Curating Research Assets in Behavioral Sciences: A Tutorial on the Git Version Control System

Matti Vuorre[1] & James P. Curley[1,2]

[1] Department of Psychology, Columbia University, New York, USA
[2] Department of Psychology, University of Texas at Austin, Texas, USA

Abstract

Recent calls for improving the reproducibility of behavioral sciences have increased attention to the ways in which researchers curate, share and collaborate on their research assets. However, these discussions have largely failed to provide practical instructions on how to do so. In this tutorial paper, we explain how version control systems, such as the popular Git program, address these challenges to reproducibility. We then present a tutorial on how to use Git from the computer's command line, and with the popular graphical interface in the R Studio development environment. This tutorial is especially written for behavioral scientists with no previous experience with version control systems or command line interfaces, and covers single-user and collaborative workflows. Git is easy to learn, presents an elegant solution to specific challenges to reproducibility, facilitates multi-site collaboration and productivity by allowing multiple collaborators to work on the same source simultaneously, and can be implemented on common behavioral science workflows with little extra effort. Git may also offer a suitable solution to transparent data and material sharing through popular online services, such as GitHub and Open Science Framework.

Keywords: reproducibility; version control; git; research methods; open science

Word count: X

Correspondence concerning this article should be addressed to Matti Vuorre , 406 Schermerhorn Hall, 1190 Amsterdam Avenue MC 5501, New York, NY 10027 . E-mail: mv2521@columbia.edu

Introduction

The lack of reproducibility is increasingly recognized as a problem across scientific disciplines, and calls for changing the scientific workflow to enhance reproducibility have been published in a wide range of research areas, including Biology (Markowetz, 2015), Ecology (Ihle, Winney, Krystalli, & Croucher, 2017), Neuroscience (Eglen et al., 2017) and Psychology (Munafò et al., 2017). However, although calls to focus on reproducibility are now commonplace on the pages of leading scientific journals (e.g. Baker, 2016), only a small minority of researchers are familiar with the tools and practices that enable implementing reproducibility in their workflows. Therefore, although there now is a broad consensus that efforts to improve reproducibility are important, materials instructing researchers in using them are lacking.

Reproducibility

Consider for a while if you have ever struggled with the following questions (Ihle et al., 2017, p. 2): Have you ever found a mistake in your results without knowing what caused it? Forgot what analyses you have already done and how (and possibly, why)? Lost datasets or information about the cases or variables in a dataset? Struggled in redoing statistical or computational analyses when new data became available? Had difficulty understanding what data to use or how in a project that you inherited from another researcher? Answering any of these questions in the affirmative suggests that your work might benefit from improving reproducibility (Ihle et al., 2017).

But what exactly is reproducibility? Reproducibility can be defined as follows: "A research project is computationally reproducible if a second investigator (including you in the future) can recreate the final reported results of the project, including key quantitative findings, tables, and figures, given only a set of files and written instructions." (Kitzes, Turek, & Deniz, 2017). In the context of Experimental Psychology, for example, a "project" could be an experiment or a set of experiments investigating a hypothesis, "reported results" could be figures or statistical analyses in a conference presentation or a manuscript.

This definition makes an important distinction between *reproducibility* and *replicability*. Methods and results sections in traditional journal articles have focused on ensuring replicability by giving detailed instructions on how to repeat the experiment and collect a data set similar to the original one. However, little or no emphasis is placed on issues related to reproducibility, such as details of the data processing pipeline and statistical model exploration, although these issues are often more important to the reliability of the scientific results than are their end results (e.g. reported *p*-values; Leek and Peng (2015)). To put it more generally, replicability is a broad conceptual issue concerning the epistemology of scientific claims. Reproducibility, on the other hand, is a technical requirement for allowing others (including researchers themselves in the future) to assess the reliability of a specific set of reported empirical or computational results.

Further, although they are sometimes erroneously conflated (Open Science Collaboration, 2015; Peng, 2011) reproducibility and replicability are independent of one another. Although the distinction between the two may be somewhat blurred in purely computational areas (e.g. simulation studies; Peng (2011)), it is clear that an experiment can be replicated even if the materials of the original study are not available for replicability. Likewise, the materials of an experiment might allow its results to be reproduced, yet the underlying scientific hypothesis might not be replicable. Therefore, it is important to keep these two concepts distinct; in the remainder of the manuscript we explicitly only discuss reproducibility.

