Ten Simple Rules for Making Research Software More Robust

Morgan Taschuk[1,*], Greg Wilson[2]
1) Ontario Institute for Cancer Research / morgan.taschuk@oicr.on.ca
2) Software Carpentry Foundation / gvwilson@software-carpentry.org
∗ Corresponding author.

Abstract

Software produced for research—even published software—suffers from a number of common problems that make it difficult or impossible to run outside the original institution or even the primary developer's computer. We present ten simple rules to make software robust enough to run anywhere and delight your users and collaborators.

Introduction

As a relatively young field, bioinformatics is full of newly developed software. **MT: examples here** Efforts such as the ELIXIR tools and data registry[1] and the Bioinformatics Links Directory[2] [1] have made efforts towards cataloguing it: as of May 2016, the former has 2500 entries and the latter 1700. Those numbers constantly grow as both trainees and experienced practitioners produce new software to support their research.

Typically, this software is developed by one person and produces excellent results in their hands. But what happens after the student leaves someone else from their group wants to use the program? Everyone with a few years of experience feels a tremor of fear when told to use a graduated student's code to analyze their data. Often, that software will be undocumented and work in unexpected ways (if it works at all). It will often rely on nonexistent paths or resources, be tuned for a single dataset, or simply be an older version than was used in published papers. Its new user is often faced with two unpalatable options: hack the existing code to make it work, or start over.

The root cause of this problem is that most research software isn't *robust*. The difference between running and being robust is the difference between "works for me on my machine" and "works for other people on a cluster I've never used". In particular, robust software:

- works for users other than the original author;
- is kept under version control;
- can be installed on more than one computer or account with relative ease;
- has well-defined input and output formats;

[1] http://dx.doi.org/10.1093/nar/gkv1116
[2] https://bioinformatics.ca/links_directory/

- has documentation that describes what its dependencies are, how to install it, and what the options are; and
- comes with enough tests to show that it actually runs.

These rules need not be applied to *every* coding effort: as the saying goes, not everything worth doing is worth doing right (or right away). Code that is used once to answer a specific question related to a specific dataset doesn't require comprehensive documentation, and the only sensible way to test it may well be to run it on the dataset in question. However, if a script is dusted off and run three or four times for slightly different purposes, crucial to a publication, or being passed on to someone else, it may be time to apply ten simple rules for robust software.

These simple rules are also necessary steps toward creating a reusable library that can be shared through a site like CPAN or CRAN. They are generic and can be applied to all languages, libraries, packages, documentation styles, and operating systems for both closed-source and open-source sofwtare. Whether the aim is as simple as sharing the code with collaborators or as complex as using the software in a production analysis environment, increasing the robustness of your software decreases headaches all around.

1 Have a README that explains in a few lines what the software does and what its dependencies are.

The README is the first stop for most new users. At a minimum, it needs to get a new user started and point them towards more help, if they need it. A working example using test data (per #9) is always appreciated.

Explain what the software does: At the beginning of the README, explain what the software does in one or two sentences. There's nothing more frustrating than spending the time to download and install some software only to find out that it doesn't do what you thought it did.

List required dependencies: Often, software depends on very specific versions of libraries, modules, or operating systems. This is entirely reasonable as long as it is properly documented. Often, multiple libraries exist with the same or very similar names, so either provide the commands necessary to download the dependencies or link to the software homepage. Include the version number for each dependency.

If your dependency is not publicly available, you have several options to ensure your software remains useable. You can add plain-text or code directly to your repository with appropriate attribution. If the dependency is a binary, use a binary repository manager[3] such as Artifactory or ProGet. These managers keep versioned copies of software at constant URLs so they can be downloaded

[3] `https://en.wikipedia.org/wiki/Binary_repository_manager`

as long as the manager continues to run. If the dependency must remain closed, place it at an internal location on shared disk, remove all write permissions, and link to it from your README, although this method is discouraged because of the potential for sharing and risks accidental removal.

Installation instructions: If the software needs to be compiled or installed, list those instructions in the README. Also mention if you recommend they use a pre-compiled binary instead through a system such as pip, yum or apt.

Input and output files: All possible input and output files should be listed in this section. Do the files conform to a particular standard, an extension of an existing format, or is it your own format? If using a standard format, link to the specification and version. If you extend the standard or have your own format, define it explicitly, listing all the required fields and acceptable values. (You get bonus points if you include a script to convert between a standard format and your file format). If there is no rigorous format (such as with log files), show an example file, or the first few lines, and explain what the sections mean.

Input files and their formats are included in most documentation, but intermediate, auxilliary and log files are often missing. *All* files should be listed in the README, even those considered self-explanatory. Log files are often full of valuable information that can be mined for the user's specific purpose. If your users might need to know, "Does this program report the percentage of reads trimmed to remove adapter sequences?" they should be able to check the README and confidently say, "Yes, it is in the log file".

