

SOC 4650/5650 User's Guide

Christopher Prener, Ph.D.

2017-04-11

Contents

Preface	5
License	5
1 Getting Started	7
1.1 Prep Your Computer	7
1.2 Create Accounts	7
1.3 Download and Install Software	8
1.4 Buy Course Materials	8
1.5 Download Course Data	9
2 Approaching this Course	11
2.1 Zen and the Art of Data Analysis	11
2.2 An Apple a Day	11
2.3 Reading with Purpose	12
2.4 Active Lectures and Labs	12
2.5 Typefaces and Examples	12
3 “Good Enough” Research Practices	15
3.1 Reproducibility	15
3.2 Thinking in Workflows	17
3.3 Course Tools	18
3.4 Course Workflow	19
4 Protecting Your Work	21
4.1 Data Management	21
4.2 Creating a Sustainable File System	22
4.3 Backing Up Your Data	25
5 Introduction to GitHub	27
5.1 Git	27
5.2 More Git-lingo	27
5.3 GitHub.com	28
5.4 The Workflow of Git and GitHub	28
5.5 GitHub Repositories	30
5.6 Storing GitHub Repositories	30
5.7 GitHub Issues	30
5.8 GitHub Desktop Application	31
5.9 Learning More	31
6 Advanced GitHub	33
6.1 Even More Git-lingo	33
6.2 Creating Branches	33

7	Introduction to Atom	39
7.1	Packages	39
7.2	Languages	39
7.3	Using Snippets	40
7.4	Using Panes	42
7.5	Using Project Folders	43
8	Reproducible Do-Files	45
8.1	Using the Stata Snippet	45
8.2	Weaving Do-files	46
8.3	Snippet Details: Header	47
8.4	Snippet Details: Body	48
8.5	Snippet Details: Footer	49
9	Introduction to Markdown	51
9.1	Markdown Syntax	51
10	This is the largest heading	53
10.1	This is the second largest heading	53
10.2	Markdown and Stata	55
10.3	Markdown and Atom	55

Preface

This text is a companion document for **SOC 4650/5650 - Introduction to Geographic Information Sciences**. It is designed to help you be *successful* in this course. The idea behind a course **User's Guide** is to create a reference for many of the intangible, subtle or disparate skills and ideas that contribute to being a successful researcher. In creating a **User's Guide**, I draw inspiration from the work of Donald Knuth.¹ Knuth has discussed his experiences in designing new software languages, nothing that the developer of a new language

...must not only be the implementer and the first large-scale user; the designer should also write the first user manual... If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important...

While there is nothing particularly new about what I am writing here, and I am certainly not developing a new language for computing, the goal of the **User's Guide** remains similar to Knuth's experience. By distilling some of key elements for making a successful transition to being a *professional developer* of knowledge rather than a *casual consumer*, I hope to both improve the course experience itself and also create an environment that fosters a successful learning experience for you.

If you read through the course objectives included in the syllabus, you will note that creating maps is only one of them. As much as this is a GIS course, it is a course in research methods. In particular, we are concerned with *high quality* research methods and the *process* of conducting research. We therefore focus on a combination of mental habits and technical practices that make you a successful researcher. Some of the skills and techniques that we will discuss this semester are not taught as often in graduate programs. Instead, they are often the products of “learning the hard way”. These “habits of mind and habits of method” are broadly applicable across methodologies and disciplines.

License

Copyright © 2016-2017 Christopher G. Prener

This work is licensed under a Creative Commons Attribution 4.0 International License.

¹Donald Knuth is the developer of TeX, a computer typesetting system that is widely used today for scientific publishing in the form of LaTeX. He also established the concept of literate programming, which forms the basis of some of the practices we will follow with Stata this semester.

Chapter 1

Getting Started

Before you begin the semester, there are a number of things that I recommend that you do to help set yourself up for success. Before you do *anything* else, you should read through the **Syllabus** and the **Reading List**. Make sure you have a good sense of what is *required* for the course. If you have questions, bring them to the first day of class!

1.1 Prep Your Computer

Before you do anything else for this course, make sure you get your computer ready for the work you are about to undertake:

1. Make sure your operating system is up-to-date. If you are able, I would also recommend upgrading your computer to the most recent release of its operating system that the computer can run.
2. We'll be sharing computer files throughout the semester, so you should ensure that you have functioning anti-virus software and that it is up-to-date.
3. You'll also need to download files, so you'll need to make sure you have some free space on your hard drive. If you have less than 10GB of free space, you should de-clutter!
4. Make sure you know how to access your computer's file management system.
 - On macOS, this means being comfortable with Finder.app.
 - On Windows, this means being comfortable with Windows Explorer.

This of course assumes that you own a computer. Owning a computer is not required for this course. All students who are enrolled in SOC 4650 or SOC 5650 will be given 24-hour swipe access (*just what you always wanted!*) to Morrissey Hall to facilitate access to lab computers.

1.2 Create Accounts

There are two major web services that we will use this semester, and you'll need to create accounts for both:

- **GitHub** - you can sign-up at GitHub.com. Once you've signed up, fill out your profile, set-up two-factor authentication, and let Chris know (via email) what your user name is. Once he has it, he can add you to the SOC 4650/5650 organization.
- **Slack** - you can ask Chris (via email) for an invitation to sign-up for our team. Once the sign-up process is complete, you can log-in by going to our team's Slack site. Fill out your profile, set-up two-factor authentication, and change your timezone.

1.3 Download and Install Software

There are a number of software applications that we will use this semester. Most of them are free, and I recommend downloading those free ones right away. All of these applications are available for macOS and Windows.

- **Atom** - Atom is a flexible, open-source text editor that is produced by GitHub. You can download it from Atom's website.
- **GitHub Desktop** - GitHub makes a desktop client that you can use to easily interact with repositories that are stored on the site. You can download it from GitHub's website after you sign-up for an account there. You'll need that account information to complete the desktop client's set-up process.
- **Slack** - Slack has a number of applications for desktop and mobile operating systems. I recommend downloading Slack on your personal computer, and optionally installing it on your mobile device as well. You can download their desktop applications from their website and the mobile applications from your App Store.

For Graduate Students *only*

If your computer meets the operating system requirements for ArcGIS and you think you'd benefit from having access to the software at home, let Chris know (via email).

If you are in the Public and Social Policy Ph.D. program and your computer meets the hardware and software requirements for Stata, you should consider purchasing it for yourself. I recommend purchasing a perpetual license for Stata/IC. This is the most cost-effective solution for typical students.

1.4 Buy Course Materials

Books

There are three required books for this course:

1. Brewer, Cynthia. 2015. *Designing Better Maps: A Guide for GIS Users*. Redlands, CA: ESRI Press. ISBN-13: 978-1589484405; List Price: \$59.99; ebook versions available.
2. Gorr, Wilpen L. and Kristen S. Kurland. 2013. *GIS Tutorial 1: Basic Workbook*. 10.3.x edition. Redlands, CA: ESRI Press. ISBN-13: 978-1589484566; List Price: \$79.99; ebook versions available.
3. Thomas, Christopher and Nancy Humenik-Sappington. 2009. *GIS for Decision Support and Public Policy Making*. Redlands, CA: ESRI Press. ISBN-13: 978-1589482319; List Price: \$24.95.

There is one additional book that is optional:

- Mitchell, Michael N. 2010. *Data Management Using Stata: A Practical Handbook*. College Station, TX: Stata Press. ISBN-13: 978-1597180764; List Price: \$48.00.

Buying Mitchell (2010) is *highly* recommended for graduate students who will continue using Stata in the future and those who are concerned about the command-line interface. I recommend waiting for a week or two before purchasing this.

External Media

You will need a USB external storage device (either an external hard drive or a thumb-style drive) that has at least 20GB of storage capacity. This will be used for storing spatial data for this course.

1.5 Download Course Data

Mots of the course data is available for download via Dropbox in a single **.zip** file. If you want, you can let Chris know (via email) that you'd like to download these data before the beginning of the semester. Once you download them, extract the data from the **.zip** file and transfer them to your external storage device.

Chapter 2

Approaching this Course

Students have varying experiences learning GIS techniques. For some, the spatial logic and programming that are the foundation for GIS methods come naturally. For others, being introduced to these concepts can be an anxiety producing experience. I am fond the phrase “your mileage will vary” for describing these differences - no two students have the exact same experience taking a methods course.

2.1 Zen and the Art of Data Analysis

One of the biggest challenges with this course can be controlling the anxiety that comes along with learning new skills. ArcGIS processes, Markdown syntax, and Stata commands can seem like foreign alphabets at first. Debugging Stata do-files can be both challenging and a large time suck, in part because you are not yet fluent with this language. Imagine trying to proofread a document written in a language that you only know in a cursory way but where you must find minute inconsistencies like misplaced commas.

For this reason, I also think it is worth reminding you that many students in the social sciences struggle with quantitative methods at first. It is normal to find this challenging and frustrating. I find that students who can recognize when they are beginning to go around in circles are often the most successful at managing the issues that will certainly arise during this course. Recognizing the signs that you are starting to spin your wheels and taking either ten minutes, an hour or two, or a day away from GIS coursework is often a much better approach than trying to power through problems.

2.2 An Apple a Day

Being able to walk away from an assignment for a day requires excellent time management. If you are waiting until the night before or the day of an assignment’s due day to begin it, you give yourself little room for errors. I recommend approaching this course in bite size chunks - a little each day. The most successful students do not do all of their reading, homework, and studying in a single sitting. I find that this approach not only creates unnecessary anxiety around assignments, it also dramatically limits the amount of course material you can absorb. Keep in mind that I expect the *median* student to spend approximately six hours on work for this class each week (twice the amount of in-class time).

A sample approach to the class might look something like this:

- Tuesday: class
- Wednesday: finish lab
- Thursday: Start problem set
- Friday: Finish problem set

- Saturday: First reading
- Monday: Second reading

2.3 Reading with Purpose

The book and article **reading assignments** for this course are different from most of the other reading you will do in your graduate program because they are often very technical. Students who are most successful in this course read twice. Read the first time to expose yourself to the material, then take a break from the reading. During this first read, I don't recommend trying to complete the example problems or programming examples. Focus on the *big picture* - what are the concepts and ideas that these readings introduce?

During the second read, try to focus in in the *details* - what are the technical details behind the big picture concepts? I recommend doing this second read with your computer open. Follow along with the examples and execute as much of them as you can. By using this second read through as a way to test the waters and experiment with the week's content, you can come into the lecture better prepared to take full advantage of the class period. Students who follow this approach are able make important connections and focus on the essential details during lectures because it is their third time being exposed to the course material. They are also in a much stronger position to ask questions.

2.4 Active Lectures and Labs

During **lectures**, I introduce many of the same topics that your readings cover. This again is intentional - it gives you yet another exposure to concepts and techniques that are central to geospatial science. One mistake students sometimes make is focusing on the details of *how* to do a particular task rather than focusing on *when* a task should be done. If you know when a task is needed but cannot remember how to do it in Stata or ArcGIS, you can look this information up. Conversely, detailed notes on executing Stata commands may not be helpful if you are unsure when to use a particular skill. There is no penalty in this course for not knowing how to execute a command from memory; this is what reference materials are for. The most successful students will therefore focus on *when* a particular skill is warranted first before focusing on *how* to execute that skill

Getting experience with executing tasks is the purpose of the **lab exercises**. Time for beginning these exercises is given at the end of each class meeting, and replication files will be posted on GitHub for each lab.

2.5 Typefaces and Examples

2.5.1 Typefaces and Fonts

Technical publications that describe scientific computing processes use a **monospaced typewriter style typeface** to refer to commands (inputs) and results (outputs). In some documents, like lecture slides and cheat-sheets, I may highlight a command by using a to increase the visibility of the command name itself.

The **typewriter typeface** is also used to refer to filenames (e.g. `auto.dta`) or filepaths (e.g. `C:\Users\JSmith\Desktop`). Finally, we will use the **typewriter typeface** to refer to GitHub repositories (e.g. `Core-Documents`, the repository that contains this file).

Technical publications use *italicized text* to refer to text that is meant to be replaced. These references will typically appear in a **typewriter typeface** since they are often part of commands. For example, **describe varname** (with *varname italicized*) indicates that you should replace the text **varname** with the appropriate variable name from your dataset.

These publications also use a sans serif typeface to refer to areas of the user interface, menu items, and buttons. I cannot replicate that here because of the publishing software that I use, but you'll notice this text in course documents. We will therefore use the **typewriter typeface** in the User Guide to identify these same features.

Technical documents also use a sans serif or **typewriter** typeface to refer to keyboard keys (e.g. **Ctrl+C**) where the plus sign (+) indicates that you should press multiple keys at the same time. A sans serif typeface combined with a right facing triangle-style arrow (>) is used to refer to actions that require clicking through a hierarchy of menus or windows (e.g. **File > Save**).

2.5.2 Examples

Throughout the semester, I will give you examples both in lecture slides and in an example do-file. Examples in lectures and course documents can be easily identified by their use of the **typewriter typeface**:

```
. summarize mpg
```

Variable	Obs	Mean	Std. Dev.	Min	Max
-----+-----					
mpg	74	21.2973	5.785503	12	41

Examples will almost always use the file **census.dta**, which comes pre-installed with Stata. To open it, use the **sysuse** command: **sysuse census.dta, clear**. This allows you to easily recreate examples by minimizing dependencies within do-files.

Chapter 3

“Good Enough” Research Practices

This section introduces some of the core concepts that we will emphasize in this course throughout the semester. The title takes inspiration from a recent article titled “Good Enough Practices in Scientific Computing”¹. The authors note in their introduction that scientific computing advice can sometimes be both overwhelming and focused on tools that are inaccessible to many analysts. Their goal, and the goal of this course, is to de-mystify the simplest tools that enable researchers to streamline their workflows:

Our intended audience is researchers who are working alone or with a handful of collaborators on projects lasting a few days to a few months, and who are ready to move beyond emailing themselves a spreadsheet named `results-updated-3-revised.xlsx` at the end of the workday...Many of our recommendations are for the benefit of the collaborator every researcher cares about most: their future self.

