Appendix A

The DIME Analytics Coding Guide

Most academic programs that prepare students for a career in the type of work discussed in this book spend a disproportionately small amount of time teaching students coding skills in relation to the share of professional time those students will spend writing code in their first few years after graduation. Recent masters-level graduates joining the DIME team have demonstrated very good theoretical knowledge but require a lot of training in practical skills. This is like an architecture graduate having learned how to sketch, describe, and discuss the concepts and requirements of a new building very well, but lacking the technical ability to contribute to a blueprint following professional standards that others can use and understand. The reasons for this disconnect are a topic for another book, but, in today's data-driven world, people working in quantitative development research must be proficient collaborative programmers, which requires more than being able to compute correct numbers.

This appendix starts by offering general and language-agnostic principles on how to write "good" code. Good code not only generates correct results *but also* is easily read and adapted by other professionals. The second section contains instructions on how to access and use the code examples provided in this book. The last section presents the DIME Analytics Stata Style Guide. Widely accepted and used style guides are common in most programming languages, but, as yet, there is no sufficiently encompassing style guide for Stata. The DIME Analytics Stata Style Guide is intended to increase the emphasis given to using, improving, sharing, and standardizing code style among the Stata community. It shares practices that greatly improve the quality of research projects coded in Stata. By applying the guidance provided in this appendix, readers will learn to write code that, like an architect's blueprint, can be understood and used by everyone in the trade.

Writing good code

Good code has two elements: (1) it is correct, in that it does not produce any errors and its outputs are the objects intended, and (2) it is useful and comprehensible to someone who has not seen it before (or even someone who sees it again weeks, months, or years later). Many researchers

have only been trained to code correctly. However, when code runs on a computer and obtains the desired results, the job of writing *good* code is only half done. Good code is easy to read and replicate, making it easier to spot mistakes. Good code reduces sampling, randomization, and cleaning errors. Good code can be reviewed easily by others before it is published and can be reused afterward. The best code is written as if a stranger will be reading it.

Good code has three major elements: structure, syntax, and style. The *structure* is the environment and file organization in which the code lives: good structure means that it is easy to find individual pieces of code, within and across files, that correspond to specific tasks and outputs. It also means that functional code blocks are sufficiently independent from each other such that they can be shuffled around, repurposed, and even deleted without affecting the execution of other portions. The *syntax* is the literal language of code. Good syntax means that the code is readable in terms of how its mechanics implement ideas; it should not require arcane reverse-engineering to figure out what a code chunk is trying to do. It should use common commands in a generally accepted way so that others can easily follow and reconstruct the researcher's intentions. Finally, *style* is the way that the nonfunctional elements of code convey its purpose. Elements like spacing, indentation, and naming conventions (or lack thereof) can make code much more (or much less) accessible to someone who is reading it for the first time and needs to understand it quickly and accurately.

One key tool for writing good code is using help documentation. Regardless of how much experience a person has with a particular programming language—Stata, R, Python, or one of the many others—it is helpful to revisit help files frequently. It is impossible to overemphasize how important it is to get into the habit of reading help files. Even for the most common commands, there is always something new to learn. A help file window should be open at all times, making it easy to look up commands or uses of commands that are unfamiliar or whose functionality has been forgotten. The belief that help files are only for beginners could not be further from the truth. The only way to get better at a programming language is to read help files often. Stata help files can be accessed by writing `help [command]` whenever necessary. For example, to learn about `isid`, writing `help isid` will open the help file for that command.