Attributions and licensing: Attributions are how you credit your main contributors; licenses are how you want others to use and credit your software. By putting them in the README, it is readable *before* the software is installed (or even downloaded). Leave no question in anyone's mind about whether your software can be used commercially, how much modification is permitted, and how other software needs to credit you. If your software is not open source, state that clearly. Attributions can also contain a list of "expert" users that can be contacted if new users have problems with the software.

The README file for khmer[4] is a good model: it explains the software's purpose, tells readers where to get help, and includes links to a CITATION file (explaining how to cite the project) and the license.

2 Print usage information when launching from the command line that explains the software's features.

Usage information provides the first line of help for both first-time and experienced users of command-line applications. Ideally, usage is a terse, informative command-line help message that guides the user in the correct use of your software. Terseness is important: usage that extends for multiple screens is a

[4] `https://github.com/dib-lab/khmer/blob/master/README.rst`

nuisance, especially when printed to standard error instead of standard output (where it can easily be paged).

Usage should provide all of the information necessary to run the software. It is invoked either by running the software without any arguments; running the software with incorrect arguments; or by explicitly choosing a help or usage option.

An example of good usage is the Unix `mkdir` command, which makes new directories:

```
$ mkdir --help
Usage: mkdir [OPTION]... DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.
  -m, --mode=MODE   set file mode (as in chmod), not a=rwx - umask
  -p, --parents     no error if existing, make parent directories as needed
  -v, --verbose     print a message for each created directory
  -Z, --context=CTX  set the SELinux security context of each created
                       directory to CTX
      --help     display this help and exit
      --version  output version information and exit

Report mkdir bugs to bug-coreutils@gnu.org
GNU coreutils home page: <http://www.gnu.org/software/coreutils/>
General help using GNU software: <http://www.gnu.org/gethelp/>
For complete documentation, run: info coreutils 'mkdir invocation'
```

There is no standard format for usage statements, but good ones share several features:

The syntax for running the program: This includes the name of the program and defines the relative location of optional and required flags, arguments and values for execution. Arguments in [square brackets] are usually optional. An ellipsis (...e.g. "[OPTION]...") indicates that more than one value can be provided.

Description: Similar to the README, the description reminds users of the software's primary function.

Most commonly used arguments, a description of each, and the default values: Not all arguments need to appear in the usage, but the most commonly used ones should be listed here. Users will rely on this for quick reference when working with your software.

Where to find more information: Whether an email address, web site or manual page, there should be an indication where the user can go to find more information about the software.

Printed to standard output : So that it can be piped into `less`, searched with `grep`, or compared to the previous version with `diff`.

Exit with an appropriate exit code: When usage is invoked by providing incorrect arguments, the program should exit with a non-zero code to indicate

an error. However, when help is explicitly requested, the software should not exit with an error.

3 Give the software a meaningful version number.

Most software has a version number composed of a decimal number that increments as new versions are released. There are many different ways to construct and interpret the version, but most importantly for us, a particular software version run with the same parameters should give identical results no matter when it's run. Results include both correct output as well as any errors. Every time you release your software, i.e. distribute it to someone other than yourself and/or the development team, you should increment your version number.

Semantic versioning[5] is one of the most common types of versioning for open-source software. Version numbers take the form of *MAJOR.MINOR[.PATCH]*, e.g. 0.2.6. The major and minor numbers versions are almost always provided. Changes in the major version number herald significant changes in the software that are not backwards compatible, such as changing or removing features or altering the primary functions of the software. Increasing the minor version represents incremental improvements in the software, like adding new features. Following the minor version number can be an arbitrary number of project-specific identifiers, including patches, builds and qualifiers. Common qualifiers include `alpha`, `beta`, and `SNAPSHOT`, for applications that are not yet stable or released, and `-RC` for release candidates prior to official release.

The version of your software should be easily available, both when supplying `--version` or `-v` on the command line as well as in the results. The software version should be printed to the same location as all of the other parameters (see Rule 4).

Old versions of your software should be available to ensure that results are reproducible into the far future. A number of mechanisms exist for controlled release that range from as simple as adding an appropriate commit message or tag to version control, to official releases alongside code on Sourceforge, Bitbutcket or Github, to depositing into a repository like apt, yum, homebrew, CPAN, etc. Choose the method that best suits the number and expertise of users you anticipate.

4 Make older versions available.

Software evolves, but not all software evolves at the same pace. While more consistent command-line arguments or more flexible handling of output directories may make a program better in general, it can simultaneously make work for someone who integrated the old version into their own workflow a year or

[5] `http://semver.org/`

two ago, and won't see any benefits from upgrading. Programs' authors should therefore ensure that every version they have released continues to be available.

