



Gutmacher Institute

Coding Style Guide

INTRODUCTION

This guide establishes conventions for writing analysis code at Gutmacher. It is geared towards analyses written in Stata, but many of the general principles laid out here are broadly applicable to coding in other statistical programs as well. This guide is also not meant as an introduction to how to use Stata, although there are examples of code throughout, and occasional tips for how to take full advantage of Stata's capabilities; in addition, for some more complicated concepts, there are links to either relevant Stata help files or other online resources.

The styles in which people code can be very personal; the way we program is shaped by who trained us, what software we first used, and our own working style. Programming can be a deeply creative activity, and just as with writing, there is no one-size-fits-all approach. In order for work to be reproducible and transparent, however, it is important to set out basic common standards for what is expected for code written at the Institute. The standards are designed to make our programs more robust, easier to read, and (hopefully) easier to write as well.

Gutmacher is not the first organization to put together a style guide for Stata, and this guide draws on many of the existing best practices for coding in Stata and other languages established by the field. In particular, many sections of this guide are adapted from the ADOPT: Coding Style Guide developed by the Strategic Data Project at Harvard University, as well as a report produced by Innovations in Poverty Action (IPA), Reproducible Research: Best Practices for Data and Code Management.

GENERAL PRINCIPLES: The recommendations in this guide are shaped by three fundamental principles, each of which are integral to ensuring that work at the institute is reproducible by researchers both inside and outside of Gutmacher:

1. **All steps in an analysis project, including data cleaning and management, should be reproducible.** An analysis is reproducible if someone, given your raw data and code, can recreate all of your results easily and without additional modifications. A lot of tiny decisions can get made over the course of an analysis, especially at the data cleaning stage; to the extent possible, all of these tiny decisions should be implemented in program code and well documented.
2. **Every program should be commented well enough that a research assistant with basic programming experience could follow each step.** In order for code to be reproducible, someone needs to be able to look at it and understand what you have done – even if they aren't an expert in the specific programming language you are using.
3. **Every number and figure in the text and tables of your final product should be able to be easily traced back to the code that produced it.** This includes sensitivity analyses and numbers just cited in the text. Ideally, all of the tables in a published paper are generated from the code itself.

HOW TO USE THIS GUIDE:

There are a few different ways to use this guide:

1. **Give it to a new staff member**, so they can learn coding practices and norms at Gutmacher. Different institutions have different standards, and newer staff members may have less experience working collaboratively. Distributing this guide at the beginning of a project with new staff can help establish common norms for how code should be formatted and written.
2. **Give it to a research assistant**, so that they can go through code you've written and modify the formatting to be clearer and more beautiful, following the best practices laid out here. This is particularly relevant for the techniques and standards laid out in Section 3 (Commenting and Readability), which do not require advanced knowledge of Stata.
3. **Give it to yourself**, as a reference manual for best practices. Ideally, cleaning up code shouldn't happen at the end of a project. Implementing the practices laid out here as you write your code will save you time and make it harder to introduce errors (and easier to catch them).

For many of the standards and tips laid out in this guide, there are accompanying code snippets demonstrating correct and incorrect techniques (incorrect techniques are highlighted in red). Feel free to copy and paste code from these snippets directly into your own do-files.

TABLE OF CONTENTS

[SECTION 1: DO-FILE STRUCTURE](#)

[SECTION 2: HARD CODING VS. MACROS](#)

[SECTION 3: COMMENTING AND READABILITY](#)

[SECTION 4: LOOPS AND NESTED STATEMENTS](#)

[SECTION 5: VARIABLE NAMES AND LABELS](#)

SECTION 1: DO-FILE STRUCTURE

Distinct aspects of an analysis should be broken up into smaller ‘chunked’ do-files, all of which are run in order by a larger “Master” do-file. Ideally, the Master do-file should be heavily commented, and should act as documentation of the broad steps of your analysis. One way to think about this is to treat the Master do-file as a draft of your eventual methodology section: reading it should be sufficient for a reader to be able to gain an overall understanding of the study methodology, and when drafting the methodology section of a paper, you should be able to use it as a reference to remember what decisions you made.

At its most basic, however, the Master do-file should document which programs should be run, and in what order, to be able to reproduce your entire analysis. It also serves as an easy way to run, or re-run, your full program, from recoding the raw data to producing the final tables.

