=5 =3 =9

Last Update: September 27, 2022

Principles of Quantitative Project Workflows

Data Management and Analysis

Document created by:

USC Dornsife
Center for Economic and Social Research

635 Downey Way, VPD, Los Angeles, CA 90089
https://cesr.usc.edu/care

Contents

1 Introduction

The purpose of this document is to outline and codify some general principles for working with data on projects with a quantitative component. Below, we elaborate upon three principles that should systematically and consistently guide how each analyst approaches quantitative analyses. They are:

- Replicability
- Transferability
- Accessibility.

We explain the meaning and rationale for each of these goals below. And, then, we list some expectations and overarching principles. Finally, we discuss how to best organize information (that is, folder, file, and variable naming conventions to better achieve these principals/goals). At the end, we provide an introduction to the Center for Applied Research in Education's (CARE's) computing environment, and a quick synopsis of `Stata` packages written by contributing researchers that may automate or simplify some repetitive procedures that commonly arise on our projects.

This document is inspired, and borrows heavily, from similar guides for applied researchers, including Matthew Gentzkow and Jesse Shapiro's Code and Data for the Social Sciences: A Practicioner's Guide and Julian Reif's Stata Coding Guide.[1]

2 Overarching Goals and Rationale

2.1 Some Inviolable Commandments

- Never overwrite source data.
- Never store files that contain personally identifiable data on your local machine. In fact, try not to save any data on your local machine.
- Try your best to be wrong internally before being wrong externally.
- When output or results do not look correct (e.g., odd counts, ranges, means), always investigate with a skeptical mindset. Recheck your code, examine the raw data, and establish additional examination points within your code. Hold the mindset that

[1] This document is narrowly focused on principles and rules for developing and maintaining transparent and replicable quantitative workflows in a collaborative environment. For a similar document that codifies analytic choices, see Donald Green's Standard Operating Procedures handbook for experimental designs.

everything you have done is wrong, until you have confirmed that this is not the case.[2]

- Be intensely curious and thorough about examining the raw data.
- Be assertive!
- Never ignore the results from a merge. Mismatches can be the result of missing observations, ID issues, or other ways in which the data are not structured as you expected[3]
- Always log the results of each session that manipulates a dataset or generates a calculation used in a document that's shared externally. The `preamble` .ado written will help institutionalize this rule in your workflow.
- Never haphazardly drop an observation.
- All calculations in a report must have a code footprint.
- Don't copy and paste results from a statistical program to a document that is shared externally.
- Be sure to set the seed (and, in `Stata`, your -sortseed-) at the top of each script.[4]
- In the script preamble, be sure to indicate the creation date, author, and purpose of the syntax file, and maintain an update log with a concise note summarizing the reason and date for a substantive change in the source code.
- Where possible, rely on legacy code to perform procedures that have already been done.
- If you're not sure, just ask.

2.2 Replicability

- Workflows and results generated by one analyst for one project should be easily and accurately replicated by another analyst. This principle requires that the principal analyst establishes a clear, documented workflow, and each procedure that leads to a transformation of variables, or destruction of data, is recorded in a syntax file.

[2] To facilitate this type of scrutiny, a good habit is to generate tables of summary statistics (e.g., minimum/maximum values, missing patterns) on key variables, disaggregated by natural grouping structures in your data (e.g., school year, state). This can help identify programming or coding errors before you begin analysis.

[3] The `R` package `tidylog` provides functionality similar to `Stata`'s for summarizing merge results in the `tidyverse`.

[4] Seeds and particularly sortseeds should be set immediately prior to the command requiring a stable random number or sort is used. Since pseudo random numbers come one by one from a list created by an algorithm using the seed, if commands which utilize a random number (for breaking ties in sorts or otherwise) any commands added between setting the seed and the desired command could change the result.

- Although we aim to review each other's code for every project, we typically only adhere to this principle consistently for projects with a complicated data management or analysis component, for projects with a high level of external visibility, and for new researchers. But, it is our goal to continue to make this more systematic, particularly as we add new staff.
- Each analyst should be able to replicate the results another analyst calculates in any software (with some decimal-place types of exceptions, particularly for more sophisticated procedures).
- For code review, the assigned QA researchers should denote sections requiring the attention of the primary analyst with a unique symbol that will allow the analyst to quickly control+F to find sections that require review. Below, is an example from a `Stata` .do file that was QA'd by Danial Hoepfner.

```
/*
Created by: MWG
Creation date: 4/15/2019
UPDATE LOG:
*!DH Note 5/30/2019: QC by DH look for *!DH for comments
*!DH Note 5/31/2019: More QC
*/
*!DH Note 5/30/2019: I don't have this scheme
if `""`c(username)'"" !=`"dhoepfner"' preamble, ///
        clientf(relsw_2017/sped_transitions) ///
        log mkdir sub(output syntax results documentation ///
        final raw csv converted zip clean) ///
        scheme(plotplainblind_rel) ///
        logpath(output) logname(${rq})
```

- To the extent possible, the methodological lead or primary analyst should enumerate QA steps that should be performed at different waypoints throughout a project workflow.

2.3 Transferrability

- Code should be transferable across projects and across individuals. That is, if one analyst left tomorrow, another one should be able to pick up her code from a given project and use it.
- Be sure to annotate any program or package dependencies in the preamble of your code. In `R`, this means loading all packages used in the preamble.
- Annotate your code richly, throughout each section, so that a future analyst (perhaps you!) can easily understand what was done (e.g., the workflow, processes), and why.