MT: Can you please flesh this out?

5 Do not require root or other special privileges.

Root (also known as "superuser" or "admin") is a special account on a computer that has (among other things) the power to modify or delete system files and user accounts. Conversely, files and directories owned by root usually cannot be modifed by normal users.

Installing or running a program with root privileges is often convenient, since doing so automatically bypasses all those pesky safety checks that might otherwise get in the user's way. However, those checks are there for a reason: scientific software packages may not intentionally be malware, but one small bug or over-eager file-matching expression can certainly make them behave as if they were. Outside of very unusual circumstances, packages should therefore not require root privileges to set up or use.

Another reason for this rule is that users may want to try out a new package before installing it system-wide on a cluster. Requiring root privileges will frustrate such efforts, and thereby reduce uptake of the package. Requiring that software be installed under its own user account (e.g., that `packagename` be made a user, and all of the package's software be installed in that "user's" space) is similarly limiting, and makes side-by-side installation of multiple versions of the package more difficult.

Developers should therefore allow packages to be installed in an arbitrary location, e.g., under a user's home directory in `~/packagename`, or in directories with standard names like `bin`, `lib`, and `man` under a chosen directory. If the first option is chosen, the user may need to modify her search path to include the package's executables and libraries, but this can (more or less) be automated, and is much less risky than setting things up as root.

6 Reuse software (within reason).

In the spirit of code reuse and interoperability, developers often want to reuse software written by others. With a few lines, a call is made out to the other program, the results are incorporated into the primary script. Using popular projects reduces the amount of code that needs to be maintained and adds the strength of vetted software to the final program.

Unfortunately, the interface between two software packages can be a source of considerable frustration. Support requests descend into debugging errors produced by the other project.

In addition, every package someone has to install before being able to use yours is a possible (some would say "likely") source of frustration for some

potential user. On the other hand, research software developers should re-use existing software wherever possible.

To strike a balance between these two, developers should **document *all* of the packages that theirs depends on, preferably in a machine-readable form**. For example, it is common for Python projects to include a file called `requirements.txt` that lists the names of required libraries, along with version ranges:

```
requests>=2.0
pygithub>=1.26,<=1.27
python-social-auth>=0.2.19,<0.3
```

This file can be read by a package manager, which can check that the required software is available, and install it if it is not. Similar mechanisms exist for Perl, R, and other languages.

Conversely, developers should **avoid depending on scripts and tools which are not available as packages**, even (or especially) if they are installed on the computers the original developer is using. In many cases, a program's author may not realize that some tool was built locally, and doesn't exist elsewhere. At present, the only sure way to discover such unknown dependencies is to install on a system administered by someone else and see what breaks. As use of lightweight virtualization containers like Docker becomes more widespread, it may become common to test installation on a virtual machine.

The way that the second program is invoked can throw up errors. The program may not be in the user's path, or it may be an older or newer version and produce results different than expected. Windows users will be frustrated if you invoke `bash` or `sh` explictly or use shell-specific conventions like '*' expansion. Even Linux-standard functions available vary slightly between installs. For example, GNU `sort` is available on almost every *nix distribution, but sorts differently depending on locale.

Despite these caveats, we fully support the reuse of software between projects provided three additional guidelines are adhered to. First, **make sure that you really need the auxiliary program**. If you are executing GNU sort instead of figuring out how to sort lists in Python, it may not be worth the pain of integration.

Second, **ensure the appropriate software and version is available**. Either allow the user to configure the exact path to the package, distribute the program with the dependent software, or download it during installation using a dependency management system. Regardless, check whether the program is executable and what version is running. Be sure to remind users in the documentation what the compatible versions are.

Finally, to ensure support on as many different operating systems as possible, **use native functions for starting other processes**, such as Java's `Runtime.exec` call, Python's subprocess module, and Perl's system command, and be sure to capture and report the standard error output of the subprocess to facilitate debugging.

7 Eliminate hard-coded paths.

It's easy to write software that reads input from a file called `mydata.csv`, but also very limiting. If a colleague asks you to process her data, you must either overwrite your data file (which is risky) or edit your code to read `otherdata.csv` (which is also risky, because there's every likelihood you'll forget to change the filename back, or will change three uses of the filename but not a fourth).

Hard-coding file paths in a program also makes the software harder to run in other environments. If your package is installed on a cluster, for example, the user's data will almost certainly *not* be in the same directory as the software, and the folder `C:\users\yourname\` will probably not even exist.