A very simple Master do-file (without a file header, which is discussed later in this section), might look something like this:

```
////////////////////
/// SECTION 1: Data Management //////////////////////////////////
////////////////////

////////////////////
* 1. Append 2006-10 and 2011-15 NSFG
/*
This program appends the 2006-10 and 2011-15 cycles of
the NSFG (female respondent files). There are a few
light recodes to make the two cycles compatible (mainly
to do with complex survey design variables).

It also sets the complex sample design (using the
two-year weights), and saves out a full appended dataset
called "NSFG female 2006-15.dta"
*/
run "1. Append 2006-10 and 2011-15 NSFG.do"
////////////////////

////////////////////
* 2. Recodes of demographic and outcome measures
/*
This program recodes basic demographic variables which
will later be used in many of our models, as well as the
final outcome measures xx and xx.

The demographic recodes are standard; the exception is
relcat4, for which we treat missing values slightly
differently. This is because of blah blah blah. As a
result, instead of blah we blah; this allows us to
blah...

The main outcome measures used in this analysis are
blah and blah. These are calculated in a 10-step
process, outlined below (and in more
detail in the do-file itself):

1. [brilliant analysis decision]
2. [even more brilliant]
...
10. [the coup de grace]

variables created: hisprace, educmom3, povcat3, agecat5,
                  attnd14B, relcat4, metro2,
                  briloutcome1, brilloutcome2

It then saves out the recoded file as
"NSFG female 2006-15 recoded.dta"
*/
run "2. Recodes of demographic and outcome measures.do"
////////////////////
```

(continued on next page)

```

////////////////////////////////////
// SECTION 2: Analysis //
////////////////////////////////////

////////////////////////////////////
* 3. Descriptive tables
/*
This program runs descriptive crosstabulations of our
outcome variables with the demographic measures
described above. These crosstabs correspond to the
data appearing in Table 1.

It then formats the results, and exports
them to Excel.
*/
do "3. Run descriptive tables and export Table 1.do"
////////////////////////////////////

////////////////////////////////////
* 4. Complicated analysis
/*
This program uses the Alster-washerfish procedure
described in watts et al. 2007 to blah blah blah
while accounting for up-censoring and diagonal nesting.
It runs a few checks to make sure the model has
converged correctly, and then formats the results
(which correspond to Table 2 and Figure 2) for export.
It then exports them to Excel.
*/
do "4. Run complicated analysis and export Table 2 and Figure 2.do"
////////////////////////////////////

```

A few notes on the do-file above:

- Depending on the project or programmer, the do-files run by the Master file may be narrower or broader than the ones laid out here. For example, it's possible that you might want to split file #3 into two files — one coding the demographic variables and one coding the outcome measures — depending on how many variables there are, and your own preferences.
- It's also not necessary to save out a dataset after each (or any) file; as long as the files are run in sequence, each file can call on the recoded data produced by the file before it. That said, there is nothing wrong with doing so if you want to be able to call on interim data easily, or if certain files take a long time to run.
- Use [run](#) when you don't need to see the results of the do-file; use [do](#) when you want to see the results in the output window. Note that in the above example, we use *run* in the data management section and *do* in the analysis section.
- If do-files need to be run in a particular order, it's a good idea to include numbers (as we did in the example above) at the beginning of the file name. Among other things, this allows you to view them in the order they should be run by sorting by file name in their home folder (which can be useful as a quick reference). The Master do-file itself should have a file name that starts with 0, so that it appears at the top of the list in your program folder.
- Do-file names should strike a balance between brevity and clarity, and should contain verbs.

File Header for Master Do-file

Every do-file in your analysis should have a *file header*, to orient the reader with a few key pieces of information as well as set up the system environment. The file header should have the program name, the project code, a short description of what the program does (along with any important notes), the date it was created, and the person or people who created it. A typical file header might look like this:

<pre>* Program name: Example of Master Do-File * Project(s): 488 * Program task: /* This file runs the do-files for the blah blah analysis (using NSFG data) in order. This allows us to keep track of the correct order the files should be run in and easily update the analysis when there are small changes or additional data. It can also be used as a reference for the methodology used to calculate these numbers, as it is commented thoroughly throughout (though more detailed comments are in the individual do-files it calls). The main research question for this project was blah blah. NOTE : This is just an example file, created for the coding style guide, and will not run. */ * Date created: 04/10/2017 * Written by: Isaac MZ * System set-up clear all version 14.1 capture log close set more off freely * Set linesize. set linesize 90 * Increase maximum number of variables to 10,000. set maxvar 10000</pre>	<pre>// clear everything in memory // set Stata version // close any open log files // allow program to scroll</pre>
---	--

The Master Do-File header should follow this general format as well, but in addition to the general information listed there, it also should include several other key pieces of information to orient the reader, as well as set up the system environment.