I would argue that the skills they describe are useful beyond just a few months. Indeed, most of the skills here can dramatically improve students’ dissertation experiences:

Most importantly, these practices make researchers more productive individually by enabling them to get more done in less time and with less pain. They also accelerate research as a whole by making computational work (which increasingly means all work) more reproducible. But progress will not happen by itself. Universities and funding agencies need to support training for researchers in the use of these tools. Such investment will improve confidence in the results of computational work and allow us to make more rapid progress on important research questions.

While much of what we will talk about in this course is aimed at supporting your work, there are benefits that extend beyond your dissertation or your research projects. These benefits, which include developing sustainable workflows and structuring the way you interact with your own computer, can make everyday computing practices like checking email or organizing files an easier, more structured process.

3.1 Reproducibility

One of the mantras of this course is our emphasis on reproducibility. The unifying feature of all of the “good enough” research practices discussed below is that they contribute to a more reproducible research product.

Reproducibility is very much in vogue right now for number of reasons. Assessments of studies in psychology², for example, have found weaker on average effect sizes and far fewer statistically significant results than the initial studies reported. There have also been high profile instances of falsified research, including research

¹Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L. and Teal, T.K., 2016. Good Enough Practices in Scientific Computing. *arXiv preprint arXiv:1609.00037*.

²Open Science Collaboration, 2015. Estimating the reproducibility of psychological science. *Science*, 349(6251), p.aac4716.

by a graduate student at UCLA. This particular instance of fraud was identified by graduate students intent on replicating the original study.

At the same time, there is a recognition that the skills necessary for producing reproducible research are not being fostered in academic disciplines and graduate programs. Thus one of the goals of this course, and this **User’s Guide** in particular, is to help develop a working knowledge of many of these skills.

One challenge, however, is that reproducibility does not have a consistent definition. Some researchers use the term to narrowly refer to code that can execute without alteration on a person’s computer. Others use it to refer to research designs that can be replicated by other researchers. Still others discuss reproducibility as the ability to obtain a similar set of results or draw similar inferences from identical research designs.

When we talk about reproducibility in this class. We’ll be primarily concerned with **methods reproducibility**:

the ability to implement, as exactly as possible, the experimental and computational procedures, with the same data and tools, to obtain the same results.³

Methods reproducibility in GISc means that other analysts have full access to both the original data and the steps used to render those original data into a final research product, such as a set of maps. This is increasingly seen not just a matter of good research methodology, but as a matter of research ethics as well. Being able to be transparent with research decreases the potential for cases like the fraudulent dissertation research conducted by a UCLA graduate student named Michael LaCour. It was the efforts of two Stanford graduate students who wanted to reproduce LaCour’s findings that ultimately led to the identification of problematic work.

For GISc, methods reproducibility is derived from a number of sources. The first source is the use of **computer code** for working with data. Rather than making manual changes to tabular data in a spreadsheet application like Microsoft Excel, computer code provides detailed records of each individual alterations. Code can be used execute tasks repeatedly, meaning that errors can be easily fixed if they are discovered an hour, a day, a week, or a month later. During this semester, we’ll use Stata’s programming language to execute reproducible data cleaning processes.

Operations in ArcGIS can also be scripted using the programming language Python. Python is an open-source language that is widely used by data analysts and computer programmers. We will not learn ArcPy, the library of Python commands for ArcGIS, this semester. However, it is important to know that many of the things we will learn this semester *can* be scripted, dramatically increasing the reproducibility of your work.⁴

Since we won’t focus on scripting for ArcGIS this semester, much of the work we will do will be done manually. This means that no record exists of the changes we make or the steps that we take to complete a task. From a reproducibility standpoint, this is problematic. Even if we were scripting our work in ArcGIS, there are often aspects of projects that must be completed manually. In GISc, this often arises in initial steps like download data or in the production of final map products, which often require using graphic design software.

The second source of reproducibility in GISc is therefore derived from the **documentation** that we create to accompany our research products. These documents outline where our data originated (GIS metadata files), what specific variables mean (a codebook), what steps were taken to create specific maps (a research log), and how our data files are organized (a metadictionary).

Our code can also be used as documentation if it is written using literate programming techniques. In Stata, these techniques produce well annotated output that “weaves” together code, output, and narrative text that describes the function of the code and the results of the output.

The third and final primary source of reproducibility in GISc is derived from our **organizational approach** to our work. GISc projects can require many gigabytes of data spread across dozens or even hundreds of

³Goodman, S.N., Fanelli, D. and Ioannidis, J.P., 2016. What does research reproducibility mean?. *Science translational medicine*, 8(341), pp.341ps12-341ps12.

⁴For those of you who are interested, we’ll be providing Python/ArcPy examples for many of the ArcGIS tasks we learn this semester. These will be available on GitHub in the **ArcPy** repository for those of you interested in expanding your knowledge.

files, feature classes, and databases. A disorganized file system can make replicating your work difficult if not impossible. Much of the research practices discussed in the remainder of this section are aimed at supporting one or more of these three major sources of reproducibility.

3.2 Thinking in Workflows

One way to increase the reproducibility of a project is to approach each and every task with purposeful organization and thoughtfulness. **Workflows** are the processes that we use to approach a given task. Think of checking your email. You (hopefully!) follow a series of steps when you check your email that help you organize your inbox. When I check my email for the first time each day, my workflow looks something like this:

1. Delete junk mail
2. Read and then delete New York Times and Washington Post morning newsletters
3. Read and then delete SLU Newslink newsletter
4. For each remaining email:
 - a. Respond if response will take less than two minutes and/or
 - b. forward to task management inbox if email requires an action, or
 - c. snooze⁵ the message until “later today” or “tomorrow morning” if response will require more time than currently available.

In our reading for the first week of classes, Scott Long⁶ describes a structured strategy for approaching statistical research. In Long’s model, a data analysis project consists of four steps: (a) data cleaning, (b) analysis, (c) presenting results, and (d) protecting files. This is a useful model to build upon for GISc work, and one that we will discuss over the course of the semester.

Even more useful, not just for GISc work but for any process, are the tasks Long lays out for each step in the data analysis workflow:

1. Planning
2. Organization
3. Documentation
4. Execution

A good example of the utility of extending this logic to other workflows is with the problem sets. The “typical” approach students take with homework assignments is to sit down, open up their software, and start with question 1. Using Long’s four task approach, a workflow-based strategy to the assignment would involve beginning by reading the assignment through in its entirety to develop a **plan** for approaching it - think about what techniques and skills are needed for each step. With a plan in place, you can proceed to **organizing** yourself for the assignment - identifying and obtaining files that you will need, creating dedicated directories for saving assignment data, and getting any necessary software documentation. After pulling together all of these materials, you are ready to move on to **documentation** - setting up your assignment code and output files, and (later in the course) your research log and meta-dictionary. Once you are set-up, you would then begin to address individual assignment questions as part of the **execution** task.

The goal here is to approach everything you do for research or work with an element of mindfulness and structure about your process. This mental model for approaching research supports the creation of **reproducible** research products because we approach our work in a routinized, predictable, organized, and efficient manner. Thinking in terms of workflows also encourages a greater awareness of the complexity of tasks, which also helps you plan more accurately for how long a particular task or project will take.

In reality, there will be multiple workflows that you find yourself navigating. You will want a structured process not just for approaching a large spatial research project like the final project, but also a process for maintaining notes related to a specific assignment, a process for documenting code, a process for approaching

⁵Snoozing is a “magical” feature of the email client that I use - SparkMail.

⁶Long, J.S., 2009. *The workflow of data analysis using Stata*. College Station, TX: Stata Press.

assignments, and even a process for backing your data up. As you go through the course, think about how to best integrate these ideas into your work habits.

3.3 Course Tools

This course relies on a number of major tools to help us accomplish the work that we need to do. This makes for a complex learning curve, particularly at the beginning of the semester. The tools we’ve selected for this class have been picked not necessarily because they are the *easiest* tools to learn, but because they *increase* our ability to conduct reproducible research.

3.3.1 ArcGIS

ArcGIS is the industry standard GIS application suite. Though there are other tools out there that contain much of the same functionality, ArcGIS remains the expected skill-set for entry-level GIS jobs in nearly every sector of the labor market. ArcGIS excels at managing and manipulating spatial data, and has a wide range of tools for creating data visualizations that use spatial data. For these reasons, ArcGIS will occupy a large portion of our time this semester.

ArcGIS has two weaknesses for our purposes. While it can be scripted using Python, that functionality is not placed front and center in the application. It also is difficult to pick up ArcGIS’s Python tools, named ArcPy, without some background in GIS more generally. This limits the longterm reproducibility of the GIS work we’ll do this semester since it is driven by the difficult to reproduce point-and-click user interface.

The second weakness that ArcGIS has is a limited set of tools for cleaning and manipulating tabular data. Not only are these driven by a point-and-click user interface, and thus limited in their ability to achieve reproducibility, but they are cumbersome and lack the power of other approaches to cleaning data.

3.3.2 Stata

Since ArcGIS is limited in its approach to data cleaning, we will use Stata instead. Stata is, first and foremost, statistical software. It has its own programming language and syntax that can be used not only for statistical purposes but also for cleaning data. For tabular data, therefore, Stata will become an intermediary tool between raw data and data that is suitable for mapping.

Our approach to using Stata will involve using a technique I described above called literate programming. The implementation of literate programming in Stata comes from a user-written package called `**Markdoc`. Markdoc allows you to embed text that is formatted using the markup language Markdown. Markdown is increasingly being adopted as one of the primary data science writing tools because it is (a) simple, (b) does not require extensive software or plugins, and (c) is widely supported by applications like **R** and GitHub. We’ll describe both Markdoc and Markdown further in the chapter “Reproducible Do-files”.

3.3.3 Atom

While Stata does have a built-in do-file editor, and you could easily use it to author code for Stata, writing in an external text editor has a number of advantages. Atom is a free, highly extensible, and easy to use text editor. Unlike Stata’s do-file editor, it is not tied to a single application or programming language. And unlike Stata’s editor, it cannot be readily extended, customized, or used for work outside of Stata.

Atom, on the other hand, offers a large number of user-written packages that dramatically extend its base capabilities. One of those, `language-stata`, gives Atom support for working with Stata’s do-file format. Atom also offers a text expansion tool that will help you write consistently structured and documented do-files. As we progress through the semester, we’ll also use a number of packages for writing and previewing

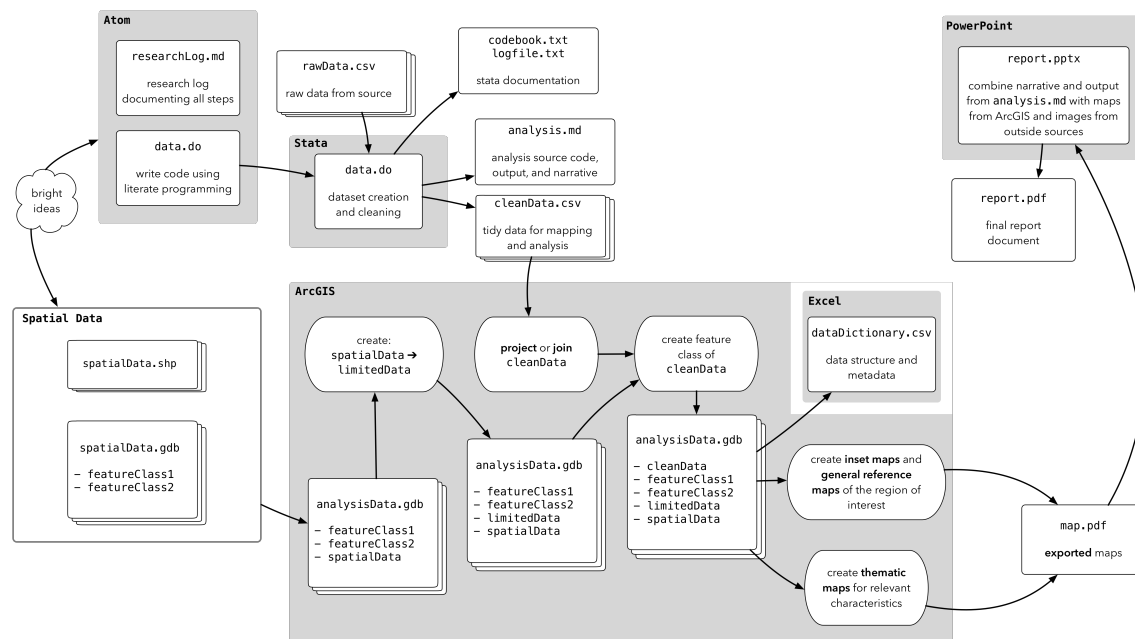
Markdown text files. So, while using Atom means adding an additional tool to your workflow, it also offers a number of improvements over what Stata comes built in with. We'll describe all of this further in the chapters on "Introducing Atom", "Introducing Markdown", and "Reproducible Do-files".

3.3.4 GitHub

The final tool we'll use, GitHub and its desktop application GitHub Desktop, is an exceptionally powerful tool for conducting version control on an entire directory. This allows you to track changes in individual files as well as changes that impact an entire sub-folder or entire project directory. GitHub is increasingly recognized as one of the key tools available for making research reproducible because it allows users to maintain logs of every change they make on a project. It also offers other tools that support project management, including to-do lists, issue tracking, and even website maintenance. Since GitHub provides support for the GeoJSON standard for storing spatial data, we can store and preview(!) shapefiles on GitHub as well. GitHub, and the software that powers it called Git, are both described further in the "Introduction to GitHub".

