that is not used because you *might* use it some day
- don't re-factor as you write code
- get something that works first
- spend time trying to find existing solutions before engineering your own

## BP3: optimize code only after it works correctly, and only if you need performance gains

Not all code needs to be fast. If you need speed:

- use a profiler to identify which bits of code should be optimized (e.g., profvis for R, cProfile for Python)
- think about which parts of your code might be parallelized
- do some math to find a more efficient implementation; ten minutes of math can save you ten hours of computation.

*profvis demo

## WP3: let your computer do the work for weeks at a time

- computers don't mind working without overtime, let it run
- if you need to revise your paper and update the analysis, reviewers/editors/collaborators can wait

## BP4: plan for spontaneous workstation combustion

Computers and hard drives die all the time. It is your responsibility to make sure that this has no negative impacts on your science. Back it all up:

- e.g., code & manuscript on GitHub, GitLab, Bitbucket
- data on PetaLibrary, external hard drives, Dropbox, Figshare, Dryad

## WP4: keep it all on your laptop

"It all goes away. Eventually, everything goes away."
Elizabeth Gilbert, Eat, Pray, Love

## But seriously your laptop will fail soon

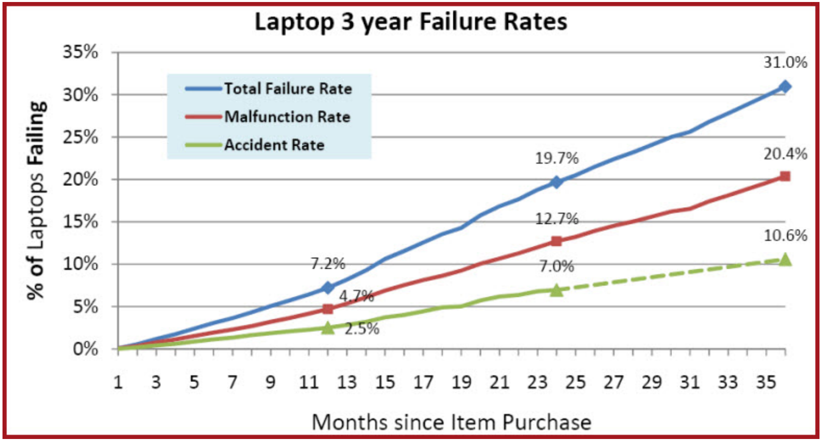

**Figure 4:** `https://www.squaretrade.com/htm/pdf/SquareTrade_laptop_reliability_1109.pdf`

## BP5: make incremental changes to code

Why?

1. less likely to break things
2. if your program breaks, you know why

## BP5: make incremental changes to code

How?

**Test driven development**

Write tests before code, then create code to pass the test(s)

**Ad hoc academic development**

Make small changes, run your program, and evaluate manually whether your expectations match output

## BP5.1: Test suites: required in production environments, rare in academia

A set of unit tests designed to evaluate whether your program's components (unit tests), and program as a whole
e.g., `testthat` in R, `unittest` in Python

## WP5: make all the changes at once

You have a lot of coding to do, so do it all at once and then clean up the mess.

**Why this is bad**

- if you're lucky, any mistakes will raise an error (test suites and test conditions make this more likely)
- if you're unlucky, you'll introduce mistakes that go unnoticed
- if you're like me, you'll have to spend an entire day trying to identify and fix any bugs that you've created

## BP6: Use version control

Use git to keep track of different versions of your code/project
Example repo

## BP6: but I've got no time to learn git

- reverting to previous states is easy
- collaborating with your (past/future) self, and others is easier
- working on multiple machines is easier
- your files are backed up

## WP6: Store a bunch of copies of your files with clever names

```
cleaned_data August 13.xlsx
cleaned_data August 13.5.xlsx
cleaner_data August 14.xlsx
cleanest_data August 15.xlsx
cleanest_data August 15_final.xlsx
cleanest_data August 15_final_NOMISSINGDATA.xlsx
cleanest_data August 15_final_NOMISSINGDATA_EXTRAFINAL.xlsx
```

## BP7: make your work reproducible

... a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures. - Jon Claerbout

## BP7: make your work reproducible

Make your code and raw data available via:

- code supplements
- public repositories (e.g., Figshare, Dryad)
- git repositories (e.g., GitHub, BitBucket, GitLab)

## BP7.1: releasing a software environment

… a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is **the complete software development environment and the complete set of instructions** which generated the figures. - Jon Claerbout

## BP7.1 Docker for reproducible research

Docker:

- a platform to build, ship and run software
- similar to a virtual machine, but very lightweight

## Docker example

Suppose you have some code:

`https://github.com/earthlab/noaa-demo`

This code has a semi-heinous set of dependencies:

- *NIX operating system
- GNU Make
- Amazon web services command line interface
- HDF5 library
- R
- A smattering of packages: raster, rgdal, dplyr, ggplot2, plot3D, gridExtra, rasterVis, colorspace, rhdf5, data.table, akima