- The research question you are investigating. Remember that the Master Do-file is meant to orient other researchers to the purpose and structure of your code and analysis; in future years, you might not even remember what question(s) the code was designed to answer (this is particularly important if this is a side project, which may not have a formal project code, or something that never resulted in a formal paper).
- Set the version of Stata that you are using. New versions of Stata may contain updates that alter your results. For example, the [seed](#) for random sampling changed between Stata versions 13 and 14. Any do-file that used the older version of Stata to sample would no longer be reproducible.
- A statement installing any user-written programs you use in your analysis (and describing what they do). One of the wonderful things about Stata is that you can install user-written packages that can extend or modify what it can do (or write your own!). In general, these should be used with care, as they are more likely to have unexpected bugs, and because they are a “black box,” they can make it harder for someone to understand what you did. If you do use them, however, make sure to include a statement at the top of your file that explains what they do, checks if they are installed, and if they are not, installs them automatically:

```

** Check if -estpost- is installed -- if not, install it.
/* -estpost- is a user written command that makes it much
easier to store results from certain commands. */
capture which estpost
if _rc==111 ssc install estout.pkg
// checks if estpost exists
// installs if it doesn't

```

Setting directories and relative file-paths

You may have noticed one thing that the file header shown above is missing: a `cd` statement, changing the ‘home’ directory of the program using an absolute file path – something like this:

```


* Set home directory
cd "K:/Divisions/Research/Policies and procedures/Reproducibility and transparency"

```

This is because **you should never set a file directory using the `cd` command, either in the Master do-file or in any other program**, and you should avoid using *absolute file paths* (with the full path specified) whenever possible. This is, in part, to avoid one big problem with do-files: when their root folder structure changes (when a project is archived, for example), any file path that is in the do-file breaks (and the do-file will no longer run until you go through and update all of them).

Instead, you should use *relative file paths* whenever you need to call a piece of data or code from somewhere else in your project folder. By default, when you open a do-file by double clicking on it, your home directory is set to the folder that it is in (ideally, a folder called “Programs”). Relative file paths take as a given that you are starting from that home directory, and define a folder path from there. So if you wanted to use some data saved in your general project folder, housed in a folder called “Data”, instead of writing the full file path, you should refer to it based on the relationship to your Program folder.

Example folder structure:

Name	Date modified	Type
 Data	5/3/2017 9:47 AM	File folder
 Output	5/3/2017 9:47 AM	File folder
 Programs	5/3/2017 9:49 AM	File folder

Incorrect way to call data:

```

* Load auto dataset
use "I:/Policies and procedures/Data policy/Reproducibility and transparency/In Progress/ExampleCode/Example file structure/Data/auto.dta"

```

Correct way to call data:

```

* Load auto dataset
use "../Data/auto.dta"

```

As long as you opened your do-file by double clicking on it, Stata assumes you are starting in your program folder. The two dots (..) let Stata know that you want to go back a level, to the “Example file structure” folder, and then navigate from there as normal to the “Data” folder, and the “auto.dta” dataset housed within it.

This approach has several key advantages:

- As long as the general folder structure of your project folder stays the same, your file paths will never break – even when the project is archived.
- Similarly, if you want to share code or data with a collaborator (or work on it from a computer not connected to the network), all you have to do is make sure the relative folder structure stays intact (in this case, that your programs folder, data folder, and output folder

all stay on the same level). You could send the folder structure to a collaborator in a zipped file to help maintain consistent folder structure on the research team.

- Your Master do-file “knows” where to find the do-files it runs, since the home directory is by default the program folder in which all of your do-files are housed.

