Reproducible Research:

Best Practices for Data and Code Management

Erica Chuang, Harrison Diamond Pollock & Stephanie Wykstra

NOVEMBER 2015

Introduction

Journals, research funders and research groups such as Innovations for Poverty Action are increasingly recognizing the value of research transparency. Research transparency includes pre-registering studies and sharing materials such as data and code to allow others to re-analyze the reported results.

Proper data and code management during a project are essential for transparency after a project's completion. They are also important for internal use, as projects often run for multiple years, with several staff members working on them sequentially.

This guide outlines best practices in data and code management. The scope of the guide is to cover the principles of organizing and documenting materials at all steps of the project lifecycle with the goal of making research reproducible. The guide does *not* cover best practices in designing surveys, cleaning data or conducting data analysis. In each section, we explain the "what," "why" and "how" of each recommended practice.

For comments/questions, please contact **transparency@poverty-action.org**.

Quick Guide to Data and Code Management

I. **Folder and File Structure**
- Keep groups of files (such as all code, all survey instruments, all datasets, etc.) in separate but clearly labeled folders.

II. **Best Practices for Coding**
- **Variable names and labels** – Variable names should be clear to not only you, but (hypothetically) a stranger who randomly looks at the data (e.g. Age of respondents, question 5 in the survey = q5_age).
- **Missing values** – It's important to distinguish between a known non-answer (e.g. a respondent says 'Don't know' or 'No response') and an actual missing value. Known non-answers should be marked with standardized values (e.g. Don't know = .d, No response = .r).
- **Writing your code**
    - i. **Header** – At the top of your do-file, not only include your name, the project name, and the purpose of the do-file (in comments), but standard starting code such as *set more off*, *version*, and *log using filename.smcl*.
    - ii. **Body** - Make sure to maintain good organization of your code – this means commenting (using *, /**/, or //) at every part of the code where an important set of code is being run, such as merging and appending of datasets, different models being run, even notes to yourself to look at later!
    - iii. **Footer** – At the end of your do-file, always close your log file using *capture log close*, exit your dataset or Stat using *clear* or *exit*, and have a couple of blank lines at the end so that the do-file runs without error.
    - iv. **Master** – You can run all of your code at once with a Master do-file! Just type the command *do* with your do-file name and watch it go!
- **Organizing your code** – Name your code something clear and related to the purpose of the code (e.g. projectname_master.do for your master do-file). Put it in the appropriate folder within your folder structure, which is ideally in a folder named "Code".
- **Relative references** – To make it easier to change where datasets and code are called from and output files saved to, use relative references such as *cd C:\User\ProjectFolder* or the *global* or *local* macro commands.

III. **Code and Data Documentation**
- Keep a constantly updated list of all datasets and code associated with the study, the type of data each file contains/processes and for what purpose they were created.

IV. **Data Security and PII**
- Take care to remove any PII from your datasets as soon as possible over the course of the project. This includes respondents' and affiliates' names, phone numbers, location (smaller than district), account numbers, online usernames, license numbers, etc. For files that contain PII, keep them under encryption using Boxcryptor.

For more details, see the full Best Practices Guide on the following pages!

Table of Contents

I. Folder and File Structure

What?

Folder structure outlines the organization of folders containing all project files. Typically, folders are designated for certain types of files (e.g. do files, data files, tables, etc.) as well as points in time (e.g. baseline, midline, endline).

Why?

Maintaining a logical folder structure means it is much easier to find the files you are looking for. It also greatly facilitates sharing a project with others, such as the principal investigators (PIs) and new RAs working on a project. Having a disorganized set of files means project transitions become much more difficult.

How?

Here is an example folder structure which outlines a logical division of files for an entire research project, and here is a recommended folder structure for data and code files.

Regardless of the specific folder structure you decide upon, it is critical to **define it in advance**, and to limit the files you need to keep (especially datasets) to only those that are strictly necessary. If you want to keep extra/old files, it is best practice to create an 'Archive' folder where these are stored. See more below.