## Options for reproducing the analysis

1. Running locally on your OS

```
git clone https://github.com/earthlab/noaa-demo
<<install dependencies for hours>>
make
```

## Options for reproducing the analysis

1. Running locally on your OS
2. Running locally in a Docker container controlled by your OS

```
docker run mbjoseph/noaa-demo
```

## What happens when you execute docker run

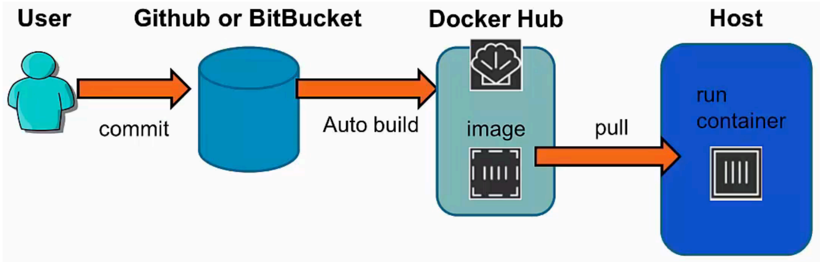

**Figure 5:** `http://slidedeck.io/lodelestra-edu/docker-slides`

## Docker Hub

A repository of Docker images:

`https://hub.docker.com/u/mbjoseph/`

With Docker:

- your code packs its own lunch (the execution environment)
- anyone with Docker can run it, and the *only dependency is Docker*

## How to Docker?

Write a Dockerfile:

- similar to a Makefile
- embeds all the Linux commands necessary to set up your environment and run your code
- you can import other Docker images as starting points (e.g., ones preloaded with GDAL, R, Python, the AMES Stereo Pipeline, etc.)

`https://github.com/mbjoseph/dockerfiles`

## More on Docker in a scientific context

Carl Boettiger. 2015. An introduction to Docker for reproducible research. SIGOPS Oper. Syst. Rev. 49, 1 (January 2015), 71-79. DOI: `http://dx.doi.org/10.1145/2723872.2723882`

## BP7.2: use free software when possible

Open source and/or free software makes your work more reproducible
If it cost you $50 to repeat someone's analysis, would you do it?
How about:

- $1,500 (ArcGIS Basic)
- $2,150 (Matlab)
- $7,000 (ArcGIS Standard)

## BP7.3: use free software for the paper itself

MS Word is not free, and it breaks a lot!

## Some free options for writing

1. $\LaTeX$
   - for beginners, try Overleaf or ShareLatex

*Overleaf demo

## Some free options for writing

1. LaTeX
2. Scholarly markdown

*Markdown demo

## Some free options for writing

1. LaTeX
2. Scholarly markdown
3. R Markdown (hybrid approach: the paper is written in Markdown but rendered to pdf via LaTeX)

## BP8: get feedback on your code

Reviewers increasingly have the skills to evaluate your code along with your paper!

**Benefits**

- double check your work
- find errors before you're at risk of retraction
- get reviewers to do your work for you (via pull request)

**Risks**

- exposing your code to the world
- getting "scooped" if they steal your code (but, you should attach a license to protect yourself)

## BP8: get feedback on your code

**“Safe” routes**

- ask for a “friendly code review”
- ask your labmate to reproduce your analysis
- collaborate with your co-authors on code, not just the manuscript (for many projects, the data, code, and analysis is the foundation upon which the paper is written)

## BP review

1. Let the computer do the work
2. Put some effort into programming well
3. Optimize only once the code works
4. Plan for spontaneous workstation combustion
5. Make incremental changes
6. Use version control
7. Make your work reproducible
8. Get feedback on your code

## Post script: worst practices according to Twitter

- Version control via file name `@mike_skaug`
- Build gui, hardware control and data analysis software in Matlab because with a hammer everything looks like a nail `@mike_skaug`
- Under-commented code. Excessively long lines of code. Data stored on a single drive. Using MS Office as a standard. `@R_you_cereal`
- copy-paste rather than writing functions. Not using version control, hence ending up with script1, script1revised, etc `@frod_san`
- no backups `@naupakaz`
- meaningful colored cells in excel. #shudder `@elindie`

# Thank you

## Further reading

- Wilson G, et al. (2014) Best Practices for Scientific Computing. PLoS Biol 12(1).
- Carl Boettiger. 2015. An introduction to Docker for reproducible research. SIGOPS Oper. Syst. Rev. 49, 1 (January 2015), 71-79.
- Littauer, Richard, et al. "Trends in use of scientific workflows: insights from a public repository and recommendations for best practice." International Journal of Digital Curation 7.2 (2012): 92-100.
- Ram, Karthik. "Git can facilitate greater reproducibility and increased transparency in science." Source Code for Biology and Medicine 8 (2013): 7.