Good practices: debugging, coding practices and reproducible research

September 4, 2018

Sources:

- Chambers
- Hadley Wickham's advanced R notes on debugging.
- Roger Peng's notes on debugging in R
- Murrell, Introduction to Data Technologies, Ch. 2
- Journal of Statistical Software vol. 42: 19 Ways of Looking at Statistical Software
- Wilson et at., Best practices for scientific computing, ArXiv:1210:0530
- Gentzkow and Shapiro tutorial for social scientists
- https://github.com/berkeley-stat243/stat243-fall-2014/blob/master/section/millman-perez.pdf

This unit covers debugging (and practices for avoiding bugs), good coding/software development practices, and doing reproducible research. As in later units of the course, the material is generally not in any fundamental way specific to R, but some details and the examples are in R.

1 Debugging and recommendations for avoiding bugs

Please see the *R debugging* tutorial for information on

- basic debugging strategies,
- using R's interactive debugging tools,

- common causes of bugs,
- tips and tools for avoiding bugs and catching errors, and
- information on how to get help online.

2 Good coding practices

Some of these tips apply more to software development and some more to analyses done for specific projects; hopefully it will be clear in most cases.

2.1 Editors

Use an editor that supports the language you are using (e.g., *Sublime, Atom, Emacs/Aquamacs*, *vim, TextMate, WinEdt*, *Tinn-R*, or the built-in editors in *RStudio*). Some advantages of this can include: (1) helpful color coding of different types of syntax and of strings, (2) automatic indentation and spacing, (3) code can often be run or compiled from within the editor, (4) parenthesis matching, (5) line numbering (good for finding bugs).

2.2 Coding syntax

The files *goodCode.R* and *badCode.R* in the *units* directory of the class repository provide examples of code written such that it does and does not conform to the suggestions listed in this section.

Here are some style guides:

- Adler has style tips.
- Hadley Wickham's style guide.
- A empirical style guide based on the code of R Core and key package developers.
- Google's R style guide.
- This R journal article summarizes the state of naming styles on CRAN.

And here's a summary of my own thoughts:

- Header information: put metainfo on the code into the first few lines of the file as comments. Include who, when, what, how the code fits within a larger program (if appropriate), possibly the versions of R and key packages that you wrote this for

- Indentation: do this systematically (your editor can help here). This helps you and others to read and understand the code and can help in detecting errors in your code because it can expose lack of symmetry.
- Whitespace: use a lot of it. Some places where it is good to have it are (1) around operators (assignment and arithmetic), (2) between function arguments and list elements, (3) between matrix/array indices, in particular for missing indices.
- Use blank lines to separate blocks of code and comments to say what the block does
- Split long lines at meaningful places.
- Use parentheses for clarity even if not needed for order of operations. For example, `a/y*x` will work but is not easy to read and you can easily induce a bug if you forget the order of ops.
- Documentation - add lots of comments (but don't belabor the obvious). Remember that in a few months, you may not follow your own code any better than a stranger. Some key things to document: (1) summarizing a block of code, (2) explaining a very complicated piece of code - recall our complicated regular expressions, (3) explaining arbitrary constant values.
- For software development, break code into separate files (<2000-3000 lines per file) with meaningful file names and related functions grouped within a file.
- Choose a consistent naming style for objects and functions: e.g. *nIts* vs. *n.its* vs *numberOfIts* vs. *n_its*
  - This R journal article summarizes the state of naming styles on CRAN.
  - Adler and Google's R style guide recommend naming objects with lowercase words, separated by periods, while naming functions by capitalizing the name of each word that is joined together, with no periods.
  - On the other hand, programmers who use other languages dislike R code with periods in it except in the context of object-oriented programming (OOP). E.g., *summary.lm* is clear in that the period distinguishes the method from the class. Naming a method something like *special.summary.lm* or an object *my.summary* then confuses things. Personally, I suggest avoiding periods except for OOP.
- Try to have the names be informative without being overly long.

- Don't overwrite names of objects/functions that already exist in R. E.g., don't use 'lm'.

```
> exists('lm')
```

- Use active names for functions (e.g., *calcLogLik*, *calc_logLik* rather than *logLik* or *logLikCalc*)

- Learn from others' code

This semester, someone will be reading your code - Omid and me when we look at your assignments. So to help us in understanding your code and develop good habits, put these ideas into practice in your assignments.

2.3 Coding style

This is particularly focused on software development, but some of the ideas are useful for data analysis as well.

- Break down tasks into core units

- Write reusable code for core functionality and keep a single copy of the code (w/ backups of course or ideally version control) so you only need to make changes to a piece of code in one place

- Smaller functions are easier to debug, easier to understand, and can be combined in a modular fashion (like the UNIX utilities)

- Write functions that take data as an argument and not lines of code that operate on specific data objects. Why? Functions allow us to reuse blocks of code easily for later use and for recreating an analysis (reproducible research). It's more transparent than sourcing a file of code because the inputs and outputs are specified formally, so you don't have to read through the code to figure out what it does.