Using the code examples in this book

This book provides code boxes throughout the chapters that offer examples of good code execution of some of the most common tasks in quantitative development research. Stata is one of several programming languages used at DIME, but this book focuses on Stata for the code boxes because few high-quality resources exist relative to Stata's frequency of use in development research. The code examples in each of the code boxes and most of the code in this appendix rely on preinstalled data

sets as often as possible, so that they will run independently of any other materials. (By contrast, the Demand for Safe Spaces case study code examples can only be run together with the rest of the reproducibility package). The code boxes also demonstrate best-practice coding style for researchers, such as the generous use of comments. In the book, code examples are presented like the following:

```
* Load the auto dataset
sysuse auto.dta, clear

* Run a simple regression
reg price mpg rep78 headroom, coefl

* Transpose and store the output
matrix results = r(table)'

* Load the results into memory
clear
svmat results, n(col)
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

The raw code used in examples in this book can be accessed in several ways. DIME uses GitHub to version-control all of the content of this book, including the code boxes. To see the examples from the code boxes, go to https://github.com/worldbank/dime-data-handbook/tree/master/code. There is no need to download any data, because the examples use Stata's built-in data sets. If Stata is installed on the computer, then it already has the data files used in the code.

For readers not familiar with GitHub, the simplest way to access the code is to click the individual file in the GitHub link provided previously and then click the button labeled "Raw." Doing so will load a page that looks like the one at https://raw.githubusercontent.com/worldbank/dime-data-handbook/master/code/code.do. There, the code can be copied from the browser window into the do-file editor with the formatting intact. This method is practical for only a single file at a time. To download all code used in this book, instead go to https://github.com/worldbank/dime-data-handbook/archive/main.zip. That link downloads a `.zip` file with all of the content used in writing this book, including the plain-text files used for the book itself. After extracting the `.zip` file, all of the code will be in a folder called `/code/`.

The code boxes use built-in commands as much as possible, but user-written commands are also used when they provide important

ietoolkit is a Stata package containing several commands to routinize tasks in impact evaluation. It can be installed through the Boston College Statistical Software Components (SSC) archive (https://ideas.repec.org/c/boc/bocode/s458137.html), and the code is available at https://github.com/worldbank/ietoolkit. To learn more, see the DIME Wiki at https://dimewiki.worldbank.org/ietoolkit.

iefieldkit is a Stata package containing several commands to routinize tasks related to primary data collection. It can be installed through SSC (https://ideas.repec.org/c/boc/bocode/s458600.html), and the code is available at https://github.com/worldbank/iefieldkit. To learn more, see the DIME Wiki at https://dimewiki.worldbank.org/iefieldkit.

new functionality. In particular, the book points to two suites of Stata commands developed by DIME Analytics, `ietoolkit` and `iefieldkit`, which were written to standardize core data collection, management, and analysis workflows.

To run the code box examples that include user-written commands, it is necessary to install the commands first. The most common place to distribute user-written commands for Stata is the Boston College Statistical Software Components (SSC) archive (https://ideas.repec.org/s/boc/bocode.html). The user-written commands in this book are all available from the SSC archive. Installation of commands from the SSC archive is straightforward, simply type `ssc install randtreat`.

Some commands on SSC are distributed in packages, in which case it is necessary to download the whole package to access the included commands. This is the case, for example, of `ieboilstart`, which is part of the `ietoolkit` package. Commands that are distributed in packages cannot be installed on their own; it will not work to type `ssc install ieboilstart`. Instead, Stata will suggest using `findit ieboilstart`, which will search SSC (among other places) for a package containing a command called `ieboilstart`. Stata will find `ieboilstart` in the package `ietoolkit` and suggest installing it by typing `ssc install ietoolkit` in Stata instead.

Although it can be confusing to work with packages for the first time, doing so is the best way to set up a Stata installation and to benefit from the publicly available work of others. After learning how to install commands like this, it will not be confusing at all. When writing code that relies on user-written commands, it is best practice to install such commands at the beginning of the master do-file, so that the user does not have to search for packages manually.

The DIME Analytics Stata Style Guide

The programming languages used in computer science always have associated style guides. Sometimes they are official, universally agreed-upon style guides, such as PEP8 for Python (van Rossum, Warsaw, and Coghlan 2013). More commonly, they are well-recognized but unofficial style guides like Hadley Wickham's *Tidyverse Style Guide* for R (Wickham, n.d.) or the JavaScript Standard Style for JavaScript (https://standardjs.com/#the-rules). It is also common for large software companies to maintain their own style guides for all languages used in their projects. However, these are not always made public.

Aesthetics are an important part of style guides, but not the main reason for their existence. Rather, style guides allow programmers who are likely to work together to share conventions and understandings of how to execute various common intentions using mutually understandable code language. They also help to improve the quality of the code produced by all programmers using that language. By using a shared style, newer

programmers can learn from more experienced programmers how certain coding practices are more or less prone to errors.

The best style guide is the one that is adopted the most widely. Broadly accepted style conventions make it easier for coders to borrow solutions from each other and from examples online without causing bugs that might be found too late. Similarly, globally standardized style guides make it easier for programmers to collaborate on each other's problems and to move from project to project and from team to team.

There is room for personal preference when using style guides, but style guides are first and foremost about quality and standardization, especially when collaborating on code. DIME Analytics created this Stata style guide to improve the quality of all code written in Stata. It is not necessary to follow the style guide precisely. All style rules introduced in this section follow the DIME Analytics suggestion for how to code, but the most important recommendation is to make sure that the style used for code is *consistent*. This guide allows the DIME team to have a consistent code style.

Commenting code

Comments do not change the output of code or how it runs, but without them code will not be accessible to other readers. It will also take much longer to update or edit code written in the past if it does not have adequate comments explaining its intent and functionality. It is important to comment a lot: not only about *what* the code is doing but also about *why* it was written the way it was. In general, writing simpler code that needs less explanation is preferrable to using an elegant and complex method in less space, unless the advanced method is widely accepted.

There are three types of comments in Stata, and they have different purposes.

Commenting multiple lines

