Ten rules of etiquette for scientific code

The most important qualities of scientific code are correctness and performance (in that order), and we rightly burn a lot of calories in their pursuit. Unfortunately we seem to be less concerned with how our software is used, and our tools have a reputation for bad manners. Mostly this is due to sloppiness, tight deadlines, non-existent budgets, and the fact that usability is a distant consideration compared to getting a result for a paper. Having worked in the field for five years now I've observed that there are many little details about how we write and distribute our programs that could easily be improved.

In 2013 Torsten Seemann wrote "Ten recommendations for creating usable bioinformatics command line software", which got me thinking about my own experiences with scientific code. Torsten's list focused on interactive command line usage, whereas my observations are more concerned with how code exists in the scientific computing ecosystem, particularly in the context of large scale High Performance Computing (HPC) centers.

Use a standard license

A standard license provides minimum fuss for users and increases the chances that your software will be widely used. The Open Source Initiative provides a good description of the main open source options.

It is very common for research centers to install software on behalf of their users. Unsurprisingly such research centers (and their parent institutions) tend to be risk averse when it comes to legal matters. A non-standard license is very likely to require vetting by lawyers, which can be a protracted exercise.

Make your license easy to find. The license should be indicated prominently near the start of the user documentation. Refer to the license by its common name (e.g. the BSD 3-Clause License), and keep a copy of the license terms in a text file in the top directory of the source repository.

If your software includes external packages in its distribution then make sure the licenses for those packages are clearly indicated both in the distribution files and in the user documentation.

It may be tempting to take a standard license and add your own little twist to it, such as a "cite my work" clause. *Resist the urge!* Even small adjustments will force research centers to send your license to the lawyers for scrutiny. A polite citation request in the documentation should be sufficient. Responsible users will do the right thing, especially if you make it easy for them.

If your code is not open source then please try to make your license comprehensible to the target users (avoid legalese).

Log progress and diagnostics to file

A log file can be a godsend to anyone who is trying to find out what a program has done, and they are incredibly useful for bug reporting.

Here are some things that are log-worthy:

- The timestamp of when the program started and ended. Timestamps should show the date and time of day down to seconds.
- The version of the program.
- The exact command line used to run the program, which can be obtained from the argument vector. This is useful if you want to run the computation again, especially so in the distant future when you've forgotten the command line syntax.
- Basic statistics about program execution, for example number of data points processed.
- Warnings and error messages.
- Milestones in the computation process.
- Exact copies of any shell commands called by the program.

Most programming languages provide logging libraries, such as logging for Python. Good logging libraries have lots of features that are tedious to implement yourself, such as:

- Automatic timestamping of messages.
- Customisable logging levels such as info, warning, error and so forth.
- Proper handling of output buffering so that your log messages arrive in the file in a timely manner.
- Support for interleaving logging messages from concurrent processes.
- Rolling log files, which are useful for long running applications.

You may also consider allowing the log messages to optionally appear on the standard error device (`stderr`). I think a file is generally the better option because it won't get lost by some accidental output redirection, and you won't clutter `stderr` with junk. While I think it is good practice to write error messages to the log file, I think it is also important that they also appear on `stderr` for the benefit of interactive command line users (see below for more thoughts on error messages).

Standardise on data formats

Standard data formats enable interoperability. In the very least you should try to avoid inventing your own ad-hoc formats when suitable alternatives already

exist. Unless your software has special I/O performance requirements, text files are preferable over binary files because they are friendlier to humans and Unix command line utilities such as `grep`. If you have special data requirements that are not well suited to text then consider HDF5.

CSV (comma separated values) is recommended for tabular data because it is relatively simple and plays nicely with spreadsheet applications. Most programming languages provide CSV libraries, such as the csv library in Python. It is good practice to provide column headers in CSV files because they allow you to refer to data by attribute name instead of column number (with support from your CSV library). Column headers are more robust than column numbers in in the presence of column reordering.

XML was all the rage a few years ago for representing (semi) structured data, but it has fallen out of fashion in some circles because it can be space inefficient and not so friendly for human consumption. JSON and YAML are human-friendlier alternatives which are quite good at representing nested data structures. See the YAML documentation for a comparison of all three formats. Despite its downsides, XML has some useful characteristics such as schema validation (plus many standard schemas) and powerful document transformation technology such as XSLT. If your data is more like a document than a data structure then XML may still be a better choice.

Use meaningful version numbers

Version numbers allow users to track the provenance of their work. This is particularly important in science where reproducibility is a primary concern. Torsten already mentioned the need for version numbers in item 3 his article. I want to reiterate that point because it is so important and so frequently overlooked in scientific code. I also want to add that if you are not sure how to go about versioning then consider the approach of Semantic Versioning.

Strive for context independence

Research centers will often want to install your code from source, and they will generally want to install it in a location of their own choosing. Ideally your program's behaviour should not be context dependent:

- Do not hard-code paths to files (item 6 in Torsten's list).
- Do not make assumptions about your software's dependencies (item 9 in Torsten's list).
- Do not assume that your program will be installed in a specific location.
- Do expect that your program will be run by multiple users concurrently.
- Do allow multiple different versions of your program to be installed on the same system without interfering with each other.