Remember that what is apparent to you while you are in the middle of a project may not be apparent to a future analyst (or your future self!).

- If one analyst develops code for preparing files for a project, another analyst should be able to use this code for similar projects and problems, or even borrow components of it for similar procedures for different projects. Clearly structuring and annotating your code will facilitate this type of sharing.
- For multi-year projects, include the date (with year) with the comment so future analysts can see which comments are most recent.
- In `R`, never `setwd` to a directory on your local machine.[5]

2.4 Accessibility

- Code should be annotated and grouped by data management and analysis procedure.
- Ideally, some sort of structure should be created for your syntax files so that it flows logically, intuitively, and clearly.
- Sections of code should be clearly demarcated, such as "begin cleaning file X","begin cleaning file Y", "merging Y and X".
- Use informative names for local and global macros and variables, even if temporary (unless created and deleted within a few lines of code).
- If any sort of cleaning occurs outside of syntax, for example using an excel sheet to organize variable names across files, annotate that process in the script extensively.

3 Expectations and Guiding Principles

- Code should have a structure or outline that (roughly) follows the workflow of analytic project (see Section 4.2.1 below for more on this):
  - Data import/conversion from .txt/.csv/.sasbdat/etc. to format of the chosen software
  - Data cleaning and managing
  - Labeling and variable recoding and construction
  - Analysis
  - Presentation and output
- Anything that destroys, manipulates, or changes a file in any way should be recorded in code

[5] In `RStudio`, it is better to maximize the Project functionality to set working directories

  - After it is finalized, the base analytic file should never be overwritten.
  - This includes duplicate removal, dropping extraneous cases, variable creation and/or recoding, etc.
  - That is, the file constructed during the data preparation and management phase should always be retained. The inevitable changes to this file that occur during the analysis phase should either not be saved (that is, the analysis portion of your .do file must be re-run each time), or should be saved as a separate file.
  - This perhaps goes without saying, but the raw files should also not be overwritten. Raw files should not be editted, and if they must be, they should be saved with a new name noting the nature of any edits made to the original.[6]

- Analysis commands that produce results that ultimately go in a deliverable or published report should be annotated in the syntax file.
  - Ideally, this will be annotated, i.e., *Table 2 crosstab.

- For analysts who use `Stata`, we strongly recommend you use the Gibson `preamble` command. The front–matter defines global path macros that will facilitate replication and quality assurance inspections.

- In general, syntax or .do files and raw source files should be stored on a network drive (e.g., the `S:/`drive) to ensure they are recoverable in case of some type of failure. If you need to have them accessible locally, please ensure they are backed-up to a network drive as soon as you are able.

4 Naming folders and files

When possible, (particularly for data and code stored on the `S:/shared/data/` drive on Azure) please name folders and files in a systematic way that is free of non-alphanumeric characters (save for underscore delimiters). Please do not include spaces, parentheses, or other symbols in folder or filenames as these can cause programming scripts to choke. Typical convention is to use snake_case rather than CamelCase, though this has varied by researcher (and, consequently, project). Once a project folder name has been set, it should almost never be changed, since doing so will break all code referring to it.

4.1 Folder Naming and Structure

The project folder should include some distinctive moniker including the project client or title (like AISD_student_survey) without project or school year included in the title. Project directories should be clearly and intuitively structured and labeled. Typically,

[6] This should only be done in rare cases where, say, a file includes a header row above variable names, or includes characters which prevent the accurate import of data. Even still, scripted management of these issues is almost always preferred. Data values in raw files should never be edited.

syntax files will be in a folder separate from data files, and raw source files will be in a separate sub-folder from final analytic files.[7]

4.1.1 Common sub-folders

We generally use the same sub-folder or sub-directory structure in each project folder. We separate information into folders like raw, converted, processed, and final data as well as syntax, output, results (e.g. from models), temp (for tempfiles), archive (for archived or deprecated code), and other associated files like latex (for LaTeX files) or `HTML` If futher organization of files is useful, nest those distinctions within these base folders. For example, if a large project has district administrative data, observation data, afterschool data, and survey data create those folders within raw, converted, processed, and so forth.

- raw: Unaltered files as received from the client or other source.
- converted: Raw files converted to format used by analyst's language.
- processed: Converted files which are partially processed (variable and value labels for example), but need more work before they are final/analytic (need to be merged or combined across years).
- final: Analytic dataset which is used for descriptive tables, analysis, figures and other results.
- output: Tables and other files which will not be used in reporting, such as tables describing data issues for the client. Files output at one stage to be imported at another(which are not data files). Examples: a list of students missing test data, an excel sheet you export to organize variable renames and re-import.
- output/logs/: Store log files from `Stata`/`R` sessions. Logs should always be saved with the date in the name so that the point where errors may have been introduced can be detected.
- results: Tables, figures, and other output that will or could be used in reporting.
- syntax: Script files.
- latex/html/temp: Use as needed to store LaTeX and HTML files or other atypical files needed or produced.

The top of your script file should contain macros for each folder to make it easy to access, or use the `preamble` command which does so.

```
foreach fol in raw converted processed output final {
        global `fol' = ${sf}${fyear}`fol'
}
```

[7] Keep in mind that there limits to folder path lengths (260 characters on our MS Windows server). This is affected by both file name length and the number of sub-folders the file is nested in; so, please keep file nesting to a minimum where possible to avoid hitting this limit.