Given this broad definition of reproducibility, it is easy to recognize that many projects fall between the extremes of completely reproducible and not reproducible at all. A completely reproducible project would provide consumers the entire raw data set with detailed and unambiguous instructions to analyze it (usually as computer code). This degree of reproducibility is very rare in Psychology (Vanpaemel, Vermorgen, Deriemaecker, & Storms, 2015; Wicherts, Borsboom, Kats, & Molenaar, 2006), although data sharing has become more common in recent years in journals that award authors who do so with special badges (Kidwell et al., 2016). However, even when authors (report to) share data, the shared data sets sometimes turn out to be incorrect, not usable, incomplete, or not available at all (Kidwell et al., 2016). This suggests that reproducibility requires more than good intentions. What, then, are some specific challenges to reproducibility?

Challenges to Reproducibility

Evidence suggests that the level of reproducibility within Psychology is not very high. One survey of leading Psychology journals found that 34-58% of articles published between 1985 and 2013 had at least one inconsistently reported *p*-value (Nuijten, Hartgerink, van Assen, Epskamp, & Wicherts, 2015): Upon recalculation, the authors couldn't obtain the same *p*-values from the reported test statistics, and therefore these 34-58% of articles were, to some degree, non-reproducible (the inferential statistics could not be reproduced).

Why, do so many research products fall short of the basic scientific standard of reproducibility? We suggest one reason is the poor level at which researchers organize, curate, and collaborate on the research assets—e.g. data and code used to analyze the data—which support the product's conclusions. In one study, researchers asked authors of 141 Psychology articles (with a total of 249 studies) to share their data for reanalysis. 73% of the authors refused to share their data, leading the study's authors to suggest that the considerable efforts involved in cleaning datasets at the final stages of publication may make data sharing unattractive to many (Wicherts et al., 2006). Another similar investigation (Vanpaemel et al., 2015) highlighted some common reasons provided by authors for not sharing their data:

"To our surprise, some authors are apparently willing to share, but have

> no easy access to their own data or have lost their data altogether, due to computer crashes or collaborators having left the university. Many authors cite a lack of time as a reason not to share, and note that sharing their data would take too much effort, which is probably due to poor documentation and storage practices." (p. 2-3)

In short, scientists usually excel at *producing* research assets, but commonly fail at *curating* them (a topic commonly known as Digital Asset Management; Jacobsen, Schlenker, and Edwards (2012)). This state of affairs is not surprising because researchers are trained to do research, but receive little or no training in formal methods or accepted gold standards for curating their research assets. For simple[1] projects, such as writing a solo-author manuscript, this may not be a problem. However, with increasing complexity of the type of data collected by behavioral scientists, and the increased number of collaborators, research sites, and technical skills required in modern scientific workflows, not knowing how to curate research assets can lead to wasted time, or worse, as detailed above.

Fortunately for the empirical sciences, challenges related to organizing and curating materials ("digital assets") across time, space, and personnel have been solved to a high standard in computer science with Version Control Systems (VCS). In the remainder of this article, we introduce a popular VCS called Git, and illustrate its use in the scientific workflow with a hypothetical example project. In the tutorial below, we show how to use the Git VCS by using text commands from the computer's command line. We also explain in detail how to use Git with the popular on-line service GitHub for collaborative workflows. We also show how to use Git easily with a Graphical User Interface (GUI) implemented in the popular R Studio Integrated Development Environment (IDE; RStudio Team (2016)).

Version Control Systems

Version Control Systems (VCS) are computer programs designed for tracking changes to computer code, and collaborating on the code with others. VCS were initially developed by software engineers for writing code collaboratively[2], but are increasingly being adopted to enhance workflows outside computer science. To help understand why, it is helpful to think of "code" more broadly as any text written on a computer: Manuscripts, books, statistical analysis scripts, source code of computerized experiments, and even data files are "code" or have source code, which is just plain text written on a computer, and which can benefit from version control. Going forward, when we use the word "code" in this tutorial, we mean it in this broad sense (e.g. this manuscript's source "code" was written on a computer, and was version controlled.)

[1]By "simple", we only mean that the technical aspects of the project, related to curating materials related to it, are simple. We of course do not suggest that there might be anything theoretically or scientifically simple in such projects.

[2]The software we present below is used used by major software developers such as Microsoft, Google, and Facebook on code bases with hundreds of contributors.