Where possible you should follow the prevailing software installation conventions. For example, if your program is written in Python then you should make it a Python package that can be installed with `pip`. If your program is in a compiled language, such as C, then you should use a build tool such as make, and ideally a configuration tool such as autoconf and related autotools. You may also consider CMake if building on multiple platforms (such as Windows) is important for your project.

When developing and testing your code consider using a sandboxing environment such as Python's virtualenv. This can help ensure your application does not depend on idiosyncrasies of your development computer.

Use and document exit status values

It is quite common for scientific code to be used as part of a *pipeline* (or *workflow*), where the outputs of one computation are fed as inputs to another. Such pipelines can become large and complex and can run for hours or days on HPC systems, therefore the likelihood of failure is high. One of the worst things that can happen in a pipeline is for one stage to fail and produce partial or incorrect results but for subsequent stages to carry on oblivious to the error. This can lead to data corruption, or much worse, it can produce false results which go undetected. Most pipeline systems know very little about the programs they run, so they rely heavily on the exit status of each computation to decide how to proceed. The Unix convention is that an exit status of zero means that the computation ran to completion successfully, and any other exit status means something exceptional happened. Therefore one of the most heinous sins you can commit in scientific programming is to return a misleading exit status code, especially so if you return zero when your program failed! You should be careful to only use a zero exit status when your program has successfully run to completion. Furthermore you should use different exit status values for different kinds of errors and be sure to *document what they mean*.

You can minimise the risk of false exit values by reducing the number of exit points in your program and by defining exit codes as constants in their own module.

Generate informative error messages

Few things are more infuriating than cryptic error messages; they tell you that something went wrong, but little else. Good error messages provide information that can help the user to find and fix the problem.

Consider the case of a missing file. What might the user need to know?

- What was the name of the file that was missing?

- Where did the program search when it tried to find the file?
- How can the user influence the places that are searched?

Your program might be called from a pipeline or a shell script where its output, including errors, are mixed together from other programs. Therefore your error messages should include the name of your program to clearly identify the source of the error.

Here is a template that caters for many kinds of errors in an informative way:

```
program-name ERROR: general description of error

    What happened.

    How the user could fix the problem.
```

for example, assuming the program is called `frobnicate`:

```
frobnicate ERROR: could not find configuration file.

    Tried to read configuration file /home/foo/frobnicate.config
    File does not exist.

    Create a configuration file called /home/foo/frobnicate.config
    or specify an alternative path with --config <filename>
```

Give your users naming rights to their files

For maximum flexibility you should always let users name their input and output files. This is especially important in pipelines where we want to reason about the dependencies between computations. Implicit files act as a kind of hidden state which is invisible to the pipeline infrastructure. This makes it difficult to fully parameterise each stage of the pipeline, which in turn complicates many important pipeline features such as:

- Parallelisation of stages.
- Pipeline restarts which avoid the re-execution of up-to-date computations.
- Automatic dependency determination.

If your program reads or writes lots of files then it is unreasonable to require the user to name them all individually. Unix command line wildcards will often suffice for inputs, but you will still need to provide a mechanism for naming outputs. In such circumstances you should let the user specify the naming

pattern used by the files, and perhaps also allow them to specify a directory containing the files.

Of course default filenames are often convenient, especially for interactive use, but you should always let the user override the default.

Follow command line argument conventions

Over the years various conventions for command-line argument syntax have developed, most notably the POSIX guidelines, which are recommended by the GNU Standards for Command Line Interfaces.

As Torsten says most languages provide a `getopts` library which makes it easy to follow the POSIX conventions. Some languages provide even more featureful libraries, such as argparse in Python.

In the very least your program should provide these two arguments:

- `--version`: for printing the version number of your program. Other programs will want to parse the version number, so just print the version number and nothing else.
- `-h` and `--help`: for printing usage information. Each command line argument of your program should be explained in simple terms. Try to avoid jargon and technical terminology. Show default values and any constraints that may be imposed on arguments.

All argument "flags" should start with a dash character, a single dash for a short flag and a double dash for a long flag. I recommend providing short and long flags for each argument, such as `--log` and `-l`. The long form is more readable and meaningful, whereas the short form is more convenient for interactive use.

Options with flags should be able to appear in any order on the command line. The GNU standards also provide the `--` (double dash) flag which marks the end of options; anything following is taken as an operand regardless of whether it starts with a dash or not.

When deciding what your flags should be called it is a good idea to consider what has been used before. The GNU project provides a (probably incomplete) list of long options which have been used in their programs.

Provide test data and a small worked example in your user documentation

Test data and worked examples give your users a leg-up in getting started with your program. In research centers the person installing your program is often a system administrator. A worked example provides them with a quick and

easy way to check that your program is working as expected, and increases their confidence that the installation went smoothly.

Ideally test data should be sufficiently small that it is not onerous to download and test computations don't take too long. However, it should be sufficiently large to exercise the key parts of the program.