What about inheriting an existing folder structure? While the ideal case would be to re-arrange files and folders to fit with a structure like the one outlined above, it would likely be an impractical and time consuming step. Before making any changes, an RA would need extensive understanding of the current structure and relationship between files. Creating a **folder map** is a great practice in such a case. This map (in Excel/Word) would outline the key folders in the structure and what files are found within them. Such a document would form the basis for the project's ReadMe file, and will prove useful throughout the project lifecycle.

Note: Keep file and folder names simple and, whenever possible, without spaces.

II. Best Practices for Coding

NAMING AND LABELING VARIABLES

What?

Variable names should be in a consistent format. Labels should be applied to all variables in a consistent format. Value labels are also necessary for some variables.

Why?

Without clear names and labels, it would be very difficult to understand your data. They provide essential information about variables, without having to look back to how they were constructed. Including question numbers and/or text in labels allows one to link the dataset with the original questionnaire.

How?

Variable labels should be attached to all variables once they are imported into Stata. This can be done in several ways. If you are running an ODK/SurveyCTO survey, using the odkmeta command will label your variables with the corresponding label you've defined in the ODK form. Otherwise, you can define labels manually in Stata. In addition, don't forget to **label the variables you create after the initial import**, such as during data cleaning and analysis.

See this resource on variable labeling for more on proper labeling conventions. In summary, there are two primary options for labeling in relation to the variable name. While either can be used, be sure to choose one format and stick with it throughout.

1) Assign a short descriptive label, while the variable name references the question number from the questionnaire
2) Assign a short descriptive label including the question number, with the variable name being descriptive itself

Note: If a variable comes directly from a questionnaire, a clear link should be made between the associated question text and the variable. Including the question text in the label could work well. However, there is an 80 character limit on label length. In such a case, we recommend including questions text as a variable **note**.

Assigning proper variable names is also important. As outlined above, variable names can take several forms. If you need to rename variables, Stata's *rename* command offers much flexibility in doing so. See the Stata 102 training module on Naming and Labeling for more. (Note: Stata modules not yet publicly available).

Stata 102 also discusses **value labeling**. It is critical to assign value labels to coded numeric variables. For instance, it is common to have questions that ask about occupation, but codes (numbers) are entered instead. Assigning a value label is necessary for interpreting output so as to make the link between the underlying number and the occupation it represents.

DEALING WITH MISSING VALUES

What?

Missing values will likely appear often in your data: they are usually represented in Stata by a dot (for numeric values) or double quotes (for strings). Missing values may result from several sources, such as skips in the survey, "don't know" or refusals to respond. It is critical to ensure these values are standardized across your datasets. In addition, it is important to always be aware of missing values when coding and convert to extended missing values (e.g. .d, .r, .n) where possible.

Why?

Keeping track and standardizing missing values can mitigate numerous coding problems. In Stata especially, many bugs or errors are the result of failing to realize how missing values might affect analysis. Using extended missing values allows one to attach value labels to them, greatly facilitating understanding of what types of responses the missing values refer to.

How?

First, browse IPA's guide to standardizing missing values. The key takeaway is to convert all don't know/refusal/NA values to the "IPA standard" extended missing values as .d, .r and .n. Note that this does not apply to any values in the survey which were correctly skipped as part of a normal skip pattern, which remain as dots for numeric variables.

Other key points:

- Remember that in Stata, all missing values are considered to be bigger than all non-missing values [all non-missings < . < .a < .b < .c < ... < .z]. This means that you must be extremely careful when recoding and replacing values. For instance, if you recode values that are greater than a given number, you'd also be recoding any missing values as well. **Always be cognizant of missing values when coding!** Be careful to closely read the help files of egen functions as missing values can be interpreted in a manner that can greatly affect your results (e.g. rowtotal)

- It is critical to clearly outline any rules dealing with missing values in your do-file so changes can be checked for consistency. For instance, include a comment at the top of a do-file stating "Values were recoded to zero in cases x and y."

- Extended missing values (unlike regular missing values) can be value labeled. This is highly recommended given the additional information this provides.