3.4 Course Workflow

One of the largest learning curves with this course is keeping track of how all of these tools fit together. This process is described in detail in Week 2's lecture, but what follows is a short description of the "big picture" at work here. As we said above, it is important for you to be thinking in workflows. We've mapped out the major aspects of our course workflow to help aid that process:



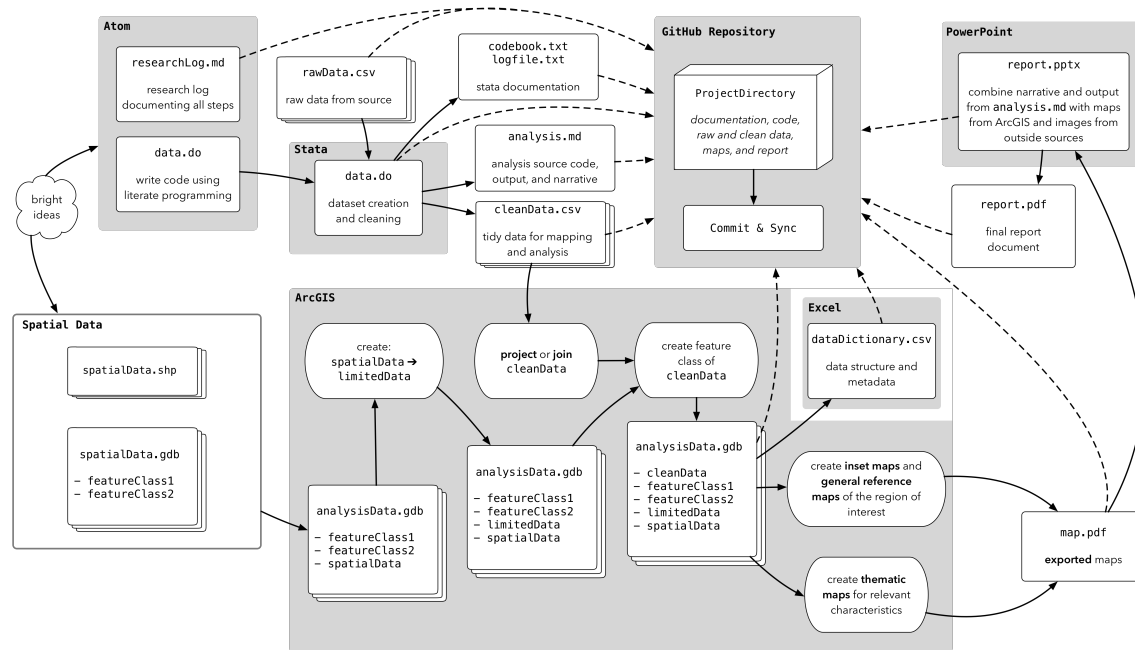
This workflow is premised on a common GISc situation: you have data stored in some type of database in a **tabular** or spreadsheet-like format that have a **spatial reference** like an address, which would allow them to be mapped. This main dataset of interest needs cleaning, as real world data often do, before it can be mapped. We'll use code written in Atom and executed by Stata to accomplish this task. We'll also use Atom to maintain and edit documentation that helps you increase the reproducibility of your work.

Once the data are cleaned, we'll want to start working on mapping them. This cannot occur in a vacuum. Rather, you will need to seek out data sources that describe the physical or human geography in the area of interest. These may come as **shapefiles** or **geodatabases**, and they may also require some sort of data

cleaning. Often the spatial data we have access to cover a larger area than what we need, or they cover too small an area and have to be merged with other files to capture the extent we require.

Once both our tabular and spatial data are cleaned, we can bring our tabular data in ArcGIS so that we can further clean it, if necessary, and map it. When we have maps ready for export, we often combine them into deliverables like presentations or printed booklets. This is best accomplished outside of ArcGIS in an application like PowerPoint or Word, or a more advanced publishing tool.

Finally, as we noted above, we will capture and track *almost* all of these files using GitHub:



This workflow captures every aspect of a project from the bright idea that launches it to data acquisition, data cleaning, mapping, dissemination, and archiving.

Chapter 4

Protecting Your Work

Each semester that I teach this course or SOC 5050 (Quantitative Analysis), two things happen. The first thing that happens is that students regularly lose files. The effects of losing files can range from being a minor frustration to a major headache depending on the file in question. Losing files often results in downloading multiple copies of the same data and recreating work. Both of these are wastes of your time. Moreover, files are rarely gone. They are typically just misplaced. This is bad for reproducibility, particularly when you happen across multiple versions of the same file and have to sort out which version is the version you last worked on.

The second thing that happens is that students lose their thumb drives. Depending on the timing of this loss, this can again range from being a minor frustration (very early in the semester) to being downright anxiety attack producing (last few weeks of the semester). Recreating an entire semester's worth of work on the final project is both a tremendous waste of your time and a particularly unpleasant experience.

Fortunately, I have never had a student's computer hard drive die during the course of the semester. However, I assume that if I teach this course long enough a hard drive failure will indeed occur. The backup provider Backblaze has analyzed their own hard drives and found that about 5% of drives fail within the first year. After four years, a quarter (25%) of drives in their data center fail.

Similarly, it is only a matter of time before a student's computer is stolen along with all of their hard work. A less likely though still very plausible scenario involves the destruction of a student's belongings (computer and thumb drive included) in a fire, car accident, or natural disaster.

Despite the likelihood that you will at some-point lose a thumb drive (if not during this semester than sometime down the road) and the near certainty that your computer's hard drive will eventually fail if a rogue wave does not get it first, few students and faculty take these risks seriously. While you cannot prevent many of these things from happening, I want to suggest to you that you can take some simple steps to sure that *when* (not if) they happen, you are well prepared to get back to work with minimal disruption.

4.1 Data Management

One of the themes in “Good Enough Practices in Scientific Computing”, referenced in the previous chapter, is an emphasis on data management. One of their core messages is to “save the raw data”. In GISc work, the raw data can be expansive - dozens of shapefiles, tabular files, and associated metadata. These files often come from disparate sources - city open data sites, the U.S. Census Bureau, state data repositories, and other federal agencies. Moreover, GIS data are often updated over time to reflect on-the-ground changes. Saving the raw data in GISc work therefore means not only creating a well-organized directory containing *all* of your original data. It also means logging the source of each file, when it was downloaded, and (if

applicable) a permanent web link to your data source. For that reason, we'll give you not just the course data but a read me file and a metadictionary that lists all of the files we've disseminated to you.

A second message in the paper is to “create the data you wish to see in the world”. The authors encourage readers to “create the dataset you wish you had received.” First and foremost, this means using open and not proprietary data formats. For spatial data, ESRI shapefiles are technically proprietary, though their standard is open. This means that other software applications, like R, QGIS, and even Stata can read and in some cases write shapefiles. For sharing spatial data, a better option is the GeoJSON, which is a plain text file format.

Tabular data are best stored as CSV files, which is also a plain text file format that can be opened by a wide variety of applications. In contrast, common file formats like Microsoft Excel's XLS and XLSX are proprietary file packages that cannot be read as plain text and are therefore less desirable for storing data.

Both tabular and spatial data, in their final forms, should be what we consider “tidy data”¹ Tidy data are defined by a number of common attributes - each column represents a single variable or attribute and each row represents a single, unique observation. This arrangement should produce clear, easy to read datasets that represent a single observational unit.

Tidy datasets also have other characteristics. Variable names should be short, clear, and self-explanatory (i.e. `streetAddress` and `zipCode` are preferable to `add1` and `add2`). Missing data should be properly declared in a machine-readable format instead of using a code like `-1` or `9999`. Filenames should also be clear and self-explanatory (i.e. `stlouisHomes_011717.csv` is preferable to `final.csv`).

4.2 Creating a Sustainable File System

In his excellent document *The Plain Person's Guide to Plain Text Social Science*, Kieran Healy describes two important revolutions in computing that are currently taking place. One of them is the advent of mobile touch-screen devices, which he notes

hide from the user both the workings of the operating system and (especially) the structure of the file system where items are stored and moved around.

For most users, I would argue that this extends to their laptop or desktop computers as well. I would venture to guess that the majority of my students are used to keeping large numbers of files on their desktops or in an (distressingly) disorganized **Documents** folder.

For research, particularly quantitative research, such an approach to file management is unsustainable. It is difficult to produce *any* research, let alone work that is reproducible, without an active approach to file management.

4.2.1 Create a *Single* Course Directory

The most successful approach to organizing files is to identify *one and only one* area that you will store course files in. Having files scattered around your hard drive between your **Desktop** directory, **Downloads**, **Documents**, and a half dozen other places is a recipe for lost files. It can also add complexity to the task of backing these files up. I recommend naming this directory simply `SOC4650` or `SOC5650`. This is short, has no punctuation or spaces (which can create conflicts with software), and explicitly connects the directory to this course as opposed to other courses you may take that are also GIS courses (a good reason to avoid naming the directory **GIS**!).

¹Wickham, H., 2014. Tidy Data. *Journal of Statistical Software*, 59(i10).

4.2.2 Approach Organizing Systematically

Within your single course directory, I recommend following much of Long’s (2009) advice on organization. Approach this task systematically and mindfully. This approach begins with having a number of dedicated subfolders within your course directory:

```
/SOC5650
  /Core-Documents
  /Data
  /DoeAssignments
  /FinalProject
  /Labs
  /Lectures
  /Notes
  /ProblemSets
  /Readings
  /Software
  /WeeklyRepos
```

Note again how these directories are named - there are no spaces, special characters, and the names are deliberately short but specific. For a directory with two words (**FinalProject** or **ProblemSets**), I use what is known as camelCase to name the file where the second (any any subsequent) words have their first character capitalized. You could also use dash-case (**Core-Documents**) or snake_case (**Core_Documents**) as a naming strategy. Regardless of which of these approaches you take, try to use it consistently.

The course data release is embedded in an otherwise empty folder structure that mirrors this layout. When you download these data and the accompanying directories, un-zip them and move the entire contents to the root of your thumb drive or external hard drive. If you are registered for SOC 4650 and want your directory to match your registration, feel free to rename it **SOC4650**.

4.2.3 The Core-Documents Directory

This directory will *not* be included in the folder structure that you download along with the course data release. This directory will be added to your file system during **Lab-03**, when it is **cloned** from GitHub. A cloned directory is one that retains a digital link to the data stored on GitHub, meaning that it can be easily updated if changes are made. This will be explained in greater depth in the next chapter of the User’s Guide. **Do not edit the files in these repositories.**

4.2.4 The Data Directory

The data directory should have copies of all original data and their documentation. Most of these data are included in the initial data release, but you will have to add some additional data to this directory over the course of the semester. The data in this directory should be used as needed but not altered (one of the of the “good enough” research practices from the previous chapter).

4.2.5 The DoeAssignments Directory

Like the **Core-Documents** repository, this will not be included in the course data release. You will add it to your file system during **Lab-03**. It will also have a different name - your last name instead of ‘Doe’. Once you add it, it will contain a number of subdirectories:

```
/SOC5650
  /DoeAssignments
```

```

/FinalProject
  /Documentation
  /Memo
  /PosterDraft
  /PosterFinal

/Labs
  /Lab-01
  ...
  /Lab-16

/ProblemSets
  /PS-01
  ...
  /PS-10

```

The `FinalProject` directory contains submission folders for each component of the final project. If you are registered for SOC 4650, your directory will look like what appears above. Students registered for SOC 5650 will have three additional subfolders for deliverables related to the final paper element of the course.

The `Labs` and `ProblemSets` directories have subfolders dedicated to the 26 individual assignments you'll have to submit over the course of the semester. **These directories are intended to store only the deliverables that are requested in each assignment's directions.** All other files related to each assignment should be stored elsewhere in your folder structure.

4.2.6 The FinalProject Directory

The final project directory should be a microcosm of the larger directory structure, with most major directories replicated so that your final project files have a dedicated, organized home:

```

/SOC5650
  /FinalProject
    /Data
    /DataAnalysis
    /Documentation
    /Memo
    /Notes
    /Poster
    /Readings

```

You'll notice that there are a number of new directories dedicated to specific aspects of the project.

SOC 5650 students: you will want to add directories for the `/AnnotatedBib` and `/Paper` aspects of the assignment. I also recommend using some type of bibliography software. (Endnote, for example, can be obtained for free by SLU students). Whatever application you choose, keep its primary database for your project in the `Readings` folder along with copies of all `.pdf` readings.

4.2.7 The Labs Directory

This directory contains subfolders for each of the sixteen lab assignments for this course. Save *all* of the associated materials for each lab assignment here, including text files, documentation, map files and output, data tables, and any new data that you are asked to create and save.

4.2.8 The Lectures Directory

This directory contains subfolders for each of the sixteen weeks of the course. When we create new data files, make maps, or write code during lectures, save these documents in the appropriate week's folder.

4.2.9 The Notes Directory

Use this as a home for course notes.

4.2.10 The ProblemSets Directory

This directory contains subfolders for each of the ten problem set assignments for this course. Save *all* of the associated materials for each problem set here, including text files, documentation, map files and output, data tables, and any new data that you are asked to create and save.

4.2.11 The Readings Directory

Use this as a home for .pdf copies of course readings.

4.2.12 The WeeklyRepos Directory

Clone each of the weekly repos to this directory, and sync them when updates are made to ensure you have the latest versions of files.

Do not edit the files in these repositories. If you want or need to work with them, make a copy and save it into the relevant assignment directory.

4.3 Backing Up Your Data

There are a number of different ways to think about backing up your data. The most successful backup strategies will incorporate all of these elements.

4.3.1 Bootable Backups

“Bootable” backups are mirrored images of your *entire* hard drive, down to temporary files, icons, and system files. With a bootable backup, you can restore your entire computer in the event of a hard drive failure or a corruption of the operating system files. They are named as such because you can plug in the external drive that you are using for this backup and literally boot your computer up from that drive (typically a *very* slow process).

These backups are often made less frequently because they can be resource intensive and it is best not to use your operating system while creating a clone. They are typically made to an external hard drive, which is subject to similar failure rates as the hard drives inside your computer. So bootable drives need to be replaced every few years to maintain their reliability.

Both major operating systems come with applications for creating clones of your main hard drive that are bootable, and there are a number of third party applications that provide this service as well.

4.3.2 Incremental Backups

Incremental backups are designed to keep multiple copies of a single file (how often depends on the type of software you use and the settings you select). These can be used to restore an older copy of a file if work is lost or a newer file is corrupted.