For example, computerized behavioral experiments are written with a computer programming language such as MATLAB or Python. The experiment's source code (text written by human, interpreted by computer to e.g. display stimuli to participants) may have multiple authors and usually goes through multiple versions. Keeping track of the versions of the program and changes to the code, and allowing many authors to contribute to it (without breaking the experimental program) are problems that VCSs were specifically designed for. It might be less obvious that writing a manuscript is quite similar, as far as the computer is concerned: Multiple authors write multiple versions of a text document, and sometimes previous versions need to be inspected, and the text needs to be "merged" across the many authors. Even less obvious is that data itself is often plain text: In most computerized behavioral experiments, the output data are numbers and text written into a text file. These text files can then be version controlled (and researchers would usually not want to see that their raw data files have changed after they were created—VCS allows verifying that they haven't changed because their history is logged.)

The core concepts of version control are that contributors to a project create small checkpoints of the changes they make to the source code—analogous to saving an intermediate version of a file on the computer's hard disk—and then submit those changes to the VCS. The VCS maintains a history of changes to the code between these little checkpoints, and therefore allows coming back to any earlier version by browsing the history. These concepts—i.e. the typical Git workflow—is illustrated in Figure 3.

To understand what VCS is, it is useful to contrast an example scenario with VCS to what we believe is a fairly standard workflow. Consider collaborating with one person on a manuscript that reports results from a data set using some statistical model. In the standard workflow, one person might format the raw data in one way to fit a statistical model, and then write a draft of the manuscript. This would create three files: `data.csv`[3], `analysis.R`, and `manuscript.doc`. If the coauthor then wished to explore another statistical model, which requires the data in a different format, and then edit the manuscript, she would create three more files: `data_new.csv`, `analysis_new.R`, and `manuscript_new.doc`. This cycle would then repeat as many times as required, each time creating more files, making it more and more difficult for the authors to exactly remember which data was paired with which analysis, and which analyses (and data format) was reported in which version of the paper.

VCS greatly simplifies and streamlines this workflow: With VCS, there would be only three files (data, analysis, and manuscript), but they would all be under version control, which would keep track of changes to the files. Because VCS monitors changes to the files and allows saving a history for each of them, creating new files for every new idea or edit is unnecessary. This lack of duplicity, in turn, possibly reduces room for error in remembering which data file was linked with which analysis, and

[3]`.csv` stands for comma separated values, and is a plain text file commonly used to store two-dimensional data.

which manuscript version had the correct numerical results, and so forth. Further, as we explain below, VCS allows many coauthors to work on the files simultaneously, eschewing the need for emailing different versions of the same file to the coauthors, then waiting for them to make changes before continuing with your work. Box 1 shows practical examples of scenarios where VCS (specifically, Git, see below) can be used to improve the scientific workflow.

Box 1. How Git facilitates the scientific workflow.

- How to try different ways of visualizing and modeling data while keeping track of the different versions—and the differences between them?
  - Git saves each version of the analysis, allowing testing new features without losing previous versions or proliferating files in the project's directories. All the past versions can be directly compared with each other, and retrieved from the Git's history.
- How to work on the same code or manuscript files simultaneously with collaborators?
  - Git was designed for distributed collaborative work: Collaborators work on their local copies and "push" and "pull" material to and from each other or a central "repository". Git keeps precise track of who has done what, when, and (possibly) why. Git never loses information or overwrites work with another collaborator's work, but allows for many authors to work on the same ideas simultaneously.
- How to share my work easily and in an organized manner with others?
  - Git enforces a common organization scheme among collaborators, making it easier to keep everyone "on the same page" with what goes where, and how to contribute to specific parts of the project. Sharing the project with others is built into Git, and can be facilitated with online services such as GitHub: Once a project uses Git, it can be very easily copied to GitHub, from where others can easily download the entire project onto their local computers.

The Git Version Control System

Version control software has a long history in software engineering, and there are many VCS programs. Some popular ones are Apache Subversion[4], Mercurial[5], and Git[6]. In this tutorial, we focus on Git because it is already incraesing in popularity within the scientific community, and is especially good for scientific collaboration because of online tools (GitHub) which allow seamless collaboration even for very large research teams. Further, Git is free and open source, works on Windows, Mac, and Linux OSs (among others).