WRITING DO-FILES

At the top of every do-file you must include:

- Your name and names of everyone who has modified the file with contact information.
- The date the do-file was created as well as the date it was most recently modified
- The purpose of this particular do-file: what does it do?
- *version 13.1* (or whatever version of Stata you are currently using); this ensures that your file will be compatible and replicable in future versions of Stata. This should be written as a command, not a comment.

We recommend that you also include these elements in a code 'header':

- List the data the do-file draws upon as well as any datasets that it produces.
- *set more off* : this allows do-files to run un-interrupted; otherwise, Stata will periodically stop and ask if you'd like to see more output from the currently running program.
- *log using 'X':* Creating a log file which captures all output from the resulting do-file. [Note: it is also common practice to include a line of code that closes any open log file, with *capture log close,* to prevent Stata from encountering an error if a log file is already open].
- Set the working directory by using *fastcd, cd***,** or globals. Working with file paths will be discussed more below.

See this do-file for an example of a good header: example_header.

We recommend that you also include these elements in a code 'footer':

- *capture log close*: this will ensure that Stata stops logging at the end of your do-file.
- *Exit:* this will stop the do-file from running.
- A few blank lines: Sometimes Stata does not recognize the last line of a file, so always include a bit of spare blank space.

In the body of every do-file you write, you should:

- Organize code into sections: with each discrete task (e.g. renaming, working with unique IDs, running regressions) clearly labeled with a comment header. This is essential for locating and organizing key elements in your code (for comparison think of book chapters). A great practice to help with organization is to use **pseudocode -** where you write a half English/half code version of your do-files before writing the formal code. See this document, Pseudocode 101,[1] for more information about what these code snippets should contain.
- Make comments often, whenever either the code itself or the motivation is unlikely to be self-evident. If you're not sure whether or not to comment, definitely err on the side of commenting. Concentrate on the **how** and the **why** of what is being done in the code. Keep in mind that what is obvious to you may not be obvious to either your future self or others who use your code!
- Create comment markers at "important code events." These markers can be any unique set of characters that is easily searchable, e.g. //decision. You should likely mark the following important events:
  - Creating/exporting a table, graph or chart
  - Merging and Appending
  - Reshaping and Collapsing
  - Dropping duplicates, changing of unique ID
- Moreover, it is critical to mark what the **expected result** is from these important code events. For instance, what percentage of observations in a merge should match between datasets, which should be only in the using dataset, etc.? The idea here is that whenever an important change to the data is made, the expectation behind it should be clearly set out so it can be easily verified and understood by subsequent users.

[1] Introduction to Computer Science and Programming Using Python, MIT 6.00.1x. Accessed February 2015.

- Mark points in your code that you need to revisit, e.g. a decision that will need to be made later. An example marker: /!\

See this do-file for an example of well commented code. Notice the different ways in which comments were used, the marking of key events, issues to revisit, and explanations for why decisions were made.

HOW TO WRITE A MASTER DO-FILE

Creating a master do-file is essential to outline and run all of the project's code. Here are the key points to consider:

- The master runs all do-files that are contained in your project or a specific subsection of your project. For instance, you might have a master cleaning do-file, a master analysis do-file and so on, and a master project do-file that runs all of these.
- The master describes any relevant general information about the project and what each do-file within it does.
- All user-written commands that were used in the code should be listed along with code that installs each command. Remember that it's likely you've used a number of user-written commands that had been previously installed. These should be listed as well.
- As with any do-file, the master do-file should include standard elements in a header, as outlined above.

Here's an example master do-file: example_master.

HOW TO ORGANIZE AND REFERENCE DO-FILES

The manner in which your code files are organized is critical to its understandability and will greatly ease your work as the number of code files grows.

- There is no single perfect folder structure, but having one that is logical and that you use consistently is essential. Here is an example.
- These folders imply a clear distinction between the analysis portion of the project and the data management/cleaning portion. Within each, code files are categorized based on the round of surveying they are associated with, with a master do-file for each section.

Useful Tips:

- When do-files have to be run in order, be sure to number them appropriately, (e.g. 0_master.do). If the order does not matter, make a note of this in the master do-file.
- Use an "archive" folder to hide all old files from the working directory, so as to limit confusion about which do-files to use.
- Don't generate "junk" datasets – keep the minimum output necessary. If you want a dataset for temporary use, you can generate a temporary dataset that will disappear at the end of the program.
- Keep the raw data in the purest form possible, this means:
    - Import directly into Stata in the form the data came in – e.g. import excel if the file is an excel file – don't save as CSV first.