Apple's TimeMachine is a great example of an incremental backup - when kept on, it creates hourly backups of files that have been changed, daily backups for the previous month, and weekly backups for previous months. Once the disk is full, the oldest backups are deleted. Dropbox also provides a similar service, retaining all previous versions of files (and deleted files) for thirty days.

Incremental backups are typically good options for recovering files that have been recently changed (again, depending on the software you use and the settings you select). Since they run frequently (every time a file is changed or every hour, for example), recent changes tend to get captured. They can be limited in terms of their long-term storage - it may not be possible to recover older versions of a file past a few weeks.

They are also not always good solutions for recreating your entire computer since they do not save all necessary program and operating system files, and may be cumbersome to work with if you need to recover a large quantity of files. Like bootable backups, these are typically stored on external hard drives that need to be replaced on a regular basis.

In addition to the aforementioned Apple TimeMachine, the Windows OS also comes with a built-in service for creating incremental backups. Dropbox is a good option if you have a small number of files, but you may find the need to upgrade to a paid account if you have a large amount of data.

4.3.3 Cloud Backups

Cloud backup services like Backblaze or Crashplan offer comprehensive backup solutions for customers. These plans typically require a monthly subscription fee to maintain access to your backups. While bootable backups protect against hard drive failure and incremental backups protect against data corruption, cloud backups protect against catastrophic events like robberies, fires, and other natural disasters. A fire or a tornado that affect your house may destroy your laptop and any external hard drives you use for backup, but your cloud backup will be unaffected.

4.3.4 A Workflow for Backups

Just as we need a workflow for approaching file management, it is also important to establish a routine for backups. With backups, the most successful workflows are those that require next to no effort on your part. If you primarily use a desktop, this can be as simple as leaving two external hard drives plugged into your computer since most backup software can be set to run automatically. If you have tasks that require you to manually do something (plug an external hard drive into your computer, for instance), create a reminder for yourself on a paper calendar or a digital calendar or to-do list application.

For this course in particular, it is *imperative* that you backup the data on your flash drive. A number of possibilities exist for accomplishing this:

- Keep a local copy of your flash drive's files on your computer.
- Keep a `.zip` archive of your files in a service like Dropbox or Google Drive. (Using a `.zip` archive will prevent issues with your `.git` repositories.)
- Maintain a second flash drive copies of all of your files.

Whatever solution you select, make sure you regularly update your backup. The more often you keep your backup archive updated, the less stressful and disruptive losing your drive will be. This will likely be a manual task, so follow the guidance above about creating a repeating calendar event or to-do list task reminder.

Chapter 5

Introduction to GitHub

Much of our interaction this semester outside of class will utilize GitHub.com (or just “GitHub”). GitHub is a web service that is a social network for programmers, developers, data scientists, researchers, and academics. It is also a tool for collaborating on projects, especially projects that involve writing code.

We’ll use GitHub as an alternative to Blackboard, the *course management system* that students are typically familiar with. Course materials will be posted there, and GitHub’s features will allow you to copy them and keep them updated as changes are made. You’ll also use GitHub to submit assignments for grading, and we’ll give you feedback and grades via GitHub as well.

5.1 Git

GitHub is a web application that utilizes Git:

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Essentially, Git is a project-wide system for tracking changes to files. Think of it as Microsoft office’s track changes feature on steroids - every change to every file in a directory (a “repository” or “repo” in Git-lingo) is tracked. You do not need to host files online to use Git. If you have a project saved locally (say, a doctoral thesis), you could utilize Git to version control that project without ever uploading it to the Internet.

For our purposes, this is just about all you need to know about Git. If you want to learn more, Git’s ‘About’ page is a great place to start.

5.2 More Git-lingo

Beyond “repositories”, there are a few additional terms that are specific to Git and that are helpful to know:

- **Clone:** Make an identical copy of a repository on your local hard drive.
- **Commit:** Approve any changes you have made to a repository.
- **Sync:** For cloned repositories, files that have been changed need to be synced or **pushed** to GitHub.com after they are committed.

5.3 GitHub.com

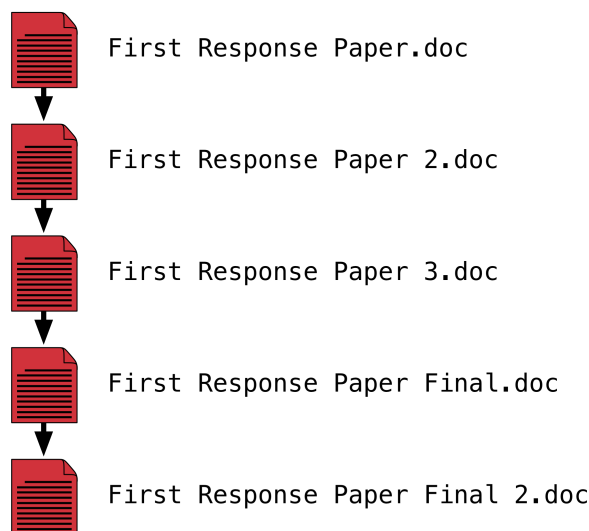
GitHub is a web service that can host projects using Git’s version tracking. It is widely used by programmers, software developers, data scientists, and academics to host and collaborate projects.

GitHub is an excellent way to backup files for a project since you can “sync” changes made to a repository up to GitHub’s servers. It is also an excellent way to collaborate on files with colleagues while also using Git’s version tracking. Repositories can be either public (like all of the repos for our seminar) or private, which means that only people who have been given access to can view the contents of the repo. Private repos require an upgraded account, which retails for \$7/month.

Students can get access to GitHub’s paid services for free, however, by signing up for a free student account. This will give you access to private repositories for as long as you are a student.

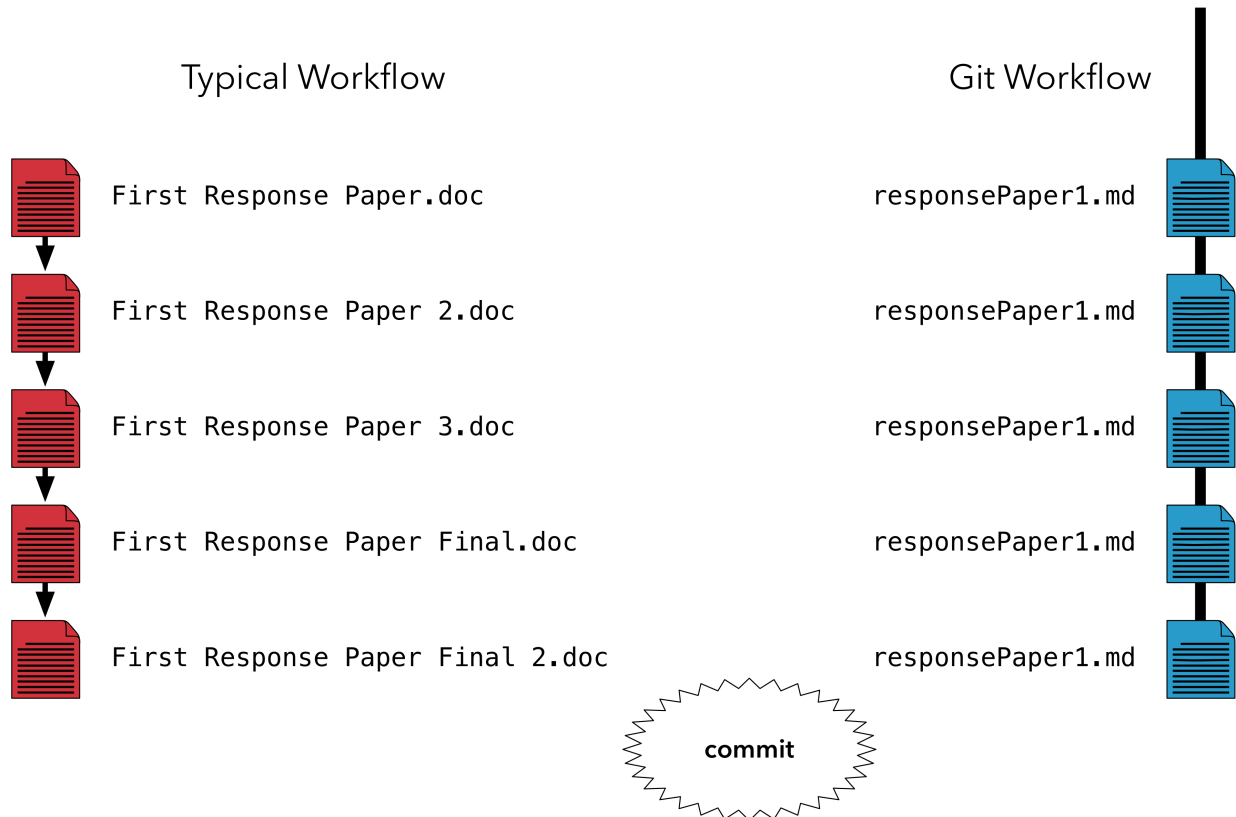
5.4 The Workflow of Git and GitHub

The typical approach to versioning for many students is manual. For a hypothetical class response paper, it might look something like this:

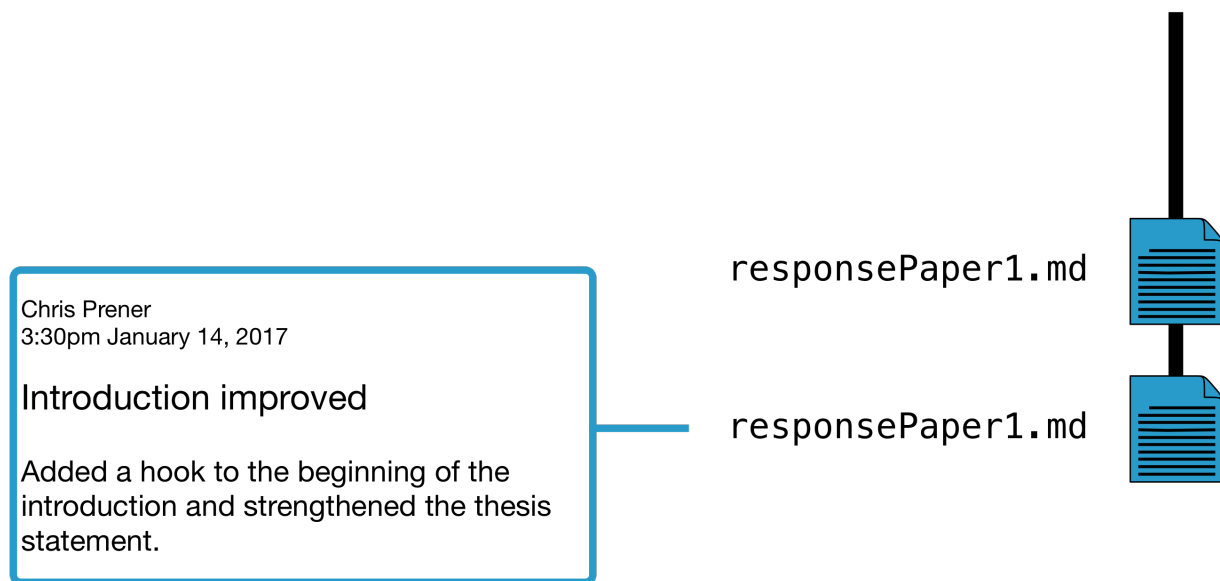


The author made an initial copy of the paper, and then used a haphazard and inconsistent approach for naming subsequent copies of the paper. We can presume that changes were made in a linear fashion, though it is easy to make changes to, say, `First Response Paper 2.doc` after `First Response Paper 3.doc` has been created and edited.

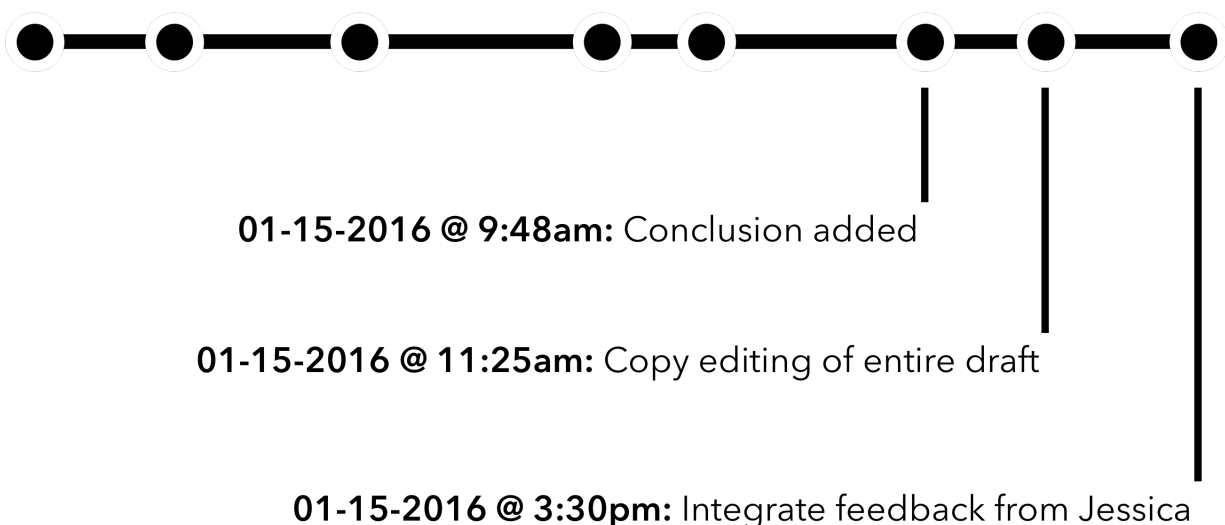
Instead of saving copies of their hypothetical paper, a student using GitHub could write the paper in a single document, **committing** their changes as they progress to take “snapshots” of their progress. These snapshots contain information on changes the student has made, tracked line-by-line. So, at each point in which a new document would have been created in the typical workflow, a student using GitHub would simply **commit** their changes:



Git provides a number of useful features beyond simply tracking changes. Each commit is accompanied a **message**. These messages must have a short summary that appears on GitHub and can also have a longer description that can be used to describe in detail what changes are being applied with a specific commit:



Messages, combined with the changes that are tracked, allow users to trace the development of a single document or an entire project overtime:



This means that, if necessary, the project can also be rolled back to an earlier period. Finally, users can **sync** their commits with GitHub.com, hosting their changes and their data in a way that protects them against certain types of computer failures and also allowing them to easily share their work with others.