A few other notes on relative file paths

- Always use forward slashes (“/”), not backwards slashes (“\”). This preserves cross-platform compatibility, which is important if you want someone on a Mac (or Linux) computer to be able to run your files.
- If you need to go up more level, just repeat the two dots, separated by a forward slash. So if you wanted to go up two levels, you would type “../..” at the beginning of your file path. If you are trying to access a file that’s in the same folder as your do-file, no need to include the “../”; just write the file name in quotes.
- Sometimes it is necessary to use absolute file paths. In particular, if you need to use data that is not in your project folder (for example, the NSFG datasets, which are housed on the I: Drive), and which would be impractical or duplicative to copy, it is fine to call the dataset using the full file path name:

```
* Load NSFG dataset  
use "I:/Datasets/External/NSFG/2011-2013/Data/Female/Final/NSFG 2011-2013 female - Guttmacher final.dta"
```

It is less likely that these file paths will break, as the I: drive structure does not get reorganized frequently, and it would be fairly simple to fix these few file paths in the occasion that it does.

- Using relative file paths requires thinking about and setting up your general project folder structure, at least for your Data, Program and Output folders, before you start coding. Look to the archiving guidelines for some suggested folder structures for different types of projects. Setting up your folders this way will also help you later on down the road with project archiving.

SECTION 2: HARD CODING VS. MACROS

Hard coding is the practice of using literal values in code instead of variables. Hard coding should be avoided whenever possible. Take the following code, which calculates the mean and standard deviation of a variable, and then uses those calculated values to standardize.

```
* Calculate mean and standard deviation of weight
summ weight
* The mean was 3019.459, SD was 777.1936

* Standardize weight variable
gen weightstd = (weight-3019.459)/777.1936
```

The problem with coding in this way is that if the values of the mean and standard deviation change (because you drop values, or use a newer dataset), the code will now be incorrect, and all of these values will have to change (and if you forget, it will introduce an error). It is much preferable to use Stata's own stored results - in this case, stored in *r(mean)* and *r(sd)* - to carry out the same calculation:

```
* Calculate mean and standard deviation of weight
summ weight

* Use stored results to standardize weight variable
gen weightstd = (weight-r(mean))/r(sd)
```

Now the mean and the standard deviation adapt automatically, without a need to change the code (and with less potential for errors). It also becomes easier to use in other programs directly.

There are almost always ways to avoid hard coding numbers into your code. If it is unavoidable, however, it is best to define it in a [local](#) at the beginning of the file (with a large and distinct call out to the hard coding in a comment), and then refer to this local later on.

```
* Total number of cars in the US
* Source: IHS Automotive
* HARD CODED: MUST BE CHANGED MANUALLY
local carpop 253000000

* ...[other code]....

* Multiply mean by the total number of cars in the US
* (This gives us the total weight of cars in the country)
summ weight
gen totalweight = r(mean) * `carpop'
```

Other tips and guidelines:

- You can always see what results are stored by a command by typing [return list](#) and/or [ereturn list](#) after the command is run; these will give you a list of stored results and their names. If you can't find a stored estimate, it's a good idea to check both the *return list* and the *ereturn list*, as it's not always clear which commands store results in which.
- For more complicated situations or commands, it can also be useful to access the full table produced by Stata. This is almost always stored in a matrix (often called *r(table)* or *e(b)*); you can then refer to specific cells within that matrix using Stata's matrix notation (not covered in this guide).
- Stata has a set of [extended macro functions](#), which can be helpful for extracting other aspects of variables (such as value labels, or longer variable names) and then using them elsewhere in your file. Find out more about these by typing `help extended_fcn`.

- *Locals* are “local” to the do-file that created them. This means that after the do-file runs, they disappear: there is no way to access their contents. If you ever need to see the contents of a local, just include a *display* statement in your code that prints the contents to the results screen:

```
* Display contents of local carpop  
display "`carpop'"
```

- One thing to watch out for is that Stata reads an undefined local as blank space. So if we hadn't written “local carpop 2530000000” in the code above, Stata would have read the subsequent line as “gen totalweight = r(mean) *”. In that case, it would have just thrown an error and stopped the program, but sometimes this can have unexpected effects.
- Avoid using *globals* whenever possible. If you don't what these are, don't worry about them! If you do, know that there is voluminous documentation on the types of problems these tricky types of macros can cause in your program, in pretty much any programming language.

SECTION 3: COMMENTING AND READABILITY

Commenting code adequately is fundamental to making analyses reproducible and transparent. As a basic expectation, every program that is a part of your analysis should be commented sufficiently that a research assistant with basic programming experience could follow each step. Comments should be a mix of:

- *Section headers*, which should contain longer descriptions of the goal(s) of the code in that section.
- *One-line or block comments*, which describe what specific pieces of code are doing, and
- *In line comments* (using //), to explain more complicated pieces of syntax that may not be familiar to most readers (or which you may not remember how to interpret upon returning to the code). Programmers should use their judgment to determine how much inline / line-by-line commenting is necessary.