```
/*
    This is a do-file with examples of comments in Stata.
    This type of comment is used to document all of the do-file or a large
    section of it
*/
```

Commenting a single line

```
    * Standardize settings, explicitly set version, and clear memory
    * (This comment is used to document a task covering at maximum a few lines of code)
    ieboilstart, version(13.1)
    `r(version)'
```

(Continues on next page)

Inline comments

```
* Open the dataset
sysuse auto.dta // Built in dataset (This comment is used to document a single line)
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

Abbreviating commands

Stata commands can often be abbreviated in the code. A command can be abbreviated if the help file indicates an abbreviation by underlining part of the name in the syntax section at the top. Only built-in commands can be abbreviated; user-written commands cannot. Although Stata allows some commands to be abbreviated to one or two characters, doing so can be confusing—two-letter abbreviations can rarely be "pronounced" in an obvious way that connects them to the functionality of the full command. Therefore, command abbreviations in code should not be shorter than three characters, with the exception of `tw` for `twoway` and `di` for `display`, and abbreviations should be used only when a widely accepted abbreviation exists. (Many commands also allow options to be abbreviated: these abbreviations are always acceptable at the shortest allowed abbreviation.) The frequently used commands `local`, `global`, `save`, `merge`, `append`, or `sort` should never be abbreviated. The table below lists accepted abbreviations of common Stata commands.

Accepted abbreviations of common Stata commands

Abbreviation	Command
tw	twoway
di	display
gen	generate
mat	matrix
reg	regress
lab	label
sum	summarize
tab	tabulate
bys	bysort
qui	quietly
noi	noisily
cap	capture
forv	forvalues
prog	program
hist	histogram

Abbreviating variable names

Variable names should never be abbreviated; they should be written out completely. Code may change if a variable is introduced later that makes the abbreviation no longer unique. `ieboilstart` executes the command `set varabbrev off` by default and will therefore break any code using variable abbreviations.