5.5 GitHub Repositories

Users of GitHub.com adhere to a couple of norms with their repositories that are worth knowing about. Repositories cannot have spaces in their names (much like variables in Stata), so the naming conventions that we will discuss in relation to Stata this semester all apply to GitHub as well!

Public GitHub repositories also contain (typically) at least three core files:

1. A **license** file - since the data is out there for public consumption, it is important to think about how that data is licensed. The norm among GitHub users has been to use open source licenses, which let others edit and adapt your work. There are a range of licenses that are commonly used on GitHub.
2. A **README** file - this describes the purpose and content of the project.
3. A **.gitignore** file - this stops certain types of files from being swept up by GitHub when a user syncs their files with a server.

5.6 Storing GitHub Repositories

When you clone your repositories, you will be prompted to save them on your computer. There are a number of ways in which this process can introduce sources for trouble down the road. The principle way that I have seen students run into problems with GitHub is by storing repositories on cloud storage services like Dropbox or Google Drive. In order to avoid any issues, I advise against storing GitHub repositories in an area of your computer that syncs with a cloud service.

5.7 GitHub Issues

GitHub has a powerful tool for interaction called Issues. These can be accessed by opening a repository and then clicking on the “Issues” tab. Issues can be “opened” by anyone with access to the repository. They

allow for a conversation to occur in the form of messages posted within the Issue itself. Files can be attached to Issues, and the messages can contain Markdown formatting. Once the conversation is complete, issues can be marked as “closed”, which moves them into a secondary view on the website so that they are archived.

We’ll use issues for both assignment feedback and grading. Please keep up with issues as they appear, and feel free to follow-up with specific questions about your grade or the assignment feedback in the Issue conversation. Once you are satisfied, please mark the issue as closed.

5.8 GitHub Desktop Application

GitHub Desktop is a tool that allows you to easily clone repositories hosted on GitHub, commit changes to them, and then sync those changes up to the website. You can also create new repositories, however this is not a task you will have to do this semester. GitHub Desktop is not a fully functional desktop version of GitHub. For our purposes, it is important to note that the Desktop application will not let you easily identify when repositories have been updated by other users, view Wikis associated with repositories, or view Issues.

5.9 Learning More

GitHub has a resources page with links to websites that are great for helping you learn more about how Git and GitHub work! The next chapter also has some additional GitHub and Git information.

One particularly great tutorial walks you through the command line process for creating and using a git repository. Even if you do not want to use Git via the command line, the tutorial does an *excellent* job of describing the logic and sequence behind the Git workflow.

Chapter 6

Advanced GitHub

GitHub has a number of tools available for managing multiple contributors' work simultaneously. These tools will be useful for collaborating with your teammates on the final project.

6.1 Even More Git-lingo

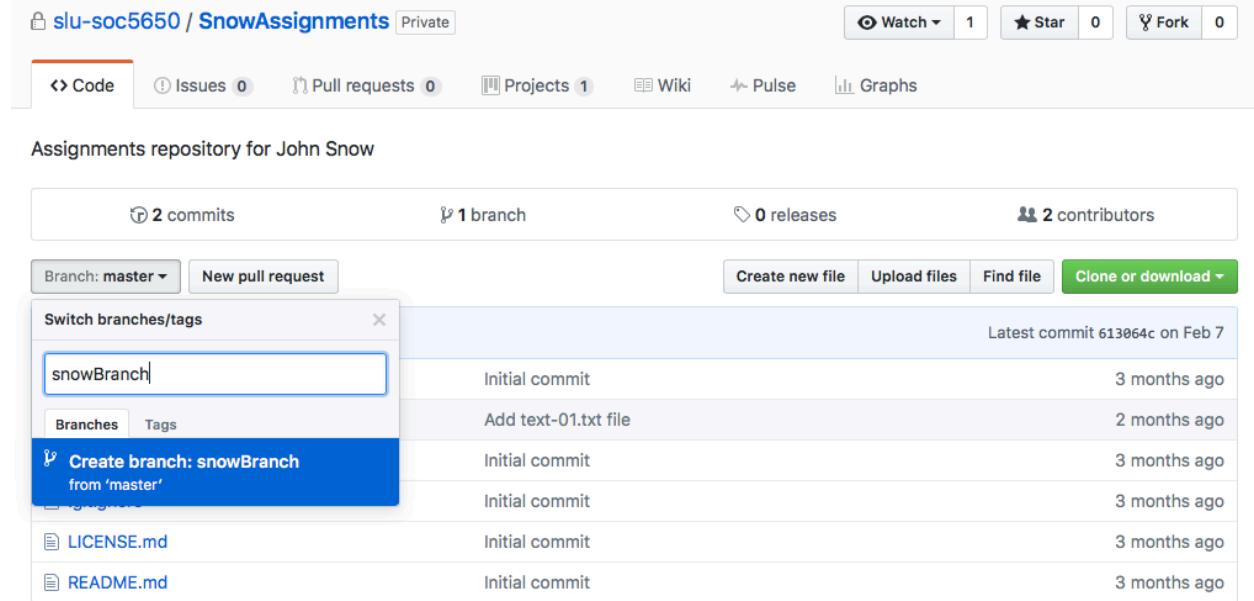
Beyond the terms introduced in the previous chapter, there are a few additional terms that are specific to collaborating that are helpful to know:

- **Branch:** An identical copy of the repository that is distinct from the `master` copy.
- **Checkout:** The act of switching to a new branch.
- **Pull Request:** Request that the changes made on a branch be integrated into the `master` copy.

6.2 Creating Branches

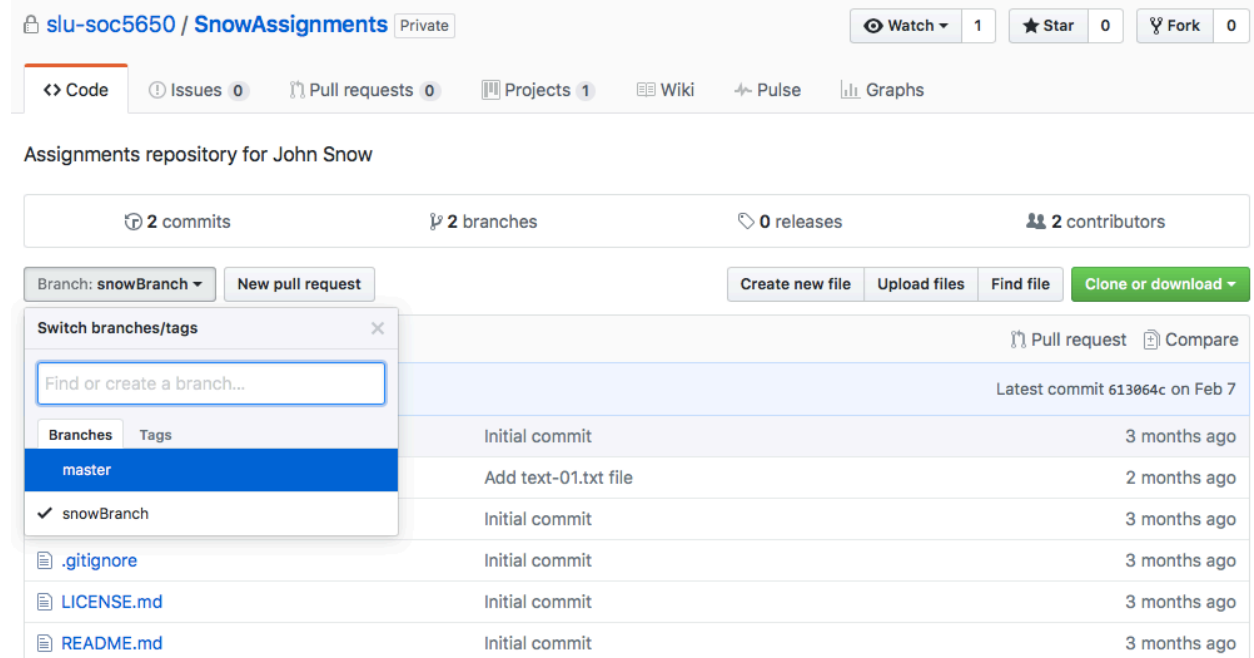
The logic behind branches is that they allow you to make modifications to the content of the repository without impacting the content that is on the `master` branch. Think of the `master` branch as your production data - this is what you know is good and in working order. If you need to experiment, say by adding new code or data sources, you can do this on a branch without worrying about breaking something in the production part of the repository.

To create a new branch, navigate to the repository in question on GitHub.com. Just above the list of the repository's contents on the lefthand side of the window is a button that will read "Branch: master". If you pull the associated menu down, you have the ability to create a new branch (by typing a name into the text field) or switch between branches:



The image above shows this menu pulled down with the fictitious user John Snow creating a new branch on his assignments repository named **snowBranch**. Once the branch is created, you will automatically be directed to the new branch's contents. These should look identical to the **master** branch at this point because the branch creation process generates a mirror image of **master**.

You can use the same pulldown menu to switch back to the **master** branch if needed:

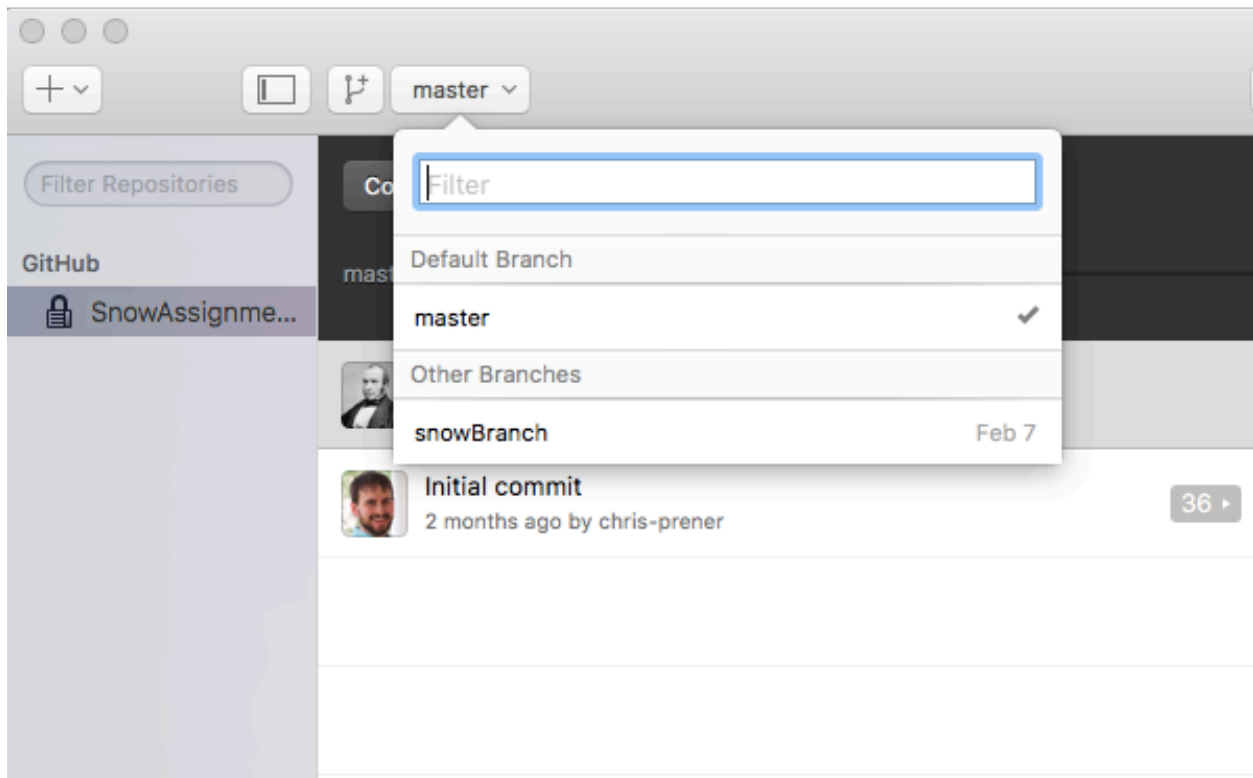


6.2.1 Working with Branches

Once your branch is created, you need to open up GitHub desktop and, if you haven't already, clone the repository you created the branch in. If you have already cloned the repository, you just need to **Sync** it so

that the branch information is downloaded to your local copy of the repository.

Once you have updated or cloned your repository, it is time to **checkout** the branch. This means to switch from the current branch (by default, the **master** branch, to your new branch). In GitHub Desktop, there is a button on the top toolbar that will say the current branch name (by default, **master**). You can click on this button to open a pulldown menu that will list all available branches:



Click on a branch to check it out. Once checked out, the files available to your operating system will change to reflect the contents of that branch!

It is important that you ensure that you are always intentionally working on the correct branch. Any time you return to your computer to continue working on a branch, open up GitHub Desktop and ensure that the proper branch remains checked out.

Branches are used just like regular repos in Git. You can commit changes to them and sync them with GitHub.com. I recommend committing changes to branches regularly and syncing them with the remote version of the repository to ensure that your work is not lost.