4.1.2 Working across Project Years

In most cases, projects take place across multiple years.[8]. The folder structure silos the data by year in most cases. Commonly, we have the sub-folders shown in the section above are nested within project year folders (named "year_1", "year_2" …"year_N") This notation also makes it easy to access to create macros which can refer to the prior year by parsing the original year_N macro. This allows for easy inclusion of data elements you want to add to the current year's data.[9]

```
global fyear `"year_7"'
global lyfyear `"year_`=substr(`"${fyear}"',-1,1)'-1"'

foreach y in `"ly"' `""' {
        global `y'sf `"${stem}project_folder/${`y'fyear}/"'
        global `y'syntax `"${`y'sf}//syntax/"'
        global `y'converted `"${`y'sf}/converted/"'
        global `y'processed `"${`y'sf}/processed/"'
        global `y'final `"${`y'sf}/final/"'
        global `y'results `"${`y'sf}/results/"'
        global `y'output `"${`y'sf}/output/"'
        global `y'raw `"${`y'sf}/raw/"'
}
```

The code example above will product raw and lyraw globals which refer to the folder in the relevant years.

4.2 File Naming

In general, filenames should:

- NOT contain non-alphanumeric characters (only contain `[A-Za-z0-9]` and underscores)
- contain underscores rather than spaces
- avoid starting folder and filenames with numbers (e.g., 2017_projectname_school)
- use consistent abbreviations
- use camel case (e.g., FileNameOne) to compress file name length where underscores are not needed or all lowercase. Do not use arbitrary case (e.g., staffSTUDENT_YR2012, myFileNAME_for2013_vER2.txt)

[8] Even if a project is expected to only last one year, create a year_1 as projects extending is not uncommon.

[9] `preamble` can also do this for you.

- be constructed to encourage looping over file names (that is, consistently named with sequenced elements). Example: surveyresponses_school1_2014_grade8.dta. Underscores can help with looping over sub-elements of a filename!
- If you must use versioning, be consistent in version labeling and meaning. File versions 1 and 2 could be suffixed with "V1" and "V2" but if you make a dataset for `Stata` version 12 or 14 make this distinct (e.g,. school1_2014_survey_stata_ver14.dta).

4.2.1 Syntax file naming

These rules apply for naming all file types in our Data folder hierarchy; however, for syntax (e.g., do-files) files we prefer that the file names are prefixed with a numeric 'order' so they can be run in the proper order across the folder.
For example:

10_clean_raw.do

11_convert_raw.do

12_process_clean.do

21_analysis_descriptivesRQ1.do

22_analysis_descriptivesRQ2.do

23_analysis_modelsRQ1.do

30_analysis_appendix.do

For surplus syntax files that don't neatly fit into the analysis process/sequence, use a distinct (or large) number to set it apart (like 99). Examples include files containing meta code (like labeling), ado-files used to process procedures across multiple syntax files in this subdirectory , or scratchpad files of code you don't want to erase/lose.
Continuing the previous example:

10_clean_raw.do

11_convert_raw.do

12_process_clean.do

21_analysis_descriptivesRQ1.do

22_analysis_descriptivesRQ2.do

23_analysis_modelsRQ1.do

30_analysis_appendix.do

99_varlabels.do

99_value_labels.do

99_scratchpad_assigngroups.do

99_adofile_maketables.do

5 Naming variables, macros, and other characteristics in Stata datasets

Variables (columns), macros, and characteristics names should follow the constraints and conditions from Stata and R. Even in Excel and other program (where these rules don't apply), if we can enter variable names with the same rules that apply in Stata and R, then transferring this data cleanly will help reduce data cleaning effort.

Rules which meet Stata and R standards: variables (and macros) can contain up to 32 characters. A variable name may contain only the digits 0 to 9, upper or lower case English alpha characters (A to Z), and underscores; and the first character cannot be a number or an underscore.[10] Except where necessary, such as folder paths, years, and macros that need to persist when other script files are called from a main script, locals should be used rather than globals.

We advise creating loopable, consistent variable names that also convey meaning when possible (though you have -variable labels- and characteristics (-char-) to store that information. If the variable attributes are too long to easily store in a variable name then use short abbreviations (that hopefully capture some attribute with loopable structure) and rely on the labels and codebook to decipher meaning.

In terms of having loopable, consistent naming, follow similar patterns as for file naming where the entity is followed by the time or level component and any adjunct details in the suffix. Since space is at a premium, abbreviate any entities as necessary.[11]
Here are some good examples:

student_grade8_2004

student_grade8_2008

student_grade8_2012

staff_grade8_2004

[10] Stata variables can be started with an underscore without issue, locals in Stata can, but should not start with an underscore, globals cannot as Stata uses the underscore internally to differentiate between locals and globals.