Using wildcards and lists in Stata for variable lists (`*`, `?`, and `-`) is also discouraged, because the functionality of the code may change if the data set is changed or even simply reordered. To capture all variables of a certain type, it is better to use `unab` or `lookfor` to build that list in a local macro, which can then be checked so that the right variables are in the right order.

Writing loops

In example code in Stata and other languages, it is common for the name of the local generated by `foreach` or `forvalues` to be something as simple as `i` or `j`. It is preferable, however, to name that index descriptively. One-letter indexes are acceptable only for general examples, for looping through *iterations* with `i`, and for looping across matrices with `i` and `j`. Best practice is for index names to describe what the code is looping over—for example, household members, crops, or medicines. Even counters should be named explicitly. Doing so makes code much more readable, particularly in nested loops.

GOOD:

```
* Loop over crops
foreach crop in potato cassava maize {
    * do something to `crop'
}
```

GOOD:

```
* Loop over crops
local crops potato cassava maize
foreach crop of local crops {
    * Loop over plot number
    forvalues plot_num = 1/10 {
        * do something to `crop' in `plot_num'
    } // End plot loop
} // End crop loop
```

(Continues on next page)

BAD:

```
* Loop over crops
foreach i in potato cassava maize {
    * do something to `i'
}
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

Using white space

In Stata, adding one or many spaces does not change the execution of code and can make the code much more readable. Most researchers are well trained in using white space in software like PowerPoint and Excel: a PowerPoint presentation would not have text that does not align, and an Excel table would not have unstructured rows and columns. The same principles apply to coding. In the example below, the exact same code is written twice, but in the better example white space is used to signal that the central object of this segment of code is the variable `employed`. Organizing the code in this way makes it much quicker to read, and small typos stand out more, making them easier to spot.

ACCEPTABLE:

```
* Create dummy for being employed
gen employed = 1
replace employed = 0 if (_merge == 2)
lab var employed "Person exists in employment data"
lab def yesno 1 "Yes" 0 "No"
lab val employed yesno
```

GOOD:

```
* Create dummy for being employed
gen      employed = 1
replace  employed = 0 if (_merge == 2)
lab var  employed "Person exists in employment data"
lab def           yesno 1 "Yes" 0 "No"
lab val  employed yesno
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

Indentation is another type of white space that makes code more readable. Any segment of code that is repeated in a loop or conditional on an `if`-statement should be indented four spaces relative to either the loop or the condition as well as the closing curly brace. Similarly, continuing lines of code should be indented under the initial command. If a segment is in a loop inside a loop, then it should be indented another four spaces, making it eight spaces more indented than the main code. In some code editors, this indentation can be achieved by using the tab button. However, the type of tab used in the Stata do-file editor does not always display the same across platforms, such as when publishing code on GitHub. Therefore, inserting four spaces manually is recommended instead of using a tab.

GOOD:

```
    * Loop over crops
    foreach crop in potato cassava maize {
        * Loop over plot number
        forvalues plot_num = 1/10 {
            gen crop_`crop'_`plot_num' = "`crop'"
        }
    }

    * or
    local sampleSize = `c(N)'
    if (`sampleSize' <= 100) {
        gen use_sample = 0
    }
    else {
        gen use_sample = 1
    }
```

BAD:

```
    * Loop over crops
    foreach crop in potato cassava maize {
    * Loop over plot number
    forvalues plot_num = 1/10 {
    gen crop_`crop'_`plot_num' = "`crop'"
    }
    }

    * or
    local sampleSize = `c(N)'
    if (`sampleSize' <= 100) {
```

(Continues on next page)

```
    gen use_sample = 0
    }
    else {
    gen use_sample = 1
    }
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

Writing conditional expressions

All conditional (true/false) expressions should be within at least one set of parentheses. The negation of logical expressions should use bang (`!`) and not tilde (`~`). Explicit truth checks should be used (`if `value' == 1`) rather than implicit ones (`if `value'`). The `missing(`var')` function should be used instead of arguments like (`if `var' >= .`). It is important always to consider whether missing values will affect the evaluation of conditional expressions and modify them appropriately.

GOOD:

```
    replace gender_string = "Woman" if  (gender == 1)
    replace gender_string = "Man"   if ((gender != 1) & !missing(gender))
```

BAD:

```
    replace gender_string = "Woman" if  gender == 1
    replace gender_string = "Man"   if (gender ~= 1)
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

When applicable, `if-else` statements should be used even if the same thing can be expressed with two separate `if` statements. The use of `if-else` statements communicates that the two cases are mutually exclusive, which makes code more readable. It is also less error-prone and easier to update if the conditional statement needs to be modified.

GOOD:

```
    if (`sampleSize' <= 100) {
        * do something
    }
    else {
        * do something else
    }
```

BAD:

```
    if (`sampleSize' <= 100) {
        * do something
    }
    if (`sampleSize' > 100) {
        * do something else
    }
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

Writing file paths

All file paths should always be enclosed in double quotes and should always use forward slashes for folder hierarchies (`/`). Mac and Linux computers cannot read file paths with back slashes, and back slashes cannot be removed easily with find-and-replace because the character has other functional uses in code. File names should be written in lowercase with dashes (`my-file.dta`). File paths should always include the file extension (`.dta`, `.do`, `.csv`, and so forth). Omitting the extension causes ambiguity if another file with the same name is created, even if there is a default file type.

File paths should also be absolute and dynamic. *Absolute* means that all file paths start at the root folder of the computer, often `C:/` in Windows or `/Users/` in macOS. Doing so ensures that the correct file is always in the correct folder. The `cd` command should not be used unless a command specifically requires it. When using `cd`, it is easy to overwrite a file in another project folder, because many Stata commands implicitly use `cd`, and therefore the working directory in Stata often changes without warning. Relative file paths are common in many other programming languages, but, unless they are relative to the location of the file running the code, using them is a risky practice. In Stata, relative file paths are relative to the working directory, not to the code file being run.

Dynamic file paths use global macros for the location of the root folder. These globals should be set in a central master do-file. Using the root folder path stored in a global makes it possible to write file paths in Stata that work very similarly to relative paths. It also achieves the functionality that setting `cd` is often intended to achieve: executing the code on a new system only requires updating file path globals in one location. If global names are unique, there is no risk that files will be saved in the incorrect project folder. Multiple folder globals can be created as needed, and this practice is encouraged.

GOOD: Absolute and dynamic paths

```
global myDocs    = "C:/Users/username/Documents"
global myProject = "${myDocs}/MyProject"
use "${myProject}/my-dataset.dta", clear
```

BAD: Relative paths

```
cd "C:/Users/username/Documents/MyProject"
use MyDataset.dta
```

BAD: Static paths

```
use "C:/Users/username/Documents/MyProject/MyDataset.dta"
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

Using line breaks

Long lines of code are difficult to read, making it necessary to scroll left and right to see the full line of code. When a line of code is wider than text on a regular paper, a line break is needed. A common line-breaking length is around 80 characters. Stata's do-file editor and other code editors provide a visible guideline. Around that length, using `///` breaks the line in the code editor, while telling Stata that the same line of code continues on the next line. Recent versions of the Stata do-file editor—and many other code editors—automatically wrap code lines that are too long. We do not recommend relying on this functionality; instead, actively using `///` to wrap lines is recommended to ensure that line breaks are placed such that the code remains the most readable. The `///` breaks do not need to be aligned vertically in code, although doing so may help to align comments and improve readability, because indentations should reflect

that the command continues to a new line. Lines should be broken where it makes functional sense. Writing comments after `///` just as with `//` usually is a good idea, especially if it is being used to separate functional parts of a single command for clarity.