For these reasons, users should be able to set the names and locations of input and output files as command-line parameters. This rule applies to reference data sets as well as the user's own data: if a user wants to try a new gene identification algorithm using a different set of genes as a training set, she should not have to edit the software to do so.

A corollary to this rule is that a package should not require users to navigate to a particular directory to do their work. "Where I have to be" is just another hard-coded path.

In order to save typing, it is often convenient to allow users to specify an input or output *directory*, and then require that there be files with particular names in that directory. This practice, which is sometimes called "convention over configuration", is used by many software frameworks, such as WordPress and Ruby on Rails, and often strikes a good balance between adaptability and consistency.

8 Allow configuration of all useful parameters from the command line.

Useful parameters are those values that a user will need to modify to suit their computer, dataset or application. The full list of useful parameters is software-specific and so cannot be detailed here, but here is a short list:

- Input and reference files/directories
- Output files/directories
- Filtering
- Tuning (e.g. alphas and gammas)
- Seeds
- Any alternatives that you've built-in, e.g. compress results, use a different calculation, verbose output, etc.

Providing parameters on the command line increases the flexibility and usability of the program. You may have determined early in development that 0.58 is an optimal seed for your original dataset, but that doesn't mean that is the best seed for every case. Being able to change parameters on the fly to determine if and how they change the results is important as your software gains more users, facilitating exploratory analysis and parameter sweeping. Keep in mind that if a parameter can be adjusted, users will want to be able to turn off the feature entirely to have a baseline comparison.

When the software starts, it should echo all parameters and software versions to standard out or a log file alongside the results. This feature supports greater reproducibility because any result can be replicated with only the previous output files as reference.

You can set reasonable default values as long as any command line arguments override those values. Configuration files should be used in preference to hardcoding the defaults directly. Configuration files can be in a standard location, e.g. `.packagerc` in the user's home directory or provided on the command line as an additional argument.

We caution against overusing configuration files. When a user needs to locate, open, change and save a file in order to change a parameter, the import of the change seems larger and discourages experimentation. Only values that are unlikely to change between runs belong in the config file, such as dependencies, servers, version numbers, network drives, and any other defaults for your lab or institution. Since all parameters should be echoed in the results, config files should be cleaned up after execution completes or they occupy valuable disk space. Use a default configuration file and specify all other parameter values on the command line.

9 Include a small test set that can be run to ensure the software is actually working.

Every package should come with a small test script for users to run after installation. Its purpose is *not* to check that the software is working correctly (although that is extremely helpful), but rather to ensure that it will work at all. This test script can also serve as a working example of how to run the software.

In order to be useful, this test script must be easy to find and run. A single file in the project's root directory named `runtests.sh` or something equally obvious is a much better solution than documenting test cases and requiring people to copy and paste them into the shell.

Equally, the test script's output must be easy to interpret. Screens full of correlation coefficients do not qualify: instead, the script's output should be simple to understand for non-experts, such as one line per test, with the test's name and its pass/fail status, followed by a single summary line saying how many tests were run and how many passed or failed. If many or all tests fail

because of missing dependencies, that fact should be displayed once, clearly, rather than once per test, so that users have a clear idea of what they need to fix and how much work it's likely to take.

10 Produce identical results when given identical inputs.

Given a set of parameters and a dataset, the package should produce the same results every time it is run.

Many bioinformatics applications rely on randomized algorithms to improve performance or runtimes. As a consequence, results can change between runs, even when provided with the same data and parameters. By its nature, this randomness renders strict reproducibility impossible. Debugging is more difficult. If even the small test set (#9) produces different results, new users may not be able to tell whether the software is working properly. When comparing results between versions or after changing parameters, even small differences can confuse or muddy the comparison. And especially when producing results for publications, grants or diagnoses, any analysis should be absolutely reproducible.

Given the size of biological data, it is unreasonable to suggest that random algorithms be removed. However, most programs use a pseudo-random number generator, which uses a starting seed and an equation to approximate random numbers. Setting the seed to a consistent value removes randomness between runs. Allow the user to optionally provide the seed as an input parameter, thus rendering the program deterministic for those cases where it matters. If the seed is set internally (e.g., using clock time), echo it to the output for re-use later.

Conclusion

GW: How to tell if the software is working:

- Test in a 'vanilla' environment, such as another user's computer or a dummy account with none of the settings of the original developer.
- Test with different sizes of data: ridiculously small, small, medium, and large. The software should run on all sizes given some parameter tweaking, or fail with a sensible error message if the data is too big.
- Compare results from multiple iterations that have the same parameters and inputs.(See rule #8.)
- Consider a container - but most of these rules still apply

References

1. Brazas MD, Yim D, Yeung W, Ouellette BFF (2012) A decade of web server updates at the bioinformatics links directory: 2003-2012. Nucleic Acids Research .