[11] For example: "grade" can be "gr_", student can be "stu_", but remember to label your variables!

staff_grade8_2008

staff_grade8_2012

student_grade12_2004

student_grade12_2006

student_grade12_2008

student_grade12_2012

staff_grade12_2004

staff_grade12_2008

staff_grade12_2012

This type of variable naming makes it easy to loop over type (student, staff),[12] grade level (8, 10, 12), and school year (2004, 2006, 2008, and 2012). Since these variable names are shorter than 32 characters, it leaves plenty of room to add suffixes to these names when generating new versions of these variables (e.g., staff_grade12_2012_meanscore, staff_grade12_2012_v2).[13]

Macro names should be descriptive and facilitate easy recall of what they are within loops. Descriptive macro names also reduce the chance an analyst will accidentally use the same macro name twice, which can be a difficult bug to figure out.[14] While Stata does not require indentation in loops or if blocks, each loop and if block should be indented to facilitate reading. For long loops or if blocks, the start and particularly the end, should be noted. For example, this may work:

```
local list race course gender grade
levelsof teacher, local(aa)
foreach a in `aa' {
levelsof student, local(bb)
foreach b in `bb' {
foreach c in `list' {
levelsof `c' if teacher == `a' & student == `b'
foreach d in `r(levels)' {
count if teacher == `a' student == `b' & `c' ==`d'
}
```

[12] We typically indicate the level of analysis (e.g., stu_ for students or tch_ for teachers) in the variable name prefix

[13] Beyond looping convenience, (like the folder naming convention above) these variable names are extensible. As new data is added in future years of the project, we can add a staff_grade6_2018 survey with no problem to the existing programming if our code takes advantage of all levels of values contained in the data (e.g., the stata code uses something like `levelsof year, loc(yy)` rather than explicitly listing each year in the dataset).

[14] Or worse, not be detected and produce the wrong result!

```
}
}
}
```

But this will work and be easier to read, especially if the manipulations or calculations or are more extensive, or if there are if blocks within it. Particularly with complex nested loops, labeling or numbering each loop eases de-bugging and future modifications.

```
local varlist race course gender grade
levelsof teacher, local(teachers)
foreach t in `teachers' { //Teacher
  levelsof student, local(students)
  foreach s in `students' { //Students
    foreach v in `varlist' { //Variable
      levelsof `v' if teacher == `t' & student == `s', local(varlevs)
      foreach vl in `varlevs' { //Levels
        count if teacher == `t' student == `s' & `v' ==`vl'
      } //End level of variable
    } //End of variable
  } //End of student
} //End of teacher
```

Procedures and lists should be unified. This is to say, if a similar non-standard procedure is going to be executed multiple times, it should be done in a loop, or using a program/function written at the top of the syntax file.[15] Lists of variables which will be used repeatedly should be defined once. Parameters that may be adjusted within a loop that are used several times should be defined as locals, for example, colors, text sizes, and spacing adjustments in a loop producing a figure composed of multiple -twoway- commands.

6 Gibson's computing environment and tools

6.1 Azure Virtual Machine

The Azure remote desktop environment is where all Personally Identifiable Information(PII) is stored and all quantitative/programming work is conducted. Researchers access it via a remote desktop connection using multi-factor authentication. While the purpose of the server is to maintain a secure environment for PII data, data files which do not contain PII are also stored here for convenience (for example, school level files and other publically available data). Never store PII data on your location machine! Other file types, such as output, draft results files, LaTeX or Markdown reports are also stored here before they are ready for consumption, either by non-quantitative staff or clients.

[15] Standard repetitive procedures should be turned into an .ado file (in `Stata`, or package library in `R`)

Quantitative data are stored on a storage device which is connected to the Azure environment. Files stored here can be accessed by mapping the device to a letter drive. To map this device follow these steps.

1. Log into the Gibson Azure VM.
2. Open file explorer.
3. Go to this PC.
4. Click on the ... at the top right.
5. Select map network drive.
6. Select Drive S.
7. enter: \\gibsondata.file.core.windows.net\data
8. Make sure reconnect at sign in is selected.
9. Make sure connect using different credentials is not selected.
10. Click Finish.

Within that device is the shared folder, where all data are stored, and Users. Each Gibson employee has a folder where they can store private files, or files which are not relevant to projects or the broader team, such as experimental syntax for .ados or libraries. While other Gibson staff cannot access these folders, the information stored is recoverable by those with administrative privileges.

Gibson is charged for use of the Azure server whenever it is "on" regardless of whether computing resources are being used.[16] The Azure server is set to turn off at 8pm Central Time each night and can be turned on in the Azure portal (https://portal.azure.com/#home). Researchers are encouraged to work at whatever times they would like, and should not hesitate to activate the sever if they would like to work and it is off, but should turn it off if it is not in use by anyone. Some projects and data sharing agreements will also require Gibson to limit access to data, and consequently, access to those folders will be restricted to those who are active on the project. If you need to be added to a folder, or be given the ability to turn the virtual machine on or off, Danial and Ali are able to do so. Most staff prefer to use the Azure Remote Desktop on one of their two screens, reserving the other for email, slack, and other items. To open it in only one screen right click the "Gibson Azure VM" icon in the remote desktop application, and turn off "Use default settings." That option will then appear.

[16] A portion of the costs are fixed, but running the environment 24 hours a day, rather than the 8–10 hours during of the day that people are typically working would increase the cost to operate it.