Comments should typically address one (or both) of these questions:

1. **What does this code block do?**
2. **Why did I implement this block this particular way?**

(In some cases, it might also be useful to write a comment about the *expected result* of a piece of code. If you merge two files, and you know that one case should be unmatched by design, then it might be useful to write a comment to that effect. Or if you write code to list cases with 'suspicious' values, it is often useful to write a comment with the number of cases you found, and what you decided to do with them.)

In addition, each do-file should have a *file header* to orient the reader with a few key pieces of information, as well as set up the system environment. The file header should have the program name, the project code, a short description of what the program does (along with any important notes), the date it was created, and the person who created it. If the code was adapted from a previous or similar project, this should be noted as well.

Example of a file header:

<pre>* Program name: Import 2013 Teen Data by Race (from DOH) * Project(s): 435 * Program task: /* This program loads the data collected and compiled from state departments of health (DOH), and reshapes it into a dataset with one row for each state, and variables for each age-category and race combination. It's extremely important that the Excel file with the DOH data is in the correct format -- see the detailed notes below Task 1. */ * Date created: 09/08/2015 * Written by: Isaac MZ * System set-up clear all version 14.1 capture log close set more off * Set linesize. set linesize 90 * Set maximum number of variables set maxvar 10000</pre>	<pre>// clear everything in memory // set Stata version // close any open log files // uninterrupted output</pre>
--	---

Example of a section header

```
////////////////////////////////////  
////////////////////////////////////1. Basic Recodes////////////////////////////////////  
/*  
  This section:  
  1. Corrects the weights (since we are looking at  
    2008-10, not 2006-10, which requires different  
    weights),  
  2. Codes a "keepme" variable to determine which cases  
    to keep,  
  3. And creates a few other generic sociodemographic  
    recodes  
*/  
////////////////////////////////////
```

Example of one-line comment:

```
* Create dichotomous indicator: was mother HS graduate?  
recode educmom (1 = 0 "Less than HS")  
              (2 3 4 = 1 "HS or more"), gen(momhs)    ///
```

Example of in-line comments:

```
* Recode pregnancy intention into three categories:  
* 1) Intended (then or sooner), 2) Mistimed (later),  
* 3) Unwanted (did not want to get pregnant at that time  
  or ever)  
gen wantresp = .  
replace wantresp = 1 if (FEEL_PG == 1 | FEEL_PG == 3)    // Then or sooner  
replace wantresp = 2 if FEEL_PG == 4                    // Later  
replace wantresp = 3 if FEEL_PG == 2                    // Did not want  
replace wantresp = .b if FEEL_PG == .b                  // For now, we code "Don't know"  
                                                         // as missing, but may want to  
                                                         // revisit this  
label define wantresp 1 "Intended" 2 "Mistimed"  
                    3 "Unwanted"  
label values wanresp wantresp
```

Other guidelines:

- Keep comments as succinct as possible (e.g., one line) while not losing meaning.
- Include verbs in your comments (e.g., "Create dichotomous indicators..." instead of just "Dichotomous indicators...")
- Leave one space between the * or // or /* and your comment's first character.

```
* Compute average test score for each student  
egen average = mean(score), by(studentid)
```

- If a single-line comment needs to be long enough to extend beyond the screen/page width, turn it into a block comment using /* and */.

```
/*  
Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
Sed sodales facilisis varius. Nulla auctor lacus nibh,  
vitae tempus leo dapibus non. Nullam neque tellus,  
placerat ornare turpis ac, maximus consequat turpis. Ut  
fermentum augue ex, sed congue velit laoreet a.  
Pellentesque viverra mi euismod ornare dictum. Nam vitae  
quam ipsum. Aenean egestas, nisl ac iaculis consequat,  
nisl dictum elit, ac pulvinar orci massa vel lorem. In et  
condimentum ante. Duis eu fringilla turpis.  
*/
```

- Similarly, if a single line of code (command) is long enough to extend beyond the screen/page, break the code into multiple lines and use the triple-slash syntax (///) at the end of each line. Always indent the continuing lines of code.