6.2.2 Opening Pull Requests

When you have made changes to your branch and want to integrate them into the **master**, production part of your repository, it is time to open a **pull request**. Below, you will see that a directory named **ExtraWork** has been added to the **SnowAssignments** repository's **snowBranch**:

The screenshot shows the GitHub repository page for 'slu-soc5650 / SnowAssignments'. The repository is private and has 1 Watch, 0 Stars, and 0 Forks. The main navigation bar includes links for Code, Issues (0), Pull requests (0), Projects (1), Wiki, Pulse, and Graphs. Below the navigation bar, the repository is described as 'Assignments repository for John Snow'. It shows 3 commits, 2 branches, 0 releases, and 2 contributors. A branch selector shows 'snowBranch' with a 'New pull request' button. Action buttons include 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. A message states 'This branch is 1 commit ahead of master.' with links for 'Pull request' and 'Compare'. A commit by 'chris-prener' is shown with the message 'Add extra work file' and a timestamp of '23 seconds ago'. Below this, a list of files and their commit history is displayed:

File	Commit Message	Time Ago
ExtraWork	Add extra work file	23 seconds ago
FinalProject	Initial commit	3 months ago
Labs	Add text-01.txt file	2 months ago
ProblemSets	Initial commit	3 months ago
.gitignore	Initial commit	3 months ago
LICENSE.md	Initial commit	3 months ago
README.md	Initial commit	3 months ago

We have verified that the file is complete and ready for integration into the **master** branch. To do this, we need to click on the **New pull request** button just to the right of the **Branch** button above your repository contents. Make sure that you are opening the pull request on your branch and not on the **master**!

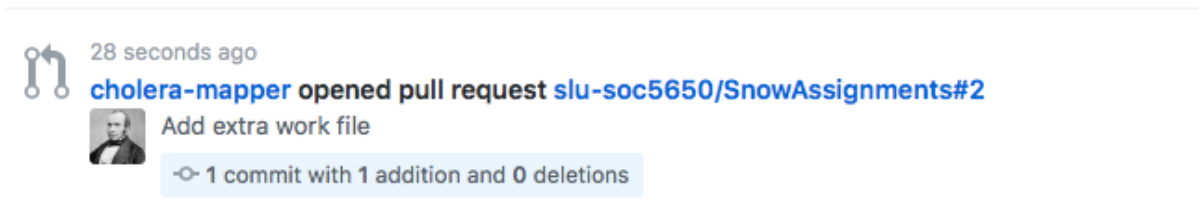
The screenshot shows the 'Open a pull request' page in GitHub. At the top, it shows the repository 'slu-soc5650 / SnowAssignments' with 1 Watch, 0 Stars, and 0 Forks. The main navigation bar includes links for Code, Issues (0), Pull requests (0), Projects (1), Wiki, Pulse, and Graphs. Below the navigation bar, the page title is 'Open a pull request'. A subtitle reads: 'Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).' Below this, a comparison bar shows 'base: master' and 'compare: snowBranch' with a green checkmark and the text 'Able to merge. These branches can be automatically merged.' The main content area is a form for creating a pull request. It has a title field with the text 'Add extra work file'. Below the title field are tabs for 'Write' and 'Preview'. The 'Write' tab is active, showing a text area with the text 'The extra work file contains important details about the assignment.' Below the text area is a dashed line indicating where to attach files. At the bottom of the form is a green button labeled 'Create pull request'. On the right side of the form, there are sections for 'Reviewers', 'Assignees', 'Labels', 'Projects', and 'Milestone', each with a gear icon for settings. The 'Reviewers' section shows 'No reviews—request one'. The 'Assignees' section shows 'No one—assign yourself'. The 'Labels' section shows 'None yet'. The 'Projects' section shows 'None yet'. The 'Milestone' section shows 'No milestone'.

The **pull request** page looks very similar to GitHub's **Issue** feature - there is a place for a title (which will

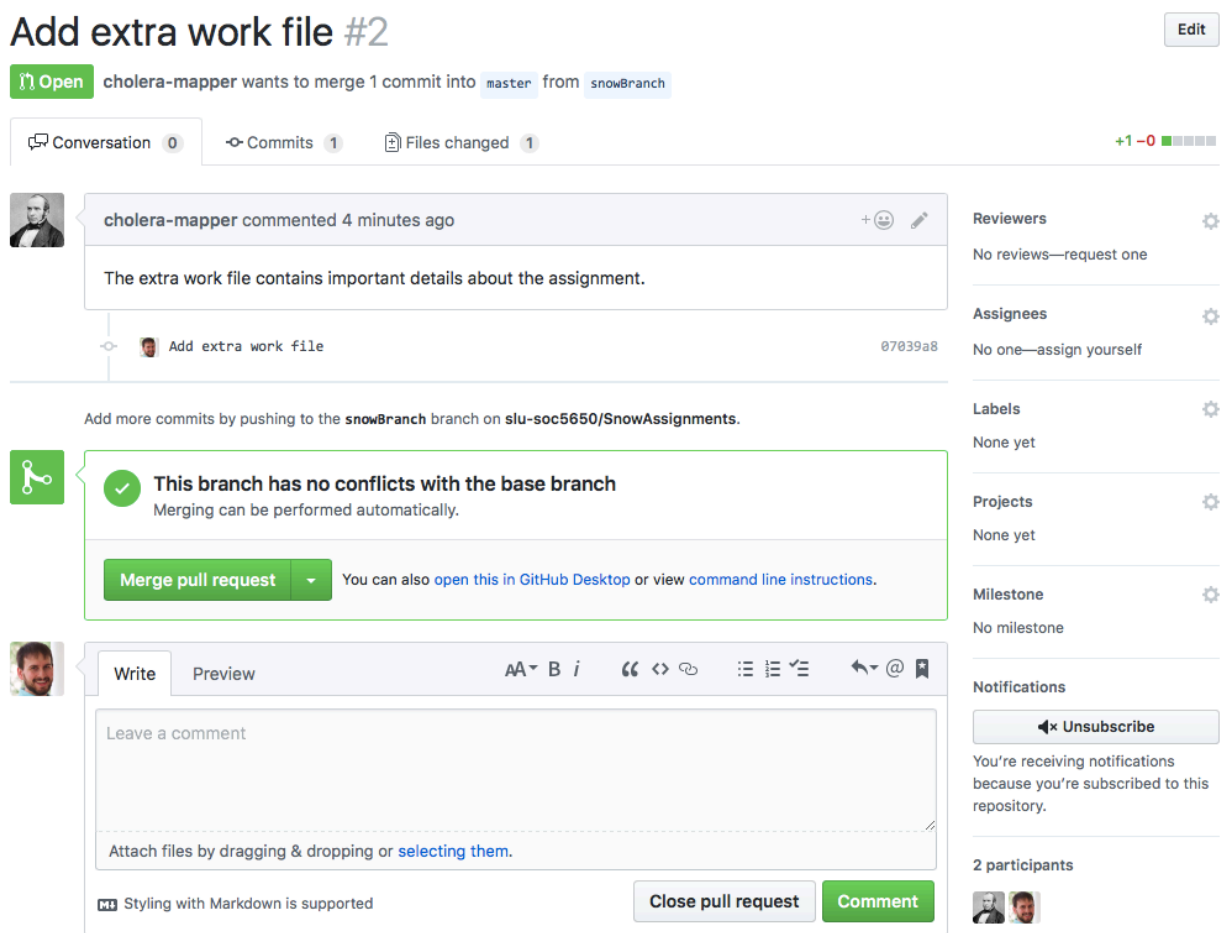
be autofilled for you) and for describing the contents and significance of your changes. You can optionally assign individual reviewers (such as your final project team members). Once you are done, click the green **Create pull request** button.

6.2.3 Evaluating Pull Requests

When a team member opens a **pull request**, you should see a notification in your news feed on the main landing page when you log into GitHub.com:



Clicking on that message (or, alternatively, selecting the **Pull Requests** tab in any repository) will take you to a page where you can evaluate changes that were made on your colleague's branch:

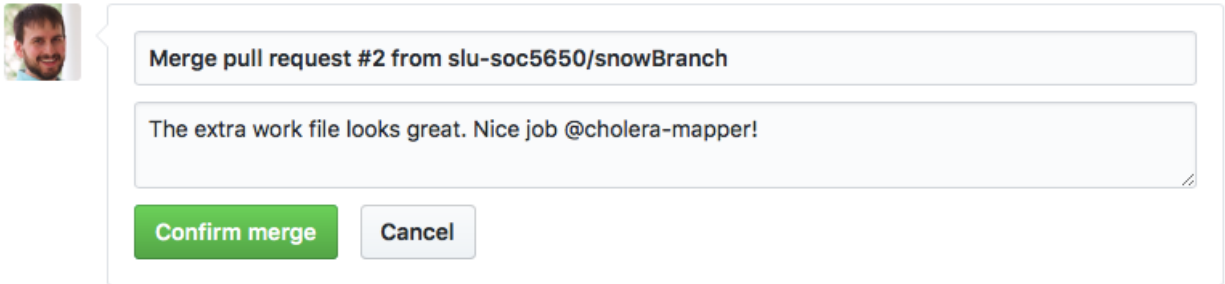


Here you can: * view files that have been changed or added, * respond to the pull request with a comment if you have follow-up questions, * view the results of GitHub's conflict check, * and choose an outcome (either merging - i.e. accepting - the pull request or closing - i.e. rejecting - the pull request)

This is an integral step. It is part of the peer review process that drives open-source software and is conducted in the same spirit of peer review that academics use to evaluate publications. The goal here is provide constructive feedback about proposed additions or changes. If you choose to close (i.e. reject) a pull request, give a detailed explanation as to why you are doing so.

If you see that there are conflicts with the base (i.e. **master**) branch, please let Chris know immediately to problem solve before any pull requests are accepted.

If you want to accept the changes, select the green **Merge pull request** button. Chosing this button will open up a space for you to add a commit message that corresponds with accepting the pull request:



Merge pull request #2 from slu-soc5650/snowBranch

The extra work file looks great. Nice job @cholera-mapper!

Confirm merge Cancel

After filling out your commit message and selecting **Confirm merge**, you have one final opportunity to either roll back the changes (by chosing **Revert**) or make them permanent by chosing **Delete branch**.

Chapter 7

Introduction to Atom

Atom is a flexible and simultaneously simple and powerful text editor. Text editors, unlike word processors, are not full fledged word processors. They are designed to work with *plain text*, which is ideal for working with computer code. Plain text is also ideal from a reproducibility standpoint, because it does not rely on proprietary file types like Microsoft’s Word document.

7.1 Packages

Packages for Atom are user-written programs that extend or expand its functionality. Atom is designed to be modular and thus can be almost endlessly customizable through the addition of various packages. When you open Atom, you can go to the Packages menu and see that some packages are already installed for you. One of these, **Markdown Preview**, is very helpful for getting a sense of what the Markdown syntax you are writing looks like. We’ll come back to that package in the “Introducing Markdown” chapter.

New packages can be added through **File > Preferences**¹. The Packages tab will summarize all of the packages you have currently installed. These will be segregated between Core Packages, those that power the base Atom distribution you downloaded, and Community Packages, which are packages that you choose to install to extend Atom. For most packages, you have the ability to access their settings and, when necessary, install updates.

The **Install** tab will allow you to search for and install those Community Packages. The following packages will be *required* for this course:

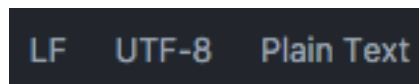
- `language-stata`
- `linter`
- `linter-markdown`
- `pdf-view`
- `tidy-markdown`

7.2 Languages

One of Atom’s strengths is its ability to accommodate various programming languages. By default, Atom documents are opened as plain text files. We’ll be using two primary file formats this semester: GitHub Markdown and Stata. Support for GitHub Markdown is included in Atom’s base distribution. Support for Stata is enabled with the installation of the `language-stata` package.

¹For users on macOS, you will find the Preferences under the Atom menu.

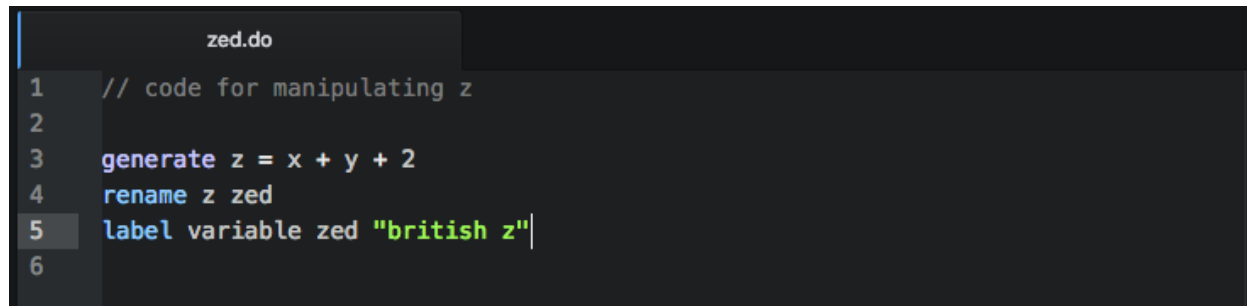
To switch between languages, you can either click on the language name in the lower righthand corner of the Atom screen. Again, by default, this will say “Plain Text” in a new document:



Use the search bar that appears to find the desired language, make sure it is highlighted, and hit Enter to select and switch to that language:



Once you select the language, Atom will provide syntax highlight to identify distinguishing features of your code, like commands, comments, and quoted text:



The exact appearance of the highlighting will change based on the theme that is enabled. These theme selections can be changed under **File > Preferences > Theme** tab.

When you save files, be sure to include the appropriate `.do` file extension for Stata do-files or `.md` for Markdown files. When you open files with `.do` file extensions, Atom will recognize that they are Stata do-files and automatically change the language setting to Stata. Likewise, it will recognize and adjust for Markdown files.