6.2 Egnyte

Egnyte is used to securely transfer PII and other files to and from clients and other project stateholders. Files containing PII should be removed from Egnyte once they are stored in the Azure environment. Egnyte can also be used to share links which allow clients or other stakeholders to download files and upload files securely. Note that the permission structures of the folder may prevent sharing. Egnyte will allow links to be generated which can't be used by those outside of Gibson so confirm they are set correctly before distributing links. Egnyte can also be mapped to your local machine[17] and the Azure environment. Egnyte should be mapped to the Y: drive in order to maintain code transferability. To map Egnyte:

1. Download, install, and open the Egnyte Desktop App.
2. Sign in.
3. Click on settings.
4. Click add drive on the manage drives tab.
5. Map the drive to Y:

6.3 Sharepoint

Sharepoint is where client, R&E management, and business development files are stored. Results files and reports should be moved here when ready for consumption. Sharepoint also requires multi-factor authentication. Sub-pages in SharePoint (Research and Evaluation, Business Development, Operations, and Consulting) are listed at the top. Click one to enter that sub-site. Within the Research and Evaluation page, the projects tab on the left side is the high-level storage container for project work. The management tab includes mangement and projection files. Within each project folder are relatively standardized subject folders.

- contracts: Client contracts.
- data: Data request documents, and may also contain copies of data sent through uncommon channels, such as paper survey data emailed to a project director. Data files stored here will generally be copied (if not PII) or moved (if PII) to Azure.
- id: Instrument development files, which are survey instruments, observation, and interview protocols and other files related to the development of those.
- pm: Project management files including meeting agendas OneNote files, progress reports, and budget tracking information.
- reporting: Reporting files and related files, generally deliverables to clients.
- results: Tables, figures, and other output that will or could be used in reporting transferred for non-Azure users' access.

[17] But don't store PII on your local machine!

Folder structures on SharePoint are more variable than in Azure, but proper organization is still important. Note that project years are nested inside topics on SharePoint, where the opposite is true on Azure. This is because it is typically more useful to toggle between years within a topic area for these tasks, whereas quantitative workflows are better suited to topic areas nested within year. SharePoint can also be linked to your local machine and the Azure environment, which is useful exporting results files from `Stata` or `R` once ready. To map SharePoint to a computer (local or Azure):

1. Open SharePoint and go to the main Projects folder under Research and Evaluation.
2. Click the Sync button.
3. This will open up a dialogue about setting up OneDrive, sign in and work through this process.
4. When it asks for the folder to use locally, please enter: C:\data\sp\
    - create that folder if it does not exist
5. This should create a linkage whose path will be: C:\data\sp\GIBSON\Research and Evaluation - Documents\...project_folder

Microsoft file types opened from SharePoint can be editted by multiple people at the same time and are typically automatically saved instantly,[18] and so SharePoint documents are ideal when collaboration on a report or proposal is needed. SharePoint is still being implemented, and so some conventions may change.

6.4 Qualtrics

Gibson uses Qualtrics for survey projects (and survey elements of other projects). Data analysts should always use `qualtrics` (`Stata`) or the `qualtRics` library (`R`), since these programs increase efficiency and reduce the potential for error in survey cleaning and preparation. The API token for Qualtrics is available in the "Qualtrics IDs" section of the account settings. Do not share the Qualtrics API token! A bad actor could use this to delete all of our survey data.

6.5 Simpletexting

Simpletexting is a service that Gibson uses to send survey inviations or other information en masse. This service is only active when it is being used for a project. Messages should be crafted carefully to decrease the probability of being marked spam, such as avoiding words typically associated with spam messages. URLs used in text messages should https:// (rather than http://) and should use Simpletextings link shortener tool. When using macros

[18] Provided all users have current versions of Office.

in simpletexting, note that the length must be set if it is greater than the default length, and if a distribution is opened again, those macro lengths must be re-set or they will return to their defaults.

6.6 Pythonanywhere

Pythonanywhere is the service that Gibson uses to house `python`-based dashboards (using the Dash library). Sample syntax is available for uploading updated data and resetting the app(s).

6.7 Shinyapps

Shinyapps is the service that Gibson uses to house Rshiny-based dashboards.

6.8 gibsonsurveys.com

Gibsonsurveys.com (along with various project-specific urls) is where web portals and dashboards[19] are housed for survey projects. Web portals or other materials should be placed in different folders for each project. For example, www.gibsonsurveys.com/pis/... for the Texas parent involvement survey and www.gibsonsurveys.com/psos/... for the Texas post-school outcomes survey. Sample syntax for uploading/updating web pages programattically using WinSCP is available.

6.9 Autohotkey

Autohotkey is a program that allows users to create their own (and more extensive) hotkeys. While a wide variety of operations can be programmed using the tool, hotkeys which add dated notes (*!DH Note 1/20/2022: A note) and section delimiters are strongly encouraged to improve the readability of code. Researchers should feel free to customize their own scripts for the programming languages and hotkeys/repetitively-typed-snippets that they use. Assistance in setting up and customizing Autohotkey scripts is available.

7 Gibson-researcher written `Stata` programs

This section lists and provides a brief description of the programs contributed by current and former Gibson researchers. The programs were developed to simplify reoccurring actions that we encounter in our projects. We are slowly migrating these to our Gibson GitHub account so that we can centralize their storage, and to facilitate collaboration. These are monolingual, and all developed for `Stata`: we encourage polyglots to port these to their preferred programming language.

[19] Using an iframe to pythonanywhere or shinyapps

7.1 preamble.ado

7.1.1 Author: Danial Hoepfner

7.1.2 Description

`preamble.ado` incorporates the typical information that's included in a premable into a single command. It has several options that simplifies this routine step.[20] It also produces date and time globals and globals for Gibson colors, which can optionally be supressed.