    - Clean as much as possible within Stata – don't make one-off changes to the original file. The key is to keep all cleaning decisions replicable, i.e. in your do-files.

USING RELATIVE REFERENCES

It is essential to use **relative references** in your code rather than absolute references. An absolute reference is the *entire* file path such as "C:/My Documents/My Project Folder/data.dta". It is never a good idea to use these absolute paths given that any other user would have to change every path to correspond with their own computer to make the code run.

Relative references involve creating some sort of shorthand that Stata recognizes to mean a certain location, and to then only change that one line when switching users, moving data, etc. Subsequently, all new locations are made relative to that initial location.

There are a few ways to use relative references:

1) We can define a local or global macro to set a particular path and then use the macro name throughout to refer to a set path.

For instance, we might define a global:

global path C:/My Documents/My Project Folder

And then whenever we call a particular file, we include the macro 'path' within it, say:

merge using "$path/data/baseline_clean.dta"

2) We can use *cd* or *fastcd* (from SSC) commands to set the working directory. All folders you refer to from then will be in made in relation to that folder.

Setting the cd:

cd "C:/My Documents/My Project Folder"

And then when you refer to files:

merge 1:1 using "data/baseline_clean.dta"

Note: If you are commonly working across users, you may use the following code to automate the directory switch:

```
if c(username) == "user1" {

    cd "C:/Users/user1/analysis/do"

}

else if c(username) == "user2" {

    cd "C:/Users/user2/analysis/do"

}
```

III. Keeping Good Code and Data Documentation

What?

Keeping your code and data well documented means keeping all files associated with your project organized, and tracking all key decisions with regard to them. It also means describing the relationship between your data and code (e.g. in a ReadMe file) and any other important files.

Why?

Good documentation is essential any time someone besides the original author wants to understand or extend existing work. Even for the author, it is often difficult to keep track of files that may have been created or decisions that were made even a few weeks prior. Keeping a good record of these issues will greatly aid in alleviating future problems and make a project much more likely to be used by others in an effective manner.

How?

One key aspect of good documentation is a ReadMe file. This file (often in PDF or Word) can take a variety of forms and lengths, but at minimum, should describe the key code and data files of the project and how they interact. Here are some sample ReadMe files. Additionally, ReadMe files can include information on any user-written commands used (also best practice to include these in a master do-file as part of the header) as well as guidelines on interpretation of particular variables or processes. Note that project implementation/logistics information should also be tracked, but kept in a separate document from information about data and code.

While you usually would publish a ReadMe file along with data and code, **it is critical to make the ReadMe a living document** that you update throughout the lifecycle of the project. This 'living log' should be updated regularly with important notes about files and variables as needed. As emphasized, this makes working on an existing project much easier, along with being a key resource when sharing.

Good documentation is also made much easier with effective file organization. Refer back to the section above – Folder and File Structure – for more.

Particularly for collaborative work on code, consider using GitHub. **GitHub** is a web platform that facilitates collaborative code writing, as well tracking versions and tasks. The platform can greatly facilitate keeping track of files and information about them over time. IPA has compiled some tools to learn and use GitHub, which can be found here. (Notice our GitHub resources are on GitHub!) Navigate to the GitHub User Guide for guidelines on how to get started with the platform.

A few important features to note:

- Groups of files can be 'tagged' at any point in history, so that one could easily return to, say, the files as they were when they were first submitted to a journal for publication.
- Notes can be attached, and conversations can happen between multiple users on individual files and/or lines of files, making it easy to browse important facts about a project.

- Files can easily be compared against one another so that changes made over time can be visualized. When files are uploaded ("committed"), descriptions of changes can be attached so that there exists a record of one's thought process at the time of making the change.

- GitHub can be used for narrowly defined tasks (e.g. code checks) or can be used more comprehensively to track all project files.