Git, unlike the operating system's (OS) default file viewer (Mac's Finder, Windows' File Explorer), is not a point-and-click program for navigating files and folders

[4]https://subversion.apache.org/

[5]https://www.mercurial-scm.org/

[6]https://git-scm.com/. The creator of Git, Linus Torvalds, named Git after himself as "the stupid content tracker" (McMillan (2005); https://git-scm.com/docs/git.html)

on a computer. Instead, Git adds functionality to the existing file system, by making available a specific set of commands—either executed from the command line, or through a graphical user interface (GUI)—which allow keeping track of files and their history, and distributing the files across multiple computers and users. Because Git is a standalone program not usually included in standard OS installs, users must first download and install the Git software on their computers.

Installing Git

Even if you already have Git installed (some computers do) it is good practice to install the latest version[7], which can be downloaded as a standalone program from https://git-scm.com/download. For Mac users, the easiest way to install or update Git to the latest version is to download the installer from http://git-scm.com/download/mac, and install it like any other program. Similarly, Windows users can download the Git software installer from http://git-scm.com/download/win and install it like any other application. Linux (and other) users can download Git and find further install instructions on the Git website (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git).

Git setup

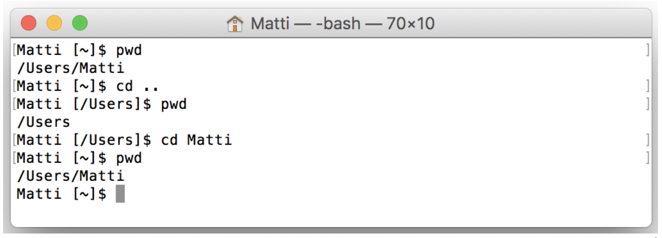
Once you have installed Git, you can use it from the computer's command line or through a GUI—there are various GUIs for using Git, and below we will explain how to use the R Studio Git GUI. First, it is important to learn a few basic Git commands from the OSs command line, because it is simultaneously the most basic and most flexible interface to Git's functionality. Understanding the basic Git commands as entered through the computer's command line also facilitates understanding its more advanced uses, and is highly recommended.

The command line. The command line is a text-based interface for interacting with your computer, and its functionality greatly extends that of the standard way of interacting with the computer by clicking and pointing with a mouse. Many advanced techniques require using your computer through a command line, and it is very helpful in e.g. scripting and scheduling tasks on your computer. Here, we introduce the command line in just enough detail so that you can navigate folders on the computer, and set up Git's basic configuration (identify Git's user).

To access the command line interface, you need to use a command line "shell" application. Mac users can open the built-in app Terminal, and Windows users can use the Git Bash application, which is installed with the Windows Git program. After opening the command line shell, you can type in commands and execute them by pressing Return (Mac) or Enter (Windows).

First, you will want to know how to navigate the folders on your computer (a task that is typically done by clicking folders in Finder (Mac) or File Explorer

[7]As of the writing of this article, the current version of Git is 2.13.1.



```
Matti [~]$ pwd
/Users/Matti
Matti [~]$ cd ..
Matti [/Users]$ pwd
/Users
Matti [/Users]$ cd Matti
Matti [~]$ pwd
/Users/Matti
Matti [~]$
```

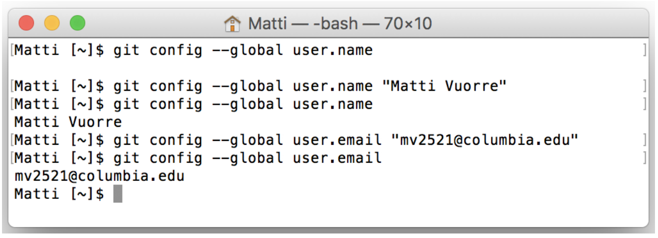
Figure 1. The Terminal command line shell. The functions are explained in more detail in the text. Each command (prepended with a *symbol)isfollowedbyitsoutputonthefollowingline.Totheleftofthe* symbol, this user's shell application displays the user's username and the current directory in square brackets. Here, the outputs are directories separated with forward slashes, and topmost (containing, also known as parent) folders are on the left of their subfolders (also known as children). Your command line interface might look slightly different because of different user and folder names, operating systems, and command line shell applications.

(Windows)). Usually, when you open up your command line shell, you begin in the user's home directory (or folder, we use these terms interchangeably). Depending on your operating system, the home directory is often represented with a `~` on the left side of the cursor. To ask for the current working directory, you can use the function `pwd`. To move up in the directory hierarchy (into the folder that contains the current directory), you can use `cd ..` (note the space). To move into a folder that is inside the current working directory, you can use `cd folder` where `folder` is the name of the desired folder. These command line functions are illustrated in Figure 1.

