Testing R Code

Goals

The tutorial aims to educate users in three things:
- How to write R code that is easy to test.
- How to perform run-time testing.
- How to perform development-time testing.

Justification

As R gains in popularity, increasing numbers of users come from outside the traditional demographic of programming-literate statisticians. There are now many scientists, social scientists, business analysts, and more trying to write R code. A good fraction of these have little knowledge of what good code looks like, and even less knowledge of how to test that code.

Since statistics is hard, it is very easy to make small mistakes with big consequences. A simple missed minus sign or a parenthesis in the wrong place can create catastrophic, hard to spot, errors that completely change your results. And if you can't be sure that your results are correct, then you might as well not bother writing any code at all.

These two previous points mean that thorough code testing is both spectacularly important but not widely performed throughout the R community.

Background knowledge

Participants should understand common data types (vectors, lists, data.frames) and know how to write a function.

Potential attendees

The tutorial covers fundamental R skills, as and such is suitable for all R users. Package developers are particularly encouraged to attend.

Outline

Section 1: How to write R code that is easy to test.

Motivation: writing buggy code is worse than pointless.
The handshake problem. The number of problems you have is very roughly proportional to the size of the project squared.
Good style can make bugs easier to spot. Quick tour of the formatR package.
"Don't Repeat Yourself" to avoid inconsistency bugs. Refactoring examples.
"Keep It Simple, Stupid": short functions are easier to reason about. What cyclomatic complexity means. Probably some jokes about penny farthings. Quick tour of the sig package. I may name and shame Hmisc if Frank Harrell isn't in the audience.
"You Ain't Gonna Need It": functions with less arguments need less tests. The Pareto principle. Attempts to simplify functions with lots of arguments (strOptions to help with str, read.table vs. data.table::fread, plyr::ddply vs. the dplyr approach).
"Fail Early, Fail Often": where error messages and warnings need to go. A trailer for the assertive package.

Section 2: How to perform run-time testing.
~~~~~~

Examples of awful obscure error messages.
What an assertion is.
stopifnot. An example with a simple algorithm, maybe geometric mean.
assertive vs. assertthat philosophies.
Rewriting the previous obscure error messages in a more readable fashion.
Checklists of common function input checks. is_numeric, is_scalar, is_not_null, etc.
Checking numbers. is_in_range, is_finite, is_real, etc.
Reflection in R. is_windows (.Platform is badly explained), is_slave_r (useful for parallel processing), r_can_compile_code (useful for Rcpp), is_r_devel, etc.
Checking complex data types. is_credit_card_number, is_us_zip_code, etc.
Processing personal data.

Section 3: How to perform development-time testing.
~~~~~~

What an unit test is, and when to use them.
How to create a test. Examples with a simple algorithm, maybe hypotenuse.
Checklists of common tests. (Data types, missing values, missing arguments, boudnary values, big and small numbers, etc.)
Testing objects with names/attributes.
Testing lists and recursive objects.
Testing connections to files/databases.

Advanced content, if there is time:
Testing S3/S4/Ref classes/R6.
Packaging tests.