Lightweight guidelines for code and data sharing

Computational Cognitive Neuroscience Lab, Harvard University

1. **Data format**. Most kinds of behavioral data can be arranged into what is sometimes called *tidy data*. A single file (e.g., in CSV format) contains the trial-by-trial data for all subjects. Each row is a single observation (e.g., a trial), and each column is a variable. For example, the column headers might look like:
    a. Subject
    b. Trial
    c. Block
    d. Stimulus
    e. Choice
    f. Outcome
    g. RT
    h. Age
    i. Gender
2. **Software**. If your data are in tidy format and you use R, then you can exploit the functionality of the *tidyverse* packages. There are also tools in Python.
3. **Language-independence**. Because different people use different programming languages and packages, data should be stored in a language-independent format, such as CSV or TXT files. In some cases, it makes more sense to use JSON as a data format, which is easier to work with for certain types of data (e.g., fields containing lists are loaded as strings from CSV files, which requires additional processing, but stay as lists when loaded from JSON files). Another benefit of JSON is that you can easily load a JSON file into a data frame and have all variables/columns and data types (factor, numeric, etc.) with no need for writing extra formatting code.
4. **Repositories**. Both data and code should be stored in a supported repository, such as GitHub. In some cases, data sets are too large and then there are a few other options. For example, if you're dealing with fMRI data, there is OpenNeuro. Repository organization is important. One fairly general approach is to organize repositories into the following sub-directories:
    a. Data (where the raw data files live)
    b. Code (analysis and modeling code)
    c. Results (code outputs that you want to save)
    d. Figures
    e. Docs (notes, manuscript drafts, bibliographies, etc.)
5. **Reproducibility**. All analyses and figures in a published paper should be reproducible with the code. Importantly, you have to check that all the necessary files and dependencies are in the repo, or otherwise provide instructions for how to obtain them (e.g., if the user has to install certain libraries). As much as possible, make code self-contained: don't rely on certain variables being preloaded in the workspace. Make sure they're defined in the functions themselves.

    a. One way to deal with software dependencies is using a package manager like Conda. This allows someone to write all of the package dependencies to a file, which can be downloaded by others when they download the project repository. Then all someone has to do is `conda env create <list_of_dependencies_name>.yml` to install all of the packages, instead of figuring it out on their own. It has the added benefit of making projects easy to develop on a cluster; in the event you need to use a cluster you can use the above method to install the needed packages easily in one line of code on the cluster, instead of manually installing each one.
    b. In R, there is similar package management functionality provided by packrat.
6. **Ease of use**. Make reproduction of analyses and figures as easy as possible. In particular, you should have a function like plot_figures(fig) which takes the figure name as input and generates the figure that appears in the paper. So for example plot_figures('fig1') should generate figure 1. In addition, this function should produce all the statistical analyses associated with the figure and display them in an interpretable way. For example, if the paper reports a t-test, the function should display the results of the t-test in a standard format, such as t(35) = 2.15, p = 0.003. Finally, make sure that the repo is well-documented: explain what's in it, what each function does, how to get started (sometimes tutorial code is useful).
7. **Front-end vs. back-end functions**. Front-end functions such as plot_figures should not take as input lots of variables, which the user may not know how to specify. There can be back-end functions that implement the underlying computations. These should be as modular as possible, to facilitate reuse across projects.
8. **Code annotation**. As much as possible, annotate your code! You might not understand what your code is doing 6 months from now. In addition, each function should have a commented header explaining how to use it (what the inputs and outputs are, what the function does, etc.).
9. **Variable naming**. Make your variable names informative so that it's easier to keep track of what they mean.
10. **Beware of notebooks.** Some of you like to code in notebooks like Jupyter. This is very useful for code prototyping, but it can be a disaster for reproducibility if you're not careful. See here for more guidance.

Further resources:

- The good research code handbook