Correct:

```
collapse (mean) income numdep biokids numpregs nummisc ///
                  numabort numpart score1 totmeth wantresp ///
                  , by(year)
```

Incorrect:

```
collapse (mean) income numdep biokids numpregs nummisc numabort numpart score1 totmeth wantresp, by(year)
```

Also incorrect

```
collapse (mean) income numdep biokids numpregs nummisc ///
numabort numpart score1 totmeth wantresp, by(year)
```

- Alternatively, if you expect to have a series of commands that extend beyond the screen/page, change the end-of-line delimiter from a carriage return to a semicolon. When you have finished the series, return the end-of-line delimiter to the default carriage return.
- Multiple commands that follow the same structure should align (within reason).

Correct:

```
gen time19 = vrylstag if vrylstag <= 19
replace time19 = 19 if vrylstag > 19 & vrylstag < .
replace time19 = ager if vrylstag >= . & ager <= 19
replace time19 = 19 if vrylstag >= . & ager > 19
```

Incorrect:

```
gen time19 = vrylstag if vrylstag <= 19
replace time19 = 19 if vrylstag > 19 & vrylstag < .
replace time19 = ager if vrylstag >= . & ager <= 19
replace time19 = 19 if vrylstag >= . & ager > 19
```

- Mathematical or logical operators (=, <, !=, etc.) should be preceded and followed by a single space.

Correct:

```
gen time19 = vrylstag if vrylstag <= 19
```

Incorrect:

```
gen time19=vrylstag if vrylstag<=19
```

- Comments can also be used to record tasks that still need to be carried out, or to communicate between multiple people working on a project. So that these comments stand out and are easily searchable, they should start with a label in all caps (such as “TO DO:”, “CHECK:”) or your name and the name of the person you are communicating with (e.g. “IMZ to LL”).

Example of comment to a collaborator:

```
* Create dichotomous indicator of >3 year age
* difference with first partner
gen agediscrep = .
replace agediscrep = 0 if agediff <= 3
replace agediscrep = 1 if agediff > 3 & agediff < .
* IMZ to LL: Did we want to set this as >3 or >=3?
```

Example of TO DO:

```
* Create a dichotomous variable for had sex at 16 years
* or younger
* Note: We purposely do not drop cases below 10 (as we
* typically do) to match the YRBS approach.
gen earlysex = .
replace earlysex = 1 if vry1stag <= 16
replace earlysex = 0 if vry1stag > 16 & vry1stag <.
replace earlysex = 0 if hadsex == 0
* TO DO: Run sensitivity analysis dropping cases below 10,
* to see if it effects our results substantially
```

- Before finalizing a project do-file, **all of these task-focused comments should be deleted** (this is also a good way to check that all outstanding issues have been addressed.)

SECTION 4: LOOPS AND NESTED STATEMENTS

Loops — using the [foreach](#) or [forvalues](#) commands — are a way to do a series of repetitive or linked commands in Stata in a relatively concise way. Loops are useful not only because they save typing, but also because they can prevent errors.

If you have a series of repetitive lines of code, it can be easy to make a typo in one of them (and it can be hard to spot):

Can you find the error in this code?

```
* For each biological child of male respondent, recode
* as unmarried at birth if mother of child is not
* coded as former or current wife
replace biomar1=0 if parent01!=1 & parent01!=3
replace biomar2=0 if parent02!=1 & parent02!=3
replace biomar3=0 if parent03!=1 & parent03!=3
replace biomar4=0 if parent04!=1 & parent03!=3
replace biomar5=0 if parent05!=1 & parent05!=3
replace biomar6=0 if parent06!=1 & parent06!=3
replace biomar7=0 if parent07!=1 & parent07!=3
replace biomar8=0 if parent08!=1 & parent08!=3
replace biomar9=0 if parent09!=1 & parent09!=3
```

Hint: look at the 6th and 7th lines of code

A better way:

```
* For each biological child of male respondent,
* recode as unmarried at birth if mother of
* child is not coded as former or current wife.
foreach mynum of numlist 1/9 {                               // (max num of children is 9)

    replace biomar`mynum' = 0 if parent0`mynum' != 1 &      /// (1 is code for current wife)
                                parent0`mynum' != 3          /// (3 is code for former wife)

} // End of mynum loop
```

Now there is only one line of code to check, as opposed to 8 – and if that line is right, so are all of the others.