Some users may find using a text-based command line interface unfamiliar, but to get started with Git, there are two required configuration commands which you need to run once, and the basic functionality requires using only four commands[8]. However, users can also read through the first part of the tutorial without executing the commands and simply focus on the workflow; we show how to use Git with a GUI in the second part of the tutorial.

Setting Git's user information. The first step in using Git is making it aware of who is using the computer. You need to set the user's name and email address by entering a few basic commands in the command line. First, to show the current user information, type `git config --global user.name`, and hit return. This should not return anything, unless a previous user of the computer has set the global Git user name. Each Git command starts with the word `git`, then a command (such as `config`), and then arguments to the command, such as `--global` (for global configuration), followed by variables, such as `user.name`.

[8]If using a text-based command line seems challenging, Codecademy (https://www.codecademy.com/learn/learn-the-command-line) has a free interactive online tutorial, and MIT offers a free online game to teach using the command line (http://web.mit.edu/mprat/Public/web/Terminus/Web/main.html).



```
Matti [~]$ git config --global user.name

Matti [~]$ git config --global user.name "Matti Vuorre"
Matti [~]$ git config --global user.name
Matti Vuorre
Matti [~]$ git config --global user.email "mv2521@columbia.edu"
Matti [~]$ git config --global user.email
mv2521@columbia.edu
Matti [~]$ 
```

Figure 2. Setting up Git's configuration using the command line. Notice that for these configuration commands, the current working directory does not matter (we did them in the user's home directory).

To ensure that Git knows who you are, type `git config --global user.name "User Name"` (where `User Name` is your name) in the command line, and hit return. This command maps `"User Name"` to Git's global `user.name` variable. If you now re-run the first command (`git config --global user.name`), the command line will return the name you entered as `"User Name"`. The user name can be anything you'd like, but it is probably a good idea to use your real name so that potential collaborators know who you are. The second piece of information is your email address, which is entered by `git config --global user.email "email@address.com"` (where `email@address.com` is your email address). You can verify that the correct email address was saved with `git config --global user.email`. These commands are shown as entered to the Terminal command line shell application in Figure 1.

Once this information is entered, Git will know who you are, and is able to track who is doing what and when within a project, which is especially helpful when you are collaborating with other people, or when you are working on multiple computers. For detailed instructions on how to use Git commands, you can type `git --help`. For help on how to use the `git config` command, type `git config --help`.

Using Git

The first operating principle of Git is that your work is organized into independent projects, which Git calls *repositories*[9]. A repository is a folder on your computer which is version controlled by Git[10]. Everything that happens inside a repository is tracked by Git, but you have full control of what is committed to Git's history and when. Because you have full control of what and when is committed to history, there

[9]For advanced users, Git *submodules* allow linking projects to each other, or organizing more complex projects into projects and their sub-projects (https://git-scm.com/book/en/v2/Git-Tools-Submodules).

[10]There are no visible changes to a folder once it is tracked by Git. Once Git is initialized in a folder, the only change is that a hidden folder, called `.git` is added, but users do not need to interact with it directly.

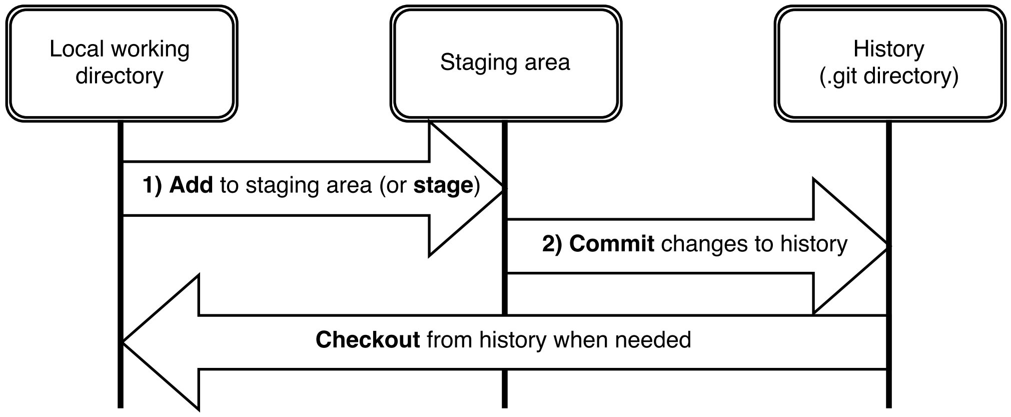

Local working directory
Staging area
History (.git directory)
1) **Add** to staging area (or **stage**)
2) **Commit** changes to history
Checkout from history when needed