7.1.3 Usage and options

1. `folder`: This is for the root directly and the shared drive alias. It will incorporate the Azure storage device path and include globals for all of the typical data folders(raw converted syntax processed output results final). This is required.
2. `fyear`: This indicates the folder year for projects with multiple years (in the format of year_N).
3. `sharepoint`: Specifies a global with the path to the SharePoint folder for the project.
4. `egnyte`: Specifies a global with the path to the Egnyte folder for the project.
5. `addfolders`: Adds non-typical folders to the project-year, such as latex or html.
6. `lastyear`: Adds globals for the prior project year (year_N-1), must be used with the `fyear` option.
7. `stem`: replace the Azure file path stem with another (useful for working on local machines).
8. `nomk`: Do not make any folders that do not already exist.
9. `noreport`:Suppress output noting the globals defined and other actions taken by the command.
10. `year`: Specify a year for globals, the year is detected from the system if not in use.
11. `noyear`: Supress the creation of year and date globals.
12. `log`: Name of the log file for the session, to be stored in output/logs/. Not required, but very strongly reccomended for most syntax files.
13. `discard`: `discard` command to drop programs in memory (and re-read from .ado path).
14. `reset`: `frames reset` command to clear all data frames in memory.

[20] The original version was written by Marshall Garland, but was re-written by Danial Hoepfner to adapt to Azure and SharePoint evironments and extend usability.

15. nocolors: Supress the creation of Gibson color globals.

```
preamble, f(region_10_sped) fy(year_7) share(R10_Ind_8_Survey) ///
        e('"Client Work\Region 9 Parent Survey Materials\"') ///
        add(latex) last nomk
```

7.2 varcount.ado

7.2.1 Author: Marshall Garland

7.2.2 Description

varcount.ado is a command that counts the number of variables in a variable list or, if a variable list isn't provided, the number of variales in the dataset.

7.2.3 Usage and options

See the GitHub readme for instructions.

7.3 graphsout.ado

7.3.1 Author: Marshall Garland

7.3.2 Description

graphsout.ado is a wrapper for graph export that allows the user to export a graph from Stata to multiple file formats.

7.3.3 Usage and options

1. The command requires a file name.
2. replace: Overwrite the file name on disk, if it exists.
3. type: File format from the list of allowable file formats for graph export. Type help graph export to see the list of available formats.
4. font: Graph text font.

```
graphsout /Volumes/Desktop/, ///
  replace type(emf pdf svg) font("Calibri")
```

7.4 qualtrics.ado

7.4.1 Author: Danial Hoepfner

7.4.2 Description

`qualtrics.ado` interacts with the qualtrics API to list, download, and clean survey data collected on Qualtrics.

7.4.3 Usage and options

The command has three sub-commands, `qualtrics set`, `qualtrics list`, and `qualtrics get`. Unlike most of our .ados it has a help file.

1. `qualtrics set`: Set a password and optional user name to use instead of the token and data center each call.
    - `token`: Qualtrics token for our organization.This is sensitive! Someone could delete all of our survey data with it, required.
    - `center`: Qualtrics data center for our organization, "co1" is our data center, required.
    - `password`: a password you specify, required.
    - `user`: a user name you specify, only needed if using multiple Qualtrics accounts or sharing an .ado folder with another.
2. `qualtrics list`: List surveys associated with your account, filter results.
    - `match`: Only list surveys whose name matches the included regular expression.
    - `active`: Only list surveys that are active.
    - `modrange(MDY:MDY)`: Only list surveys that were modified in a date range.
    - `createrange(MDY:MDY)`: Only list surveys that were created in a date range.
    - `user`: The username, optional.
    - `password`: The password, required unless specifying token and data center.
    - `token`: Qualtrics token for our organization, not needed if using password/user.
    - `center`: Qualtrics data center for our organization, not needed if using password/user.
3. `qualtrics get`: Download, convert, and clean a survey.
    - `id`: Only list surveys whose name matches the included regular expression.
    - `csv`: Folder to save the .csv file (name is maintained from Qualtricsm required.
    - `dta`: Folder and file name to save the .dta file, required if using clean option.
    - `valuelabs`: Request value labels rather than numeric codes, incompatible with clean option.

- clean: Clean dataset, apply labels, value labels and characteristics from survey meta data.
- relab: Display code to variable adjust labels.
- revallab: Display code to value adjust labels.
- preserve: Restore current dataset, rather than loading downloaded file.
- user: The username, optional.
- password: The password, required unless specifying token and data center.
- token: Qualtrics token for our organization, not needed if using password/user
- center: Qualtrics data center for our organization, not needed if using password/user

```
qualtrics set, t(`"QualtricsAPITokenHere"') c(`"az1"') ///
        password(ApassW0rd) u(danial)

qualtrics list, password(ApassW0rd) m(Fall 20[1-2][890])

qualtrics get, id(`"SV_XYXYXYXYX"') password(ApassW0rd) ///
     csv(`"C:/Users/dhoepfner/Desktop/"') ///
     dta(`"C:\Users\dhoepfner\Desktop\testdata.dta"') clean relab reval
```

7.5 gtab.ado

7.5.1 Author: Danial Hoepfner

7.5.2 Description

`gtab.ado` is a tool to build custom tables easily. It works by creating a set of string vectors which you then fill with subsequent gtab commands. You can preview, import into `Stata` or export to excel

7.5.3 Usage and options

The command has six sub-commands, `gtab init`, `gtab [column_name]`, and `gtab fill`, `gtab preview`, `gtab import`, and `gtab export`. You can also build multiple tables simultaneously by inserting a number between gtab and the subcommand.