To make your loops more useful, it can also be helpful to use conditional logic statements such as if and else. These evaluate a condition, and if it is true, execute a particular piece of code. The code below, for example, checks to see if each variable in a set of variables is numeric; if it is, it executes a recode, and calculates a mean for each gender, if it's not, it assumes it is a string variable, and tabulates it (*see example code on next page*).

```

* Descriptive statistics
local myvars numbirth age sitename
foreach myvar of varlist `myvars' {

    * See if variable is numeric
    capture confirm numeric variable `myvar'

    * If it is numeric, recode any 5s as 0s
    if _rc==0 {

        recode `myvar' (5=0)

        * And calculate mean for each gender
        foreach mygender of numlist 1/2 {

            mean `myvar' if gender == `mygender'

        } // end of mygender loop over gender
    } // end of if numeric loop

    * If it is string, tabulate
    else {

        tab `myvar'

    } // end of else loop (strings)

} // end of myvar loop over variables

```

As you can see, however, *if* and *else* statements can also make code harder to read, especially as they get more complicated. There isn't a hard and fast rule about when they are appropriate or not – as a programmer, you need to carefully balance the ability to make your code more flexible and concise with its readability.

It is also important to note that if conditions do not select individual cases!

Incorrect:

```

if year == 1960 {
    replace coolvar = 1
}
else {
    replace coolvar = 0
}

```

The *if* programming command is different from the *if* statement that appears at the end of an analysis command. The code above would check to see if year was equal to 1960 for the *first observation*, and if it was, it would code the *whole dataset* as 1 on coolvar. This is almost never what you want.

Loop formatting

- Locals defined within a loop should follow the naming convention ``my[something]'`, so that they are easy to identify within the code.
- The open-brace or left-brace (`{`) should always be on the same line as the start of the loop or condition and the close-brace or right-brace (`}`) should always be on its own line:

Correct:

```
foreach myvar of varlist `myvars' {  
  recode `myvar' (5=0)  
} // End of myvar loop
```

Incorrect (in multiple ways):

```
foreach myvar of varlist `myvars'  
{  
  recode `myvar' (5=0) }
```

- Always indent the lines after a loop begins. If loops are nested, add an additional indent to the 'inner' loop. This helps a reader keep track of which 'level' of a loop a given command operates on.

Correct:

```
foreach myvar of varlist `myvars' {  
  recode `myvar' (5=0)  
  foreach mygender of numlist 1/2 {  
    mean `myvar' if gender==`mygender'  
  } // End of mygender loop  
} // End of myvar loop
```

Incorrect:

```
foreach myvar of varlist `myvars' {  
  recode `myvar' (5=0)  
  foreach mygender of numlist 1/2 {  
    mean `myvar' if gender==`mygender'  
  }  
}
```

- Also indent after a piece of conditional logic ("if" or "else").
- Comment the end of a large block of nested conditional logic – such as a loop or if statement.

```
* Descriptive statistics  
local myvars numbirth ager sitename  
foreach myvar of varlist `myvars' {  
  * See if variable is numeric  
  capture confirm numeric variable `myvar'  
  * If it is numeric, recode any 5s as 0s  
  if _rc==0 {  
    recode `myvar' (5=0)  
    * And calculate mean for each gender  
    foreach mygender of numlist 1/2 {  
      mean `myvar' if gender == `mygender'  
    } // end of mygender loop over gender  
  } // end of if numeric loop  
  * If it is string, tabulate  
  else {  
    tab `myvar'  
  } // end of else loop (strings)  
} // end of myvar loop over variables
```


- Add display statements within your loop, especially as they get more complicated. One of the drawbacks of loops is that it can be hard to tell which iteration of the loop your code is on; adding display statements makes output more readable, and makes it easier to tell if your loop is working correctly (and makes it easier to debug). The output from the code above is much more readable if we write it this way:

```
* Descriptive statistics
local myvars numbirth ager sitename
foreach myvar of varlist `myvars' {

  * See if variable is numeric
  capture confirm numeric variable `myvar'

  * If it is numeric, recode any 5s as 0s
  if _rc==0 {

    recode `myvar' (5=0)

    * And calculate mean for each gender
    foreach mygender of numlist 1/2 {

      display _newline          ///
      "Mean of `myvar' for gender `mygender'"
      mean `myvar' if gender == `mygender'

    } // end of mygender loop over gender
  } // end of if numeric loop

  * If it is string, tabulate
  else {

    display _newline          ///
    "Tabulation of `myvar' (all genders)"
    tab `myvar'

  } // end of else loop (strings)
} // end of myvar loop over variables
```