Please contact IPA Research Support at researchsupport@poverty-action.org if you are considering using GitHub, as we'll invite you to join PovertyAction, our organizational GitHub account (for IPA staff and affiliates only).

IV. PII and Keeping Data and Code Secure

What?

Personally identifiable information (PII) refers to data that can be used to link survey data with the survey respondents. PII can be a single variable (e.g. name, or address) which directly identifies individuals ("direct identifiers") or multiple variables that when combined can identify an individual with a reasonable level of confidence ("indirect identifiers"). It is absolutely essential to keep PII secure at all times – from when data is initially collected to data analysis. Under no circumstance can PII be shared with anyone who has not expressly been given permission (within an IRB) to view it. All IPA staff are required to take our data security course.

Why?

Keeping PII secure at all times fulfills a fundamental responsibility of research: to ensure the confidentiality of our respondents.

How?

All files which contain PII must be encrypted at all points of the data flow process – from point of origin, to storage on local devices, to the cloud. IPA recommends using the BoxCryptor software to encrypt such files (and folders).

Whenever possible, PII should be stripped from your data and/or do-files. See this document for more detailed instructions on best practices for how to do this in Stata. There can be no better substitute for ensuring PII is removed than having a full understanding of the dataset you're working with. It is always preferable to err on the side of caution – if a variable (or a set of variables) might even plausibly identify particular respondents, they must be removed from unencrypted files.

Examples of PII, from IPA's Data Security Manual:

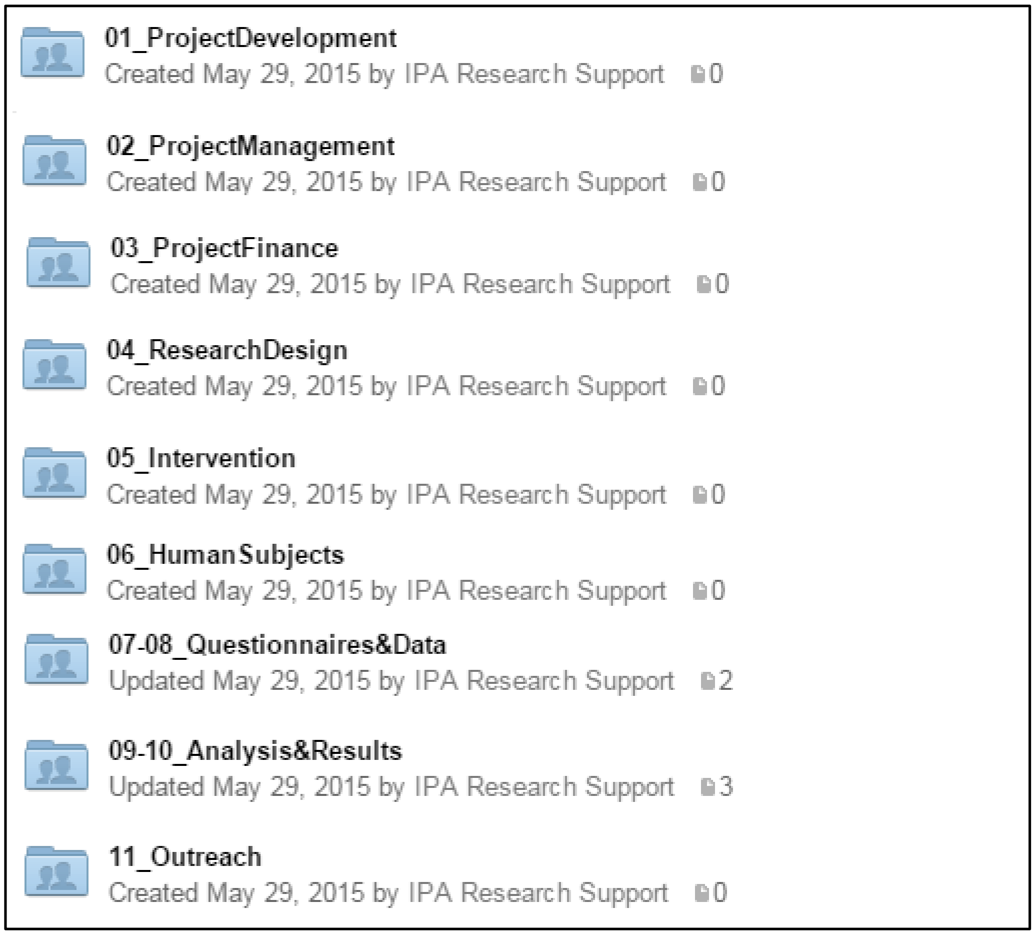
- Names
- All geographical subdivisions smaller than state/province
- All elements of dates directly related to an individual (except year)
- Telephone numbers
- Device identifiers and serial numbers
- Biometric identifiers including finger and voice prints
- Fax Numbers
- Email addresses
- Social security numbers
- Medical record numbers
- Web URLs
- Full face photos
- Health plan beneficiary numbers
- Account numbers
- Certificate or license numbers
- Vehicle identifiers and serial numbers
- Internet Protocol address numbers
- Any other unique identifying number, characteristic or code (use your discretion)