- Functions should:

  - be modular (having a single task);
  - have meaningful name; and
  - have a comment describing their purpose, inputs and outputs (see the help file for an R function for how this is done in that context).

- Write tests for each function (i.e., unit tests)

- Object orientation is a nice way to go
- Don't hard code numbers - use variables (e.g., number of iterations, parameter values in simulations), even if you don't expect to change the value, as this makes the code more readable:

  ```
  > speedOfLight <- 3e8
  ```

- Use R lists to keep disparate parts of related data together
- Practice defensive programming (see also the discussion below on assertions)
  - check function inputs and warn users if the code will do something they might not expect or makes particular choices;
  - check inputs to *if* and the ranges in *for* loops;
  - provide reasonable default arguments;
  - document the range of valid inputs;
  - check that the output produced is valid; and
  - stop execution based on checks and give an informative error message.
- Try to avoid system-dependent code that only runs on a specific version of an OS or specific OS
- Learn from others' code
- Consider rewriting your code once you know all the settings and conditions; often analyses and projects meander as we do our work and the initial plan for the code no longer makes sense and the code is no longer designed specifically for the job being done.

2.4 Assertions and testing

Both tests and assertions are critically important for writing robust code that is less likely to contain bugs.

Assertions are checks in your code that the state of the program is as you expect, including arguments provided by users. In addition to simple use of *stopifnot()* and using *if()* combined with *stop()* and *warning()*, the *assertthat* and *assertr* packages provide useful tools. The *checkmate* package specifically helps in checking arguments.

Tests evaluate whether your code operates correctly. This can include tests that your code provides correct and useful errors when something goes wrong (so that means that a test might be

to see if problematic input correctly produces an error). *Unit tests* are intended to test the behavior of small pieces (units) of code, generally individual functions. Unit tests naturally work well with the ideas above of writing small, modular functions. *testthat* and other packages are designed to make it easier to write sets of good tests.

See the materials in Lab 2 on September 4 for more details about assertions and testing.

2.5 Version control

- Use it! Even for projects that only you are working on.
- Use an issues tracker, or at least a simple to-do file, noting changes you'd like to make in the future.
- In addition to good commit messages, it's a good idea to keep good notes documenting your projects.

We've already seen Git in a lot of detail, so not too much more to say here.

3 Tips for running analyses

Save your output at intermediate steps (including the random seed state) so you can restart if an error occurs or a computer fails. Using *save()* and *save.image()* to write to *.RData* files work well for this.

Run your code on a small subset of the problem before setting off a job that runs for hours or days. Make sure that the code works on the small subset and saves what you need properly at the end.

4 Reproducible research

The idea of "reproducible research" has gained a lot of attention in recent years because of the increasing complexity of research projects, lack of details in the published literature, failures in being able to replicate or reproduce others' work, fraudulent research, and for other reasons.

We've seen a number of tools that can help with doing reproducible research, including version control systems such as git, the use of scripting such as bash and R scripts, and literate programming tools such as knitr and R Markdown.

Provenance is becoming increasingly important in science. It basically means being able to trace the steps of an analysis back to its origins. *Replicability* is a related concept - the idea is that

you or someone else could replicate the analysis that you've done. This can be surprisingly hard as time passes even if you're the one attempting the replication.

Open question: What is required for something to be replicable? What are the challenges in doing so?

4.1 Some basic strategies

- Have a directory for each project with meaningful subdirectories: e.g., *code*, *data*, *paper*
- Keep a document describing your running analysis with dates in a text file (i.e., a lab book)
- Note where data were obtained (and when, which can be helpful when publishing) and pre-processing steps in the lab book. Have data version numbers with a file describing the changes and dates (or in lab book).
- Have a file of code for pre-processing, one or more for analysis, and one for figure/table preparation.
  - The pre-processing may involve time-consuming steps. Save the output of the pre-processing as a file that can be read in to the analysis script.
  - You may want to name your files something like this, so there is an obvious ordering: "1-prep.R", "2-anal.R", "3-figs.R".
  - Have the code file for the figures produce the EXACT manuscript figures, operating on an RData file that contains all the objects necessary to run the figure-producing code; the code producing the RData file should be in your analysis code file (or somewhere else sensible).
  - Alternatively, use *knitr* (or *R Markdown* or *IPython*) for your document preparation.
- Note what code files do what in the lab book.

4.2 More formal tools

1. In some cases you may be able to carry out your complete workflow in a knitr//R Markdown document or in a Jupyter notebook (e.g., IPython notebook, Jupyter-based R notebook).
2. You might consider using the UNIX utility *make*, which is generally used for compiling code, as a tool for reproducible research: if interested, see the tutorial on *Using make for workflows* for more details.