The `#delimit` command should be used only for advanced function programming and is officially discouraged in analytical code (Cox 2005). Typing `/* */` should never be used to wrap a line: it is distracting and difficult to follow, and those characters should be used only to write regular comments. Line breaks and indentations may be used to highlight the placement of the option comma or other functional syntax in Stata commands.

GOOD:

```
graph hbar invil      /// Proportion in village
    if (priv == 1)    /// Private facilities only
  , over(statename, sort(1) descending)     /// Order states by values
    blabel(bar, format(%9.0f))              /// Label the bars
    ylab(0 "0%" 25 "25%" 50 "50%" 75 "75%" 100 "100%") ///
    ytit("Share of private primary care visits made in own village")
```

BAD:

```
#delimit ;
graph hbar
    invil if (priv == 1)
  , over(statename, sort(1) descending) blabel(bar, format(%9.0f))
    ylab(0 "0%" 25 "25%" 50 "50%" 75 "75%" 100 "100%")
    ytit("Share of private primary care visits made in own village");
#delimit cr
```

UGLY:

```
graph hbar /*
*/    invil if (priv == 1)
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

Using boilerplate code

Boilerplate code consists of a few lines of code that always appear at the top of the code file, and its purpose is to harmonize settings across users running the same code to the greatest degree possible. There is no way in Stata to guarantee that any two installations will always run code

in exactly the same way. In the vast majority of cases, they do, but not always, and boilerplate code can mitigate that risk. DIME Analytics developed the `ieboilstart` command to implement many commonly used boilerplate settings that are optimized given a particular installation of Stata. It requires two lines of code to execute the `version` setting, which avoids differences in results due to different versions of Stata. Among other things, it turns the `more` flag off so that code never hangs while waiting to display more output; it turns `varabbrev` off so that abbreviated variable names are rejected; and it maximizes the allowed memory usage and matrix size so that other machines do not reject code for violating system limits. (For example, Stata/SE and Stata/IC allow for different maximum numbers of variables, and the same happens with Stata 14 and Stata 15, so code written in one of these versions may not be able to run in another.) Finally, it clears all stored information in Stata memory, such as noninstalled programs and globals, getting as close as possible to opening a fresh Stata session.

GOOD:

```
1    ieboilstart, version(13.1)
2    `r(version)'
```

ACCEPTABLE:

```
1    set more off, perm
2    clear all
3    set maxvar 10000
4    version 13.1
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

Miscellaneous notes

Multiple graphs should be written as `tw (xx)(xx)(xx)`, not `tw xx||xx||xx`.

In simple expressions, spaces are needed around each binary operator except `^`, writing `gen z = x + y` and `gen z = x^2`.

When the order of operations applies, spacing and parentheses may be adjusted: write `hours + (minutes/60) + (seconds/3600)`, not `hours + minutes / 60 + seconds / 3600`. For long expressions, `+` and `-` operators should start the new line, but `*` and `/` should be used inline. For example:

```
gen newvar = x ///
             - (y/2) ///
             + a * (b - c)
```

To access this code in do-file format, visit the GitHub repository at https://github.com/worldbank/dime-data-handbook/tree/main/code.

Instead of printing code to the results window, which is slow, it is better to use `qui` whenever possible and to use `run file.do` rather than `do file.do` in master scripts. To minimize output printed to the command window, commands like `sum` and `tab` should be used sparingly in do-files, unless they are for the purpose of storing r-class statistics. In that case, using the `qui` prefix will prevent printing output. It is also faster to get outputs from commands like `reg` using the `qui` prefix.

References

Cox, Nicholas J. 2005. "Suggestions on Stata Programming Style." *The Stata Journal* 5 (4): 560–66.

van Rossum, Guido, Barry Warsaw, and Nick Coghlan. 2013. "PEP8: Style Guide for Python Code." Python. https://www.python.org/dev/peps/pep-0008/.

Wickham, Hadley. n.d. *The Tidyverse Style Guide.* Tydyverse. https://style.tidyverse.org/index.html.