At what stage of the project should PII be removed?

PII should be removed at the earliest possible stage, and the files that are used should be free from PII to eliminate risk. However, if it is necessary to leave PII in the dataset for analysis, then these variables should be noted and work should be done within an encrypted folder. It is essential to remove all PII before sharing the dataset for publication.

Appendix: Folder Structure

Overall project folder structure



01_ProjectDevelopment
Created May 29, 2015 by IPA Research Support 0

02_ProjectManagement
Created May 29, 2015 by IPA Research Support 0

03_ProjectFinance
Created May 29, 2015 by IPA Research Support 0

04_ResearchDesign
Created May 29, 2015 by IPA Research Support 0

05_Intervention
Created May 29, 2015 by IPA Research Support 0

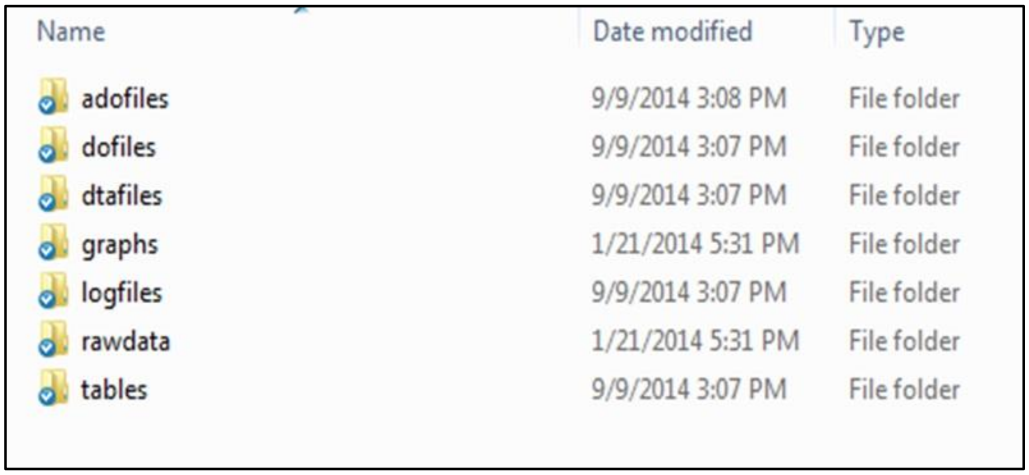
06_HumanSubjects
Created May 29, 2015 by IPA Research Support 0

07-08_Questionnaires&Data
Updated May 29, 2015 by IPA Research Support 2

09-10_Analysis&Results
Updated May 29, 2015 by IPA Research Support 3

11_Outreach
Created May 29, 2015 by IPA Research Support 0

Data and code file structure:



Name	Date modified	Type
adofiles	9/9/2014 3:08 PM	File folder
dofiles	9/9/2014 3:07 PM	File folder
dtafiles	9/9/2014 3:07 PM	File folder
graphs	1/21/2014 5:31 PM	File folder
logfiles	9/9/2014 3:07 PM	File folder
rawdata	1/21/2014 5:31 PM	File folder
tables	9/9/2014 3:07 PM	File folder