7.3 Using Snippets

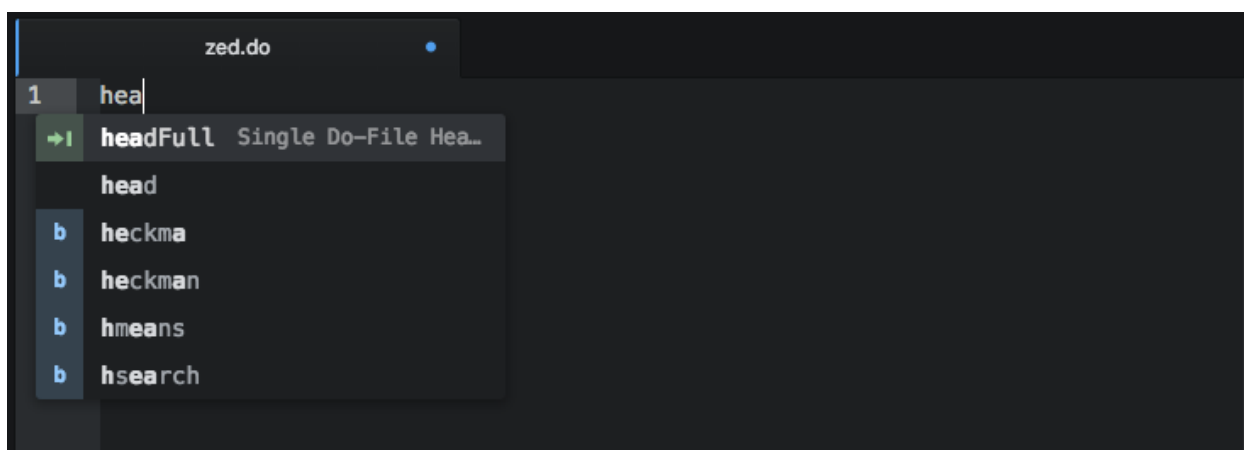
One of the advantages of working with Atom is that it has a powerful set of autocomplete tools. One aspect of these tools are what Atom calls “snippets”. These are blocks of text or code that can be expanded with a short trigger phrase called a “prefix”. The `week-02` repository contains a file with several snippets for both Stata and Markdown files.

7.3.1 Installing Snippets

To install these snippets, open up the file `atomSnippets.cson` in Atom. Then go to `File > Snippets`. The `snippets.cson` file will open. Any valid snippets saved here will be accessible to you when their associated programming language is activated in Atom. So, snippets for GitHub Markdown are only available when GitHub Markdown is selected, and snippets for Stata are only available when Stata is selected. Copy and paste the contents of `atomSnippets.cson` into `snippets.cson` beginning on a new line of the file. Save `snippets.cson`, close all open tabs, and restart Atom.

7.3.2 Expanding Snippets

To use a snippet, you'll want to change the programming language to the appropriate selection. Begin typing the prefix for your snippet, and a dropdown menu will appear. For example, typing `hea` in a Stata file when you are using our class snippets will bring up the following options:

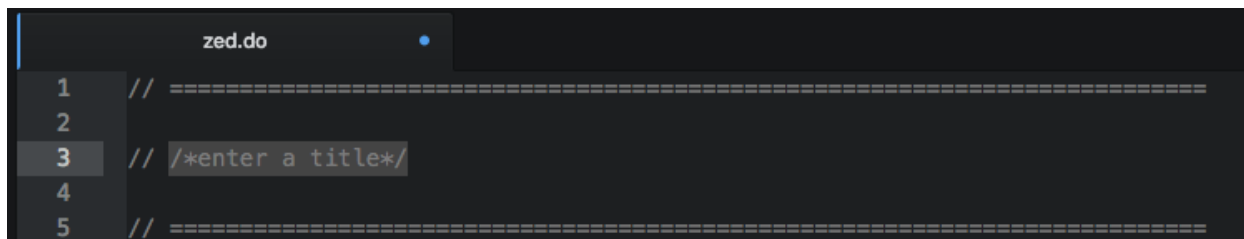


The green arrow icon indicates that the option is a snippet. No icon indicates that Atom is offering the full spelling of what it thinks is the appropriate word - “head” in this case. The blue “b” icon indicates a possible Stata command (this functionality is only available when the language is set to Stata).

You can use the up and down arrow keys to select a snippet from this list. Hit the Enter key for the selected snippet, and the snippet will expand into your file.

7.3.3 Using Tab Stops

Both of the primary snippets for this class include spaces for you to customize their content after you have expanded them. Once a snippet is expanded, the cursor will be automatically directed to the first tab stop. In the Stata snippet, that is a space to give your do-file a title:



Without clicking anywhere with your mouse, begin typing. Atom will replace the placeholder text with what you type:

```

zed.do
1 // =====
2
3 // MANIPULATE BRITISH ZED VARIABLES |
4
5 // =====

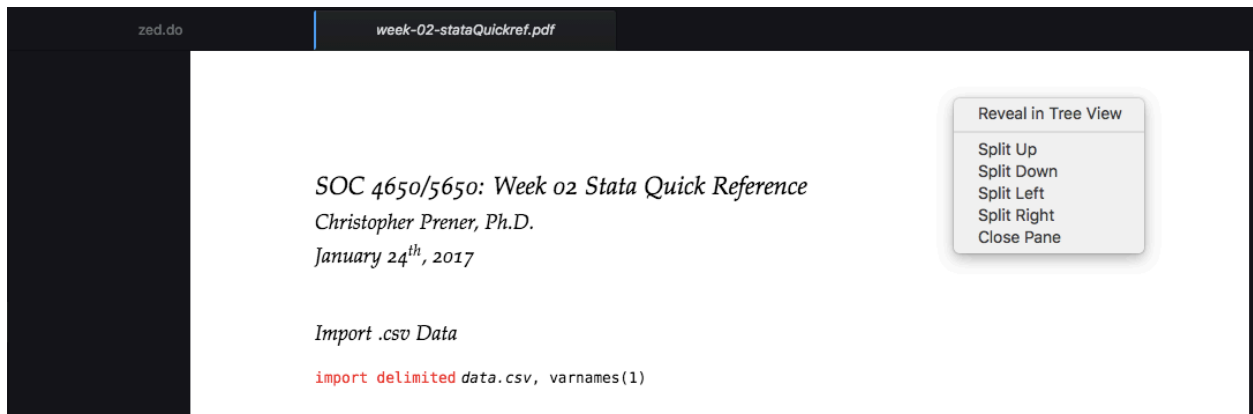
```

When you are done typing, hit the **Tab** key to be directed to the next tab stop. Continue typing and using the **Tab** key to move through the template.

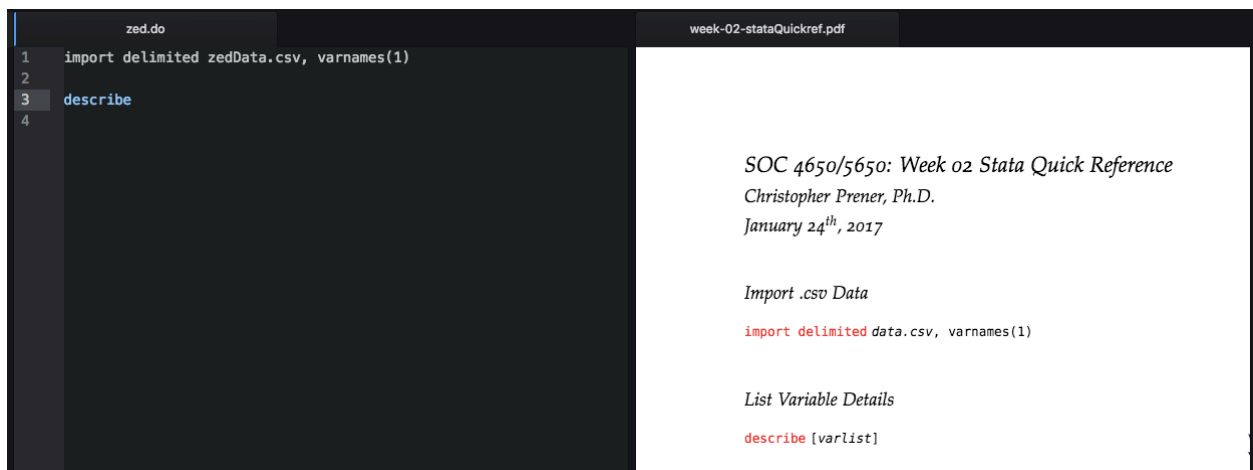
If you lose the **Tab** functionality for whatever reason, make sure you remove the `/*` and `*/` fences that sit on either side of the placeholder text. For Stata in particular, leaving these behind may cause errors or unexpected output.

7.4 Using Panes

Another advantage of Atom is that it allows you to work with multiple files open at once. This is helpful if you want to place our replication code side-by-side with your own code, or you want to keep reference materials close at hand as you write. You can achieve the split screen effect by right clicking on an open document and selecting **Split Right**:



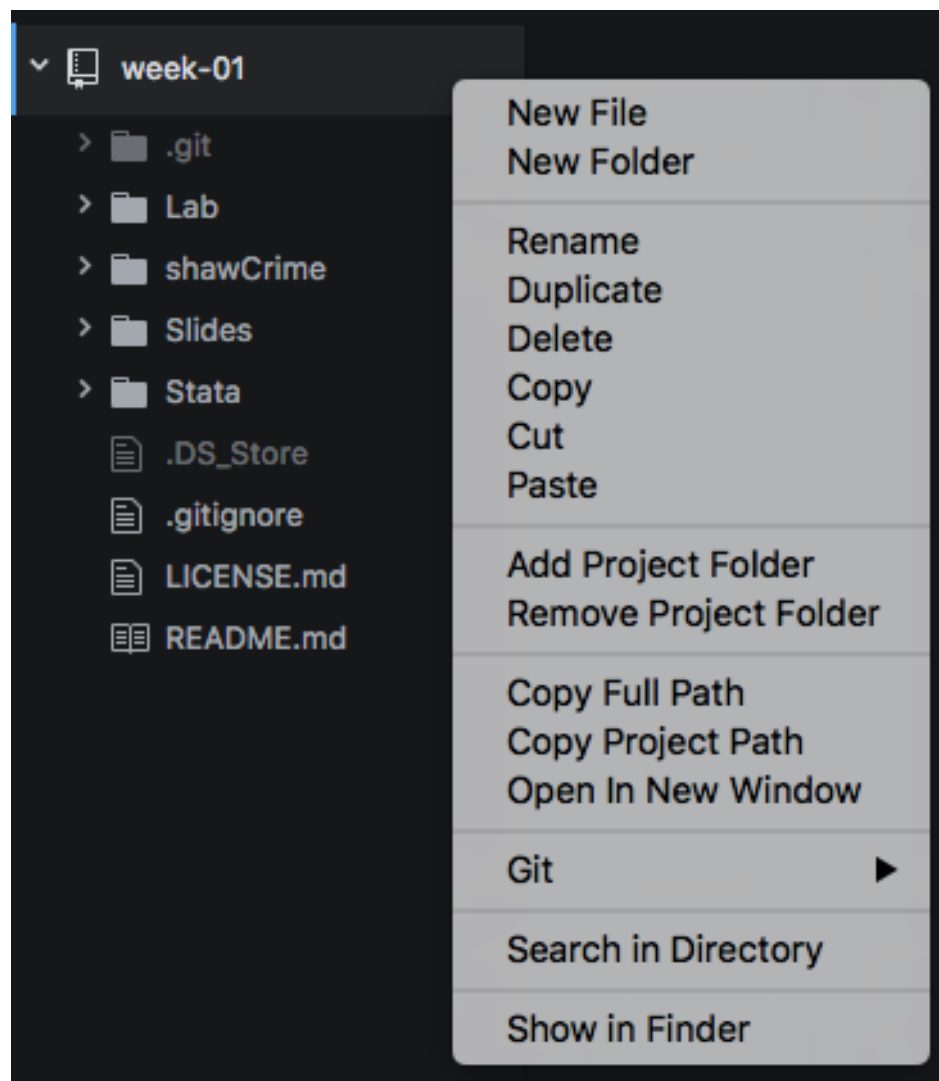
After you choose **Split Right**, you will have two panes displayed side by side:



This can allow you to refer to reference material as you write code, which you may find useful.

7.5 Using Project Folders

Project folders are directories that you make visible in the **Tree View**, the column on the lefthand side of Atom's window. Directories can be added to the Tree View by going to **File > New Project Folder....** Once a directory has been added, you have full control over its contents. This includes the ability to drag and drop files into subdirectories, copy and paste files from one project folder to another, create new files, and create new subdirectories. Much of this functionality is accessed by right clicking on the project directory itself:



To modify specific files, you can right click on the file itself.

Project folders make it possible to keep a weekly repository, the assignment directories for that week, and your assignment repository all close at hand. This will allow you to easily access example and reference materials, edit assignment files, and copy assignment files directly into your assignment repository for submission.

Chapter 8

Reproducible Do-Files

Do-files are scripts containing Stata code that can be used to execute a number of Stata commands in order. These provide significant advantages for researchers, making it easy to edit and adapt data cleaning and analysis processes. Finding an error in an analysis does not necessitate recreating hours of work. Instead, do-files allow researchers to quickly update the source of an error and re-execute the code. A do-file could be as simple as two lines of code:

```
generate z = x + y + 2  
rename z zed
```

Alternatively, large projects may necessitate hundreds or even thousands of lines of code spread out over multiple do-files.

There are some specific strategies that we can use to make our do-files serve not just a utilitarian function of executing code but also serve a larger purpose of increased reproducibility. The Stata snippet that we are using this semester in Atom is designed to incorporate these strategies. It will automatically create a series of subfolders in your working directory that contain a copy of the code you have executed, a copy of the raw data, a copy of the clean data, the log files, and any output you create.

The first section of this chapter gives the minimum amount of detail about using the snippet. The second section describes the process for “weaving” code and narrative text. If you want to make sure you are using the snippet correctly but are less concerned about how all the code works, just read these two sections.

The subsequent three sections break down the code in the snippet itself. If you want to know how all the code works, read these sections as well.

8.1 Using the Stata Snippet

The Stata snippet in Atom is named `headFull`. Once you switch a new document’s programming language to **Stata** and expand the `headFull` snippet, you can use the **Tab** stops to edit some key aspects of the file:

- line 3 - enter a title for your file - this should be a short, several word description that allows you to quickly identify a file’s purpose.
- line 9 - enter a project name - this should also be short and should not contain any spaces or special characters. Make sure when you save your do-file that its filename matches the project name entered here. So, for example, if the project’s name is `editZ`, the do-file’s name should be `editZ.do`.
- line 44 - enter a title for your analysis - for this class, this will most often be something to the effect of “Lab-01” or “Problem Set 01”.
- line 46 - enter your name
- line 47 - enter today’s date

- line 50 - enter a longer description of what this file accomplishes. This should describe in detail the goal of this file and what steps or tasks it accomplishes.
- line 70 - enter the name of the raw data file *with its file extension*. Depending on how long the project description is, this may be several lines below line 70.