1. `gtab [number] initialize`: Initialize a table (or tables using a number) with the desired columns names and order. gtab will then print commands to add to each column.
2. `gtab [number] [column_name]`: After initializing the table, add row(s) to that column. Information can be added to columns in any order, though cells meant to be empty for a particular row must be filled (either by having a blank gtab [column_name] command or using -`gtab fill`-.

3. `gtab [number] fill`: Since gtab works by using a series of string vectors, in empty cells aren't specified, the table can become mis-aligned. gtab fill equalizes the vector lengths so that all columns have the same number of rows, by adding blank rows to columns shorter than the longest column.

4. `gtab [number] preview`: Preview table in the results window.

5. `gtab [number] import`: Import table into current data frame.

6. `gtab [number] export ``file_path_and_name''`: Export table to an excel file.

   - `sheet`: Workbook sheet to export to.
   - `replace`: Replace excel file if it exists.
   - `sheetreplace`: Replace sheet if it exists.
   - `sheetmodify`:Modify sheet if it exists.

Basic usage with a number, allowing multiple tables simultaneously

```
sysuse auto, clear
gtab 1 init var mean min max

gtab 1 var Variable
gtab 1 mean Mean
gtab 1 min Min
gtab 1 max Max

ds *
foreach v in `r(varlist)' {
    if !regexm(`"``:type `v''"',`"str"') {
       gtab var ``:variable label `v''
       sum `v'
       gtab 1 mean ``:di %3.1f `r(mean)''
       gtab 1 min ``:di %3.0f `r(min)''
       gtab 1 max ``:di %3.0f `r(max)''
   }
}
gtab 1 mean Legend: Data from auto.dta
gtab 1 fill
gtab 1 pre
gtab 1 export "C/users/dhoepfner/results/example.xlsx", ///
replace sheet("descriptives")
```

Advanced Usage

```
sysuse auto, clear
/*Here I am going to use the levels of foreign to help define the columns,
```

```
this will make tables extensible to additional levels added to a variable.
I'll define and add to a local called gtab that defines the columns.
When filling the table, you can use of the same loops for the table.*/

local gtab var
levelsof foreign
foreach l in `r(levels)' {
    local gtab `gtab' m`l'
}
*Now adding a column for the difference
local gtab `gtab' d
*Now adding columns for the Ns
levelsof foreign
foreach l in `r(levels)' {
    local gtab `gtab' m`l'
}
*Now initializing gtab
gtab init `gtab'
*Now filling the title row
gtab var Variable
levelsof foreign
foreach l in `r(levels)' {
    gtab m`l' `:label (foreign) `l'' Mean
    gtab n`l' `:label (foreign) `l'' N
}
gtab  d Difference

/*Now we'll loop over each variable and level of foreign
and extract and export the desired statistics.
Notice that we can use stored results to define stars for significance,
or whatever else we want.*/

local vct=0
ds foreign, not
foreach v in `r(varlist)' {
  if !regexm(`"`:type `v''"',`"str"') {
      local ++vct
      gtab  var `:variable label `v''
      levelsof foreign
      foreach l in `r(levels)' {
        sum `v' if foreign == `l'
        local m`l' `r(mean)'
        local n`l' `r(N)'
```

```
        gtab m`l' `:di %7.1fc `m`l'''
        gtab n`l' `:di %6.0fc `n`l'''
      }
      local stars
      ranksum `v', by(foreign)
      if `r(p_exact)' < .05 local stars `"*"'
      if `r(p_exact)' < .01 local stars `"**"'
      if `r(p_exact)' < .001 local stars `"***"'
        if `vct' !=5  gtab  d `:di %3.2f  `=`m1'-`m0''`stars'
        if `vct' ==5  gtab  d `:di %3.2f  `=`m1'-`m0''Custom Tables!
   }
}
*Import the table into the current dataset/frame.
gtab import
```

7.6 fmtexl.ado

7.6.1 Author: Danial Hoepfner

7.6.2 Description

`fmtexl.ado` is a wrapper for putexcel which formats excel tables nicely. By default it loops over each sheet in an excel file. Notes: You need to use -`fmtexl using`- syntax to specify excel file. Default settings are not optimal unless formatting tables like JS likes, the bw option is closer to journal style tables.

7.6.3 Usage and options

1. `headfill`: Header fill color, default is Gibson blue
2. `headtext`: Header text color, default is white
3. `lfill`: Light row fill color, default is white
4. `dfill`: Dark row fill color, default is light blue
5. `rowcol`: Table body rows alternate colors
6. `doublerowcol`: Table body rows alternate colors in blocks of two, for example, when including standard error in row below estimate.
7. `headrows`: More than 1 header row, specify number
8. `labcols`: More than 1 label column, specify number
9. `specrange`:Specify table cell range rather then detect from import excel, allows formatting of multiple tables on the same sheet. Formatted as [A-Z]*[1-9]*:[A-Z]*[1-9]*

10. bw: Black and white journal style format. Overwrites all formatting options except for labcols and headrows

```
fmtexl using "C/users/dhoepfner/results/example.xlsx", ///
bw labc(2) headr(2)
```

7.7 apchx.ado

7.7.1 Author: Danial Hoepfner

7.7.2 Description

apchx.ado loops over all variables in current dataset and looks for type differences in variables in a specified dataset. If you attempt to append a dataset in Stata and a variable is byte in one dataset, but string in another, you will get an error. Without this, Stata errors after the first mismatch and so this issue can repeatedly bite. It also has options to compare value ranges and labels in the two datasets.

