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

GW: Write abstract

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, and those numbers are constantly growing as both trainees and experienced practitioners produce new software to support their research.

Typically, this software is initially developed by one person, and may produce excellent results in their hands. But what happens after that student leaves that lab and someone else wants to use it? Everyone with a few years of experience feels a tremor of fear when told, "Use <graduated student>'s code to analyze your data". Often, that software will be undocumented and work in unexpected ways (if it works at all without substantial modification). Equally often, the potential new user winds up shaking their fist and cursing the author's name. She then has two choices: hack the existing code to make it work for her, or start over.

The root cause of this problem is that most of the software researchers produce 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:

- Is kept under version control.
- Can be installed on systems outside the original institution
- Works for users other than the original author
- Has well-defined input and output formats
- Has documentation that describes what its dependencies are, how to install it, and what the options are.
- Comes with enough tests to show that it actually runs.

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

These are all necessary steps toward creating a reusable library that can be shared through a site like CPAN or CRAN, and apply to both closed-source and open-source software. They do not depend on specific languages, libraries, packages, documentation styles, or operating systems. 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.

Note: we do not recommend that these rules be applied to *every* coding effort. The vast majority of code produced in the marathon of a graduate thesis is "throw-away" code that is used once to answer a specific question related to a specific dataset. However, once that little script is dragged out three or four times for slightly different purposes, it may be time to apply ten simple rules for robust software. As the saying goes, not everything worth doing is worth doing right away.

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 any potential users interested in your software. At a minimum, it needs to provide or point to everything a new user needs to get started, where they can turn to for help, and which licenses apply to the software package. Exhaustive details regarding parameters and usage are not usually necessary in a README if they are present in usage (#2), although 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. The description does not need to be long or detailed. 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.

```
Debarcer (De-Barcoding and Error Correction) is a package for
working with next-gen sequencing data that contains molecular
barcodes.
```

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. Especially if you use older versions, include links where the packages can be downloaded. Package managers like apt, pip and homebrew stop offering older packages after a few years. **MT: Is this true of pip and Homebrew?**

If your dependency is to an internal package that is not available on the internet, you have several options depending on the sensitivity of the code in question. If it is plain text, you can add it directly to you repository with appropriate attribution. If the dependency is a binary, we recommend using 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 as long as the manager continues to run. As a last resort, you can place it at an internal location on shared disk, remove all write permissions, and link to it from your README, although this method is heavily discouraged because of the potential for the directory to go missing because of factors outside the developer's control.

Installation instructions: If the software needs to be compiled or installed, list those instructions in the readme. New users may not be familiar with your build system, even if it is `make`. Also mention here if you recommend they use a pre-compiled binary instead through a system such as pip 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 industry 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 here explicitly, listing all the required fields and acceptable values. (You get bonus points if you include a script to convert between 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. However, the definitions of the output files are often missing. In addition to the expected output, software will often produce intermediate files, auxilliary files, and log files. We believe *all* output files should be listed in the README. Log and auxilliary files are often full of valuable information that can be mined for the user's specific purpose. Even if the files are considered self-explanatory. Sometimes your users will need to answer a question of the format, "Does X tell you the percentage of reads trimmed to remove adapter sequences?" and you can check the documentation 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. Both are important in your README. 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 attribute to you. If your software is not open source, include a statement here. Attributions can also contain a list of 'expert' users that can be contacted if new users have problems with the software. (for better or for worse)

- This is readable *before* the software is installed (or even downloaded).
- Should also include (or better yet, point to) the license for the software,

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

so that (potential) users will know what they're allowed to do.

- **GW: example from real software: https://github.com/dib-lab/khmer/blob/master/README.rst**

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

Users who run your software after installation may not have access your well-crafted README (or may not have bothered to read it). Usage information provides their first line of help.

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, especially when printed to standard error instead of standard output (where it can easily be paged), is a nuisance, and is as unlikely to be read as the README file.

Usage should provide all of the information necessary to run the software. It is usually 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 defines the relative location of optional and required flags and arguments for execution, and includes the

name of the program. Arguments in [square brackets] tend to be optional. Multiple periods (e.g. "[OPTION]...") indicate that more than one can be provided.

A text description of its purpose: Similar to the README, the description reminds users of the software's primary function.

Most commonly used flags, a description of each flag, and the default value: Not all flags 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's help with `diff`.

Exit with an appropriate exit code: When usage is invoked by providing incorrect parameters, 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, because requesting help is sometimes used to verify that a dependency is available.

3 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.

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

You know what they say about assumptions*. As soon as you make one assumption about how a software should work, a use case will come along that will require you to change the assumption.

Every useful parameter should be configurable on the command line. Useful parameters are those that a user will need to modify to suit their computer, dataset or application. Providing parameters on the command line increases the flexibility and usability of the program. You may have determined early on 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.

The list of useful parameters is software-specific and so cannot be provided here, but here is a short list of common useful parameters.

- 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.

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 to reduce the length of the execution command as long as any parameters given on the command line override those values. Configuration files should be used in preference to hardcoding the defaults directly. 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 institutions. 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. Since all parameters should be echoed in the results, config files must be cleaned up after execution completes or they occupy valuable disk space. Use a default configuration file and specify all other parameters on the command line.

* They make fools out of you and me.

5 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 filenames 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.

6 Reuse software (without tears).

In the spirit of code reuse and interoperability, developers often want to use software written by others. The tool could be standard for the task at hand or do exactly what is necessary. 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.

The way that the second program is invoked can throw up errors. For example, the program may not be in the user's path, or it may be an older or newer version and produce results differen than expected. Windows users will be frustrated if you invoke bash or sh explictly or use shell-specific conventions like '*' expansion. It is unlikely that all of your users will all be on the operating system and version as you. 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.

We fully support the reuse of software between projects and have some suggestions to reduce the aforementioned pains. First, 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.

Second, 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.

Finally, make sure that you really need the call-out. If you are executing GNU sort instead of figuring out how to sort lists in Python, it may not be worth the tears of reuse.

7 Do not rely on the pre-installation of non-standard packages or libraries unless clearly stated in the documentaton.

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.

A common way to break this rule is to depend on scripts and tools that are installed on the computers the original developer is using, but which aren't otherwise packaged and available. In many cases, the author of a package 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. In future, as use of lightweight virtualization containers like Docker becomes more widespread, it may become common to test installation on a virtual machine.

8 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 be able to tell whether the software is working properly, eroding confidence in the application. 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.

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, which is valuable in case its documentation has fallen out of sync with recent changes to the code itself.

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 something like 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 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 correct output as well as any errors, whether they arise from incorrect input or are bugs. 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[4] 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-RC1. 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 versions, including patches, builds and qualifiers. Common qualifiers include `-SNAPSHOT`, for applications that are not yet stable or released, and `-RC` for release candidate 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,

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

CPAN, etc. Choose the method that best suits the number and expertise of users you anticipate.

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 .