Particularly early in the semester, it may be hard to come up with the right information for some of these prompts. Remember to *fully* read the assignment directions and prompts as you develop a plan for approaching the assignment. If you do this, you will get a better sense of the ultimate goal of the assignment and will therefore have an easier time answering these questions.

As I noted in the “Introduction to Atom” chapter, if you lose the **Tab** stop functionality, you will need to manually replace the placeholder text on each of the aforementioned lines. Make sure you remove the `/*` and `*/` fences that sit on either side of the placeholder text. For Stata in particular, leaving these behind may cause errors or unexpected output.

When you save your do-file, make sure you save it to the working directory where you have also copied the raw data and where you plan to house output created by the do-file.

8.2 Weaving Do-files

The workflow described here is largely adapted from the standard setting `knitr` package for R.¹ In the center of the snippet at line 72, you will see this message:

```
/* continue adding narrative and code chunks here */
```

Depending on how long the project description is, this may be several lines below line 72. This is where you want to begin entering the commands that are specific to the task you are working on completing.

The `Markdoc` package, as I have already noted, will combine your Stata commands, output, and narrative text into a single markdown file. This “weaving” of various sources produces code that can be read and executed by a computer, but is annotated in such a way that a human can easily understand its functionality. Narrative text can also be used to produce a document that purposely links code with output and description. In essence, you are writing the results section of a paper along with the code and output that produce it.

A typical combination of these items will look like this:

```
sysuse census.dta
```

```
/**
```

```
**1.** The `sysuse` command opens up pre-installed datasets that come with Stata.
The `census.dta` file contains demographic characteristics for all fifty states.
***/
```

```
summarize pop
```

```
/**
```

```
**2.** The `summarize` command produces descriptive statistics for the variable
`pop`. The mean population for a U.S. state is 4.5 million persons, though there
is considerable variability between states like Alaska with just over 400,000 persons
to states like California with over 23 million persons.
***/
```

Note how the commands precede any narrative text. Also note how narrative text is wrapped in two “fences”. These fences, `/**` and `/**/`, indicate to `markdoc` that the text should be included in the final output.

Once the do-file is fully executed, output will be added to the document as well:

¹See Xie, Y., 2015. Dynamic Documents with R and knitr (Vol. 29). CRC Press.

```
. summarize pop
(1980 Census data by state)
```

****1.**** The ``sysuse`` command opens up pre-installed datasets that come with Stata. The ``census.dta`` file contains demographic characteristics for all fifty states.

```
. summarize pop
```

Variable	Obs	Mean	Std. Dev.	Min	Max
pop	50	4518149	4715038	401851	2.37e+07

****2.**** The ``summarize`` command produces descriptive statistics for the variable ``pop``. The mean population for a U.S. state is 4.5 million persons, though there is considerable variability between states like Alaska with just over 400,000 persons to states like California with over 23 million persons.

To write the narrative correctly, it is necessary to preview the output by testing commands interactively in Stata until they produce the desired result. This gives you a chance to test all of your code, which will cut down on time spent debugging later, but will also give you the information you need to write the narrative sections.

8.3 Snippet Details: Header

The top of the do-file snippet, which we'll refer to as the “header”, is designed to create a clean environment for executing code in Stata and for saving output in your file system. The file begins with the title block described in the previous section and an area that defines the “project name”. This project name is stored in what Stata calls a **local macro**, an object that store information that can be recalled later and utilized. In the snippet, the local macro is named `projName`:

```
// =====

// define project name

local projName "projectName"

// =====

The next block of commands is to ensure that there are no holdovers from previous analyses in your Stata session. Most of these options are recommended by Long (2009).

// =====

// standard opening options

log close _all
graph drop _all
clear all
set more off
set linesize 80

// ++++++
```

The `log close _all` command closes any currently opening logs, ensuring that your do-file stack does not unintentionally edit any files. Similarly, the `graph drop _all` closes the graph window and the `clear all` command clears any other results or data stored in Stata's memory. The final two commands turn off the "more behavior" that limits the amount of output displayed by Stata (`set more off`) and constrains the output width to 80 spaces (`set linesize 80`).

After the environment within Stata is scrubbed, the snippet creates a series of subdirectories within your working directory.

```
// ++++++

// construct directory structure for tabular data

capture mkdir "CodeArchive"
capture mkdir "DataClean"
capture mkdir "DataRaw"
capture mkdir "LogFile"
capture mkdir "Output"
```

```
// ++++++
```

You will note that both the `mkdir` commands here are preceded with `capture`. The `capture` command will suppress any errors returned by the subsequent command and allow the do-file to continue executing. Used alone, the `mkdir` command will generate an error if a directory already exists with that name.

Finally, the header of the do-file begins logging the do-file's commands and output. Two log files are created. One is a plain text log file that is retained as part of your project's documentation. The second is a specially formatted type of output called a "smickle" file. It uses a special type of Stata syntax called SMCL to generate structured and formatted output.

```
// ++++++

// create permanent text-based log file
log using "LogFile/\projName'.txt", replace text name(permLog)

// create temporary smcl log file for MarkDoc
quietly log using "LogFile/\projName'.smcl", replace smcl name(tempLog)

// =====
```

That package that we will use this semester, Markdoc, relies on the temporary SMCL log file to generate Markdown formatted output. Notice that both of these commands refer to the project name **local macro** that we created earlier in the file. This is example of recalling previously stored information. If we wanted to change the name of the project and were not using a local macro, we would also have to change these two lines. However, because we used a local macro, changing the project name on line 9 will automatically result in changes to these filenames the next time the do-file is executed.

8.4 Snippet Details: Body

Once the log files are turned on, everything that is entered will be passed on to them. In addition to including the basic information like name, date, assignment, and a description of the code, the do-file snippet also contains some information about the software dependences that are required for your code to work.

```
### Dependencies
```

```
This do-file was written and executed using Stata 14.2.
```


It also uses the latest [MarkDoc](<https://github.com/haghigh/markdoc/wiki>) package via GitHub as well as the latest versions of its dependencies:
 ***/

```
version 14
which markdoc
which weave
which statax
```

```
// ++++++
```

The `version` command signals to Stata that this code was written for version 14 of the software. When running it under a future release (such as version 15 or 16), Stata will function as if it were running the older version 14. The `which` commands confirm that three packages are currently installed: **Markdoc**, **Weaver**, and **Statax**. **Markdoc** is the main tool that we'll use this semester to create do-files written in the style of literate programming. It in turn requires two other packages to function (**Weaver** and **Statax**).

After confirming that all of the necessary dependencies, the do-file template then uses another **local macro** to store the name of the raw data you are working on. As with the project name above, storing the raw data file's name this way makes editing code easier.

```
/**
### Import/Open Data
**/
```

```
local rawData "/*enter data file name with extension*/"
```

After this local, there is space reserved for you to enter your own commands along with narrative describing their function and their results. After you have completed this task, the do-file is structured to save the clean data in two formats: the Stata `.dta` file format and as a plain text `.csv` file.

```
// ++++++
```

```
/**
### Save and Export Clean Data
**/
```

```
save "DataClean/`projName'.dta", replace
export delimited "DataClean/`projName'.csv", replace
```

```
// =====
```

This ensures that your data are ready to be brought into ArcGIS, but you can also easily pick up editing them in Stata if necessary.

8.5 Snippet Details: Footer

Once the data are saved, it is time to wrap up the do-file's process. The first task in terms of ending our work is creating the markdown output that can be posted onto GitHub as part of an assignment's deliverables.

```
// =====
```

```
// end MarkDoc log
```

```
/*
quietly log close tempLog
```

```
*/
```

```
// convert MarkDoc log to Markdown
```

```
markdoc "LogFile/`projName'", replace export(md)
copy "LogFile/`projName'.md" "Output/`projName'.md", replace
shell rm -R "LogFile/`projName'.md"
shell rm -R "LogFile/`projName'.smcl"
```

```
// =====
```

The `markdoc` command takes the SMCL log file and converts its contents into markdown formatted text containing commands, output, and narrative text. Once this file is created, it is copied into the `Output` directory using the `copy` command with the `replace` option. This option is critical for being able to re-execute code without returning errors that a file with that name already exists at the location.

The SMCL log file and the original markdown file that was copied are then both deleted. It is worth noting that Stata has limited facilities for deleting content it creates. Its (limited) capacity to delete files and directories also varies by operating system. In order to accomplish the deletion of these two files in a way that works on both Windows and macOS operating systems, we need to invoke the operating system itself with the `shell` command. What follows the `shell` command in each case is actually an instruction to the operating system itself. In both Windows and macOS, the `rm` command with the `-R` (recursive) option will permanently delete a file.

With unneeded files removed, we can archive our code and raw data using the same `copy` command use above.

```
// =====
```

```
// archive code and raw data
```

```
copy "`projName'.do" "CodeArchive/`projName'.do", replace
copy "`rawData'" "DataRaw/`rawData'", replace
```

```
// =====
```

After the code are archived, the snippet is structured to close both the log file and the graph window (if it is open). It also re-sets the `more` behavior so that it will occur if Stata is subsequently used in interactive mode. Finally, the `exit` command is issued to explicitly end the execution of the do-files.

```
// =====
```

```
// standard closing options
```

```
log close _all
graph drop _all
set more on
```

```
// =====
```

```
exit
```

Chapter 9

Introduction to Markdown

Markdown is a simple markup language. Markup languages are used to give computer programs directions on how particular blocks of text should be processed. Markdown was developed in 2004 by writer and developer John Gruber. Gruber describes Markdown on his website:

Markdown is intended to be as easy-to-read and easy-to-write as is feasible.

Readability, however, is emphasized above all else. A Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. While Markdown's syntax has been influenced by several existing text-to-HTML filters — including Setext, atx, Textile, reStructuredText, Grutatext, and EtText — the single biggest source of inspiration for Markdown's syntax is the format of plain text email.

To this end, Markdown's syntax is comprised entirely of punctuation characters, which punctuation characters have been carefully chosen so as to look like what they mean. E.g., asterisks around a word actually look like *emphasis*. Markdown lists look like, well, lists. Even blockquotes look like quoted passages of text, assuming you've ever used email...

...Markdown's syntax is intended for one purpose: to be used as a format for writing for the web.

9.1 Markdown Syntax

As markup syntaxes go, Markdown is exceptionally straight forward. The following sections include examples of syntax used to create Markdown documents. These are specific to what is called GitHub Markdown - there are some subtle differences in the way GitHub uses Markdown formatting.

9.1.1 Headings

Markdown contains six heading levels. Headings are identified with # symbols:

Chapter 10

This is the largest heading

10.1 This is the second largest heading

10.1.0.0.1 This is the smallest heading

10.1.1 New Paragraphs

Leave a blank line between two lines of text to create a new paragraph.

10.1.2 Styling Text

Text can be styled using bold, italics, and strikethroughs. To create italicized text, wrap your text with a single asterisk `* *`. To create bold text, wrap your text with double asterisks `** **`. To create strikethrough text, wrap your text with two tildes `~~ ~~`.

This is an italicized sentence.

This is a bolded sentence.

~~This is a sentence with strikethrough text~~

10.1.3 Quoting Text

Quoting text (which I have used above to illustrate examples) is done with a greater than symbol `>`.

10.1.4 Quoting Code

There are two types of code quotes in Markdown. In-line quotes, which are included in a sentence, are wrapped in single backticks: `>` Use the **describe** command to list variables in Stata.

To include code blocks, which are better for including the full syntax of particular commands and their output, use triple backticks:

```
. describe make price mpg
```

	storage	display	value	
variable name	type	format	label	variable label
make	str18	%-18s		Make and Model
price	int	%8.0gc		Price
mpg	int	%8.0g		Mileage (mpg)

Note how the word ‘Stata’ is written after the first set of triple backticks. This is an indicator for GitHub that the code is written in Stata’s programming language. By including this, GitHub can apply some syntax highlighting to your files. This makes them easier to read.

10.1.5 Links

In Markdown, adding hyperlinks is a two step process. The text that you want to have hyperlinked is written first and is wrapped in brackets []. After this, you include the URL wrapped in parentheses (). This is an example of including in-line hyperlinks:

The course website is hosted using the service GitHub.

10.1.6 Embedding Images

Within the directory that contains your Markdown file, create a subdirectory called **Output**. Save all images for a particular assignment there. In your main assignment Markdown file, include a hyperlink reference:

Note how, instead of including text to be hyperlinked, we suppress this aspect of the syntax by using an exclamation point !.

10.1.7 Simple Lists

Bulleted lists are indicated in Markdown using the dash - or a single asterisk *:

- mean
- median
- mode
- variance
- standard deviation

Enumerated lists are created by preceding each line with the appropriate number:

1. calculate the mean
2. calculate the variance
3. calculate the standard deviation

You can create more complex lists by preceding a line with two single spaces. You can also combine bulleted and enumerated lists when using this approach.

10.1.8 Task Lists

If you want to create task lists on GitHub, you can include brackets separated by a space before each list item []. Completed tasks include an x in place of the space [x]. These task lists are interactive - when published on GitHub, you can click on the resulting checkboxes to toggle them between complete / incomplete.

1. `[x]` calculate the mean
2. `[]` calculate the variance
3. `[]` calculate the standard deviation

10.1.9 Mentioning Other GitHub Users

If you want to mention me or one of your classmates in a comment, include the `@` symbol before their username:

Hey `?`, thanks for the feedback. I made the changes to lines 40 and 41.

Once the document is uploaded to GitHub, the username will render as a hyperlink and the user will be alerted.

10.2 Markdown and Stata

10.3 Markdown and Atom

If you write your Markdown documents in Atom, you can use the Markdown Preview package to generate an interactive preview of your document. As you type, the preview will update. To open this preview in a tab, go to `Packages > Markdown Preview > Toggle Preview` in Atom.

Bibliography