SECTION 5: VARIABLE NAMES AND LABELS

Good variable naming is fundamental to the readability and robustness of your code. At its most basic, code isn't readable if variable names aren't, and overly abbreviated or oblique variable names can often be harder to interpret than badly commented code. There are a variety of different naming philosophies for variables, each of which make sense in different contexts. Stata itself gives few limitations. Variables can be up to 32 characters, and contain any mix of numbers, letters and underscores. No special characters or spaces are permitted, and variable names cannot start with a number or underscore; in addition, a few shorter words (such as *in* and *if*) are reserved for Stata syntax itself.

Just because you have this flexibility, however, doesn't mean that you should use it. Even if it is possible to create a variable called `gH__JH648_jN`, it fails at its most fundamental task: for other people to be able to interpret it easily, for you to remember what it signifies, and (though this last one is less important) for it not to be a pain to type out in your code. Variable names should be intuitive enough for others to interpret the meaning and content at first glance. In general, do not shy away from longer descriptive names with multiple words as opposed to shorter abbreviated names that may not convey the proper meaning. That said, the ideal length of a variable name is much less than 32 characters; good variable naming is a careful balance between brevity and meaning.

The most important principle of variable naming is to be **consistent**. It is fine to use capitalization (remember that Stata is caps sensitive) or not, or use different formats for different types of variables, as long as it is consistent throughout your code, and clearly communicated to the people that you are working with. If you are using an unusual naming convention, or one that requires explanation, it may also be worth describing in either the Master do-file header or the file header of the relevant do-file. This section is not as prescriptive as the ones that preceded it, as general rules of variable naming are likely too rigid for the many different types of projects and analyses at the institute. There are some specific guidelines however, that all projects should ideally follow – either because of peculiarities of the Stata system itself, or because of established community norms.

Specific guidelines

- All variables should have both variable labels and value labels.
- Any specific types of missing values in the dataset (refusals, skip patterns, etc.) should be labelled as well, using Stata's [extended missing values](#).
- Dichotomous variables should be named after the category that corresponds to the value "1". So, for example, if you create a dichotomous indicator for marriage (where 0 is not married and 1 is married) the variable should be called **married**. If the variable was reverse coded (so 0 was married and 1 is not married), the variable should instead be called **unmarried**.
- Along those same lines, dichotomous variables should always be coded 0/1, where 0 = no and 1 = yes. It's sometimes necessary for primary data collection for yes/no answers to be keypunched as different values (and sometimes external surveys, such as the NSFG, do not follow these rules); in these situations, all dichotomous indicators must be recoded as part of routine data cleaning. This is important for clarity and consistency; it also is important because of some of the statistical properties of a 0/1 dichotomous variable (to give just one example, only 0/1 variables can be included as the dependent measure in a logistic regression).

- It is often helpful to have a few set suffixes for common variable transformations. Continuous variables recoded as categorical can have the suffix `cat[#]` appended, where the number is the number of categories. A five category recode of a continuous variable called **age**, for example, might be called **agecat5**.
- In contrast, it is not helpful to add the letter “r” to the end of your variable name to signify a recode, as it does not give the reader sufficient information about *what* that recode was.
- For series of related variables, it is often helpful to have a common prefix. The first four contraceptive current contraceptive methods mentioned by a respondent might be called `constat1`, `constat2`, `constat3` and `constat4`. This also often applies to check all that apply questions, which might have a series of dichotomous indicators of responses to a single question. Coding in this way requires extra information in the variable label to distinguish between variable names, but can make coding in loops easier, and can make relationships between variables easier to see.
- For variable labels (not variable names), place distinguishing information for a series of variables that have a long string of repeated text at the beginning of the label to avoid truncation of the unique information at the end which can occur in some statistical procedures. For example:
CORRECT: ‘Pill: method used after last abortion in the past 2 years’
CORRECT: ‘Female Condom: method used after last abortion in the past 2 years’
INCORRECT: ‘Family planning method used after the last abortion in the past 2 years- pill’
INCORRECT: ‘Family planning method used after the last abortion in the past 2 years- female condom’
- Similar projects (AICMs, for example), should ideally use similar variable names for key variables. This makes it easier to adapt code for future work, and makes code more readable across projects.