7.7.3 Usage and options

1. range: Check variable ranges are consistent across datasets.
2. label: Check value labels are consistent across datasets.

```
apchx using "C/users/dhoepfner/cleaned/example.dta", label range
```

7.8 conwithin.ado

7.8.1 Author: Danial Hoepfner

7.8.2 Description

conwithin.ado checks whether variable list A is constant within variable list B. Very useful for panel or nested data.

7.8.3 Usage and options

1. within: Variable list B, typically ID variables in which variable list A should be constant.
2. tag: Tags observations where a variable in list A is not constant within list B.

```
conwithin race gender disability, wi(student_number) t(t)
sort student_number
list student_number race if race_t == 1
```

7.9 edrop.ado

7.9.1 Author: Eric Booth

7.9.2 Description

`edrop.ado`: -cap drop- won't drop all variables in a list if one does not exist. -edrop- drops all variables in the list, and reports those that were not present in the first place.

7.9.3 Usage and options

1. `keep`: Keep that list instead of dropping
2. `temp`: Also drop temporary variables

```
edrop race1 race2 race3
```

7.10 latest.ado

7.10.1 Author: Danial Hoepfner

7.10.2 Description

`latest.ado`: Best practice is to save analytic datasets with a date appended, so the point errors were introduced can be tracked down. However, this means you need to specify the date of the most recent file for each -`use`- command. This returns 'r(file)' which is the most recently saved file, optionally matching a regular expression.

7.10.3 Usage and options

1. `match`: Return most recent file matching regular expression.

```
latest `"${converted}"', m(`"student.+\.dta"')
use `"${converted}`r(file)'"', clear
```

7.11 char_finder.ado

7.11.1 Author: Danial Hoepfner

7.11.2 Description

`char_finder.ado`: -`charlist`- on SSC by Nick Cox works great to identify all of the unqiue characters in a string variable. However, sometimes order matters, and so char_finder returns the text of a string, with the ASCII codes displayed below. Works well for finding Microsoft Word style quotes and other symbols (which `Stata` reads as a series of ASCII codes, which can then be replaced with subinstr() and '=char(x)').

7.11.3 Usage and options

```
char_finder '";asdklfjsd ;lfjd()*)*)(@"'
```

7.12 fixtex.ado

7.12.1 Author: Danial Hoepfner

7.12.2 Description

`fixtex.ado` converts symbols in text that will cause a LaTeX script to choke into their proper codes.

7.12.3 Usage and options

```
fixtex A sentence_with $ some probl&em symbols
tex `r(output)'
```

7.13 texpdf.ado

7.13.1 Author: Danial Hoepfner

7.13.2 Description

`fixtex.ado` compiles a .tex document.

7.13.3 Usage and options

1. `copy`: Copy resulting PDF to a new location.
2. `open`: Open PDF once compiled.

```
texpdf `"${tex}region_10_report.tex"', copy(`"${to_distribute}"')
```

7.14 syntax_saver.ado

7.14.1 Author: Danial Hoepfner

7.14.2 Description

`syntax_saver.ado` attempts to recover syntax when `Stata` crashes. Copies temporary syntax files to a restore folder, or another specified folder. Typically, `Stata` only stores the most recent run in an instance, so if you only highlighted one line, that's likely all you'll get.

7.14.3 Usage and options

1. restore: Specify a folder to save restored files to. Default is C:/Users/‘c(username)’/Desktop/restore/
2. Today: Only restore files from today.

```
syntax_saver , today
```

7.15 latout.ado

7.15.1 Author: Danial Hoepfner

7.15.2 Description

latout.ado Loops over a list of variables, or all variables in a dataset, and produces a PDF to check for issues. Can select a -by- variable to present figures/tables by another variable. Particularly useful for multi-year data submissions.

7.15.3 Usage and options

1. title: Title of PDF
2. path: Where to save PDF, default is working directory
3. footer: Text to place in footer which links back to TOC, default is ”Return to Top”
4. byvars: Variables to split output by, max =2
5. holes: Where to have a hole in panelled figures, useful when byvars levels is an odd number
6. columns: Number of columns for by variable panelled figures
7. cutoff: Cutoff between number of categories before variables are treated as continuous, default is 5
8. Strings: How to display strings with more than CUT categories, options are -list-, -sample-, and -skip-, default is sample
9. varnames: Include variable names (not just labels) in output.

```
latout , title("AVID Submission 2017") ///
pa‘"(${output}"') bvy(institution year) varn
```

Authors

Marshall Garland,

Eric Booth,

Jill Carle,

Danial Hoepfner,
dhoepfner@gibsonconsult.com

David Osman,
dosman@gibsonconsult.com

Mitchell Kilborn,

Gracie Petty,
gpetty@gibsonconsult.com

For more information, please visit:

https://cesr.usc.edu/care