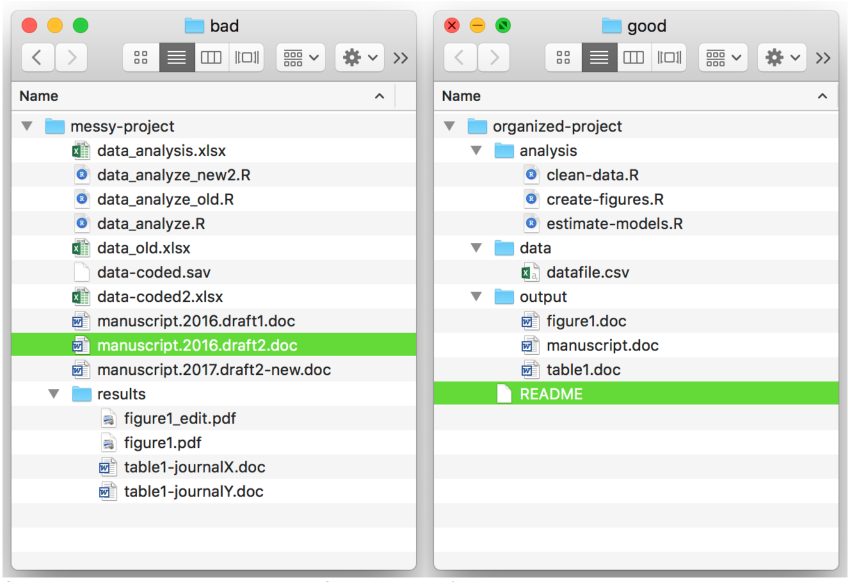
Figure 3. A diagram illustrating the typical Git workflow. Verbs indicated in bold text are Git operations and explained in detail in the main text. In brief, this workflow consists of making changes to files in the project (such as editing a manuscript.) Once the revised version of the file is saved, the user adds the changed file to the staging area (1). Many files can be added to the staging area, if desired. Once the changes in the staging area reflect a conceptual entity, such as edits to an image file and its accompanying caption in the manuscript, the user then commits the changes to Git's history (2). These commits are accompanied with short commit messages that describe the changes made in that commit. Finally, when needed, changes to files can be discarded by checking out an earlier version of the file from Git's history. Figure adapted from https://git-scm.com/book/en/v2/Getting-Started-Git-Basics.

is a small set of operations you need to know[11].

Briefly, when you work in a Git repository (make changes to files within it), Git monitors the state of each file, and when they change Git knows that they differ from the previously logged state. If you are happy with the current changes, you **add** the changed files to Git's "staging area". If you then are certain that the changes in the staging area are desirable, you **commit** the changes. These two operations are the backbone of using Git to store the state of the project whenever meaningful changes are made. Importantly, each commit in the repository's history contains information to recover the full state of the project at that point in time. Users can always go back to an earlier version by **checking out** a previous state from Git's history. This core of the Git workflow is illustrate in Figure 3.

To understand the Git workflow in practice, we now turn to a practical example using a hypothetical project. Git can be added to a project at any stage of the

[11]It helps to have a Git command cheat sheet (https://services.github.com/resources/cheatsheets/) printed and taped on your wall, but it contains many more commands than are needed for the basic use of Git in standard Psychology studies.



```
bad
Name
▼ messy-project
    data_analysis.xlsx
    data_analyze_new2.R
    data_analyze_old.R
    data_analyze.R
    data_old.xlsx
    data-coded.sav
    data-coded2.xlsx
    manuscript.2016.draft1.doc
    manuscript.2016.draft2.doc
    manuscript.2017.draft2-new.doc
  ▼ results
      figure1_edit.pdf
      figure1.pdf
      table1-journalX.doc
      table1-journalY.doc
```

```
good
Name
▼ organized-project
  ▼ analysis
      clean-data.R
      create-figures.R
      estimate-models.R
  ▼ data
      datafile.csv
  ▼ output
      figure1.doc
      manuscript.doc
      table1.doc
    README
```

Figure 4. Organizing a project's files and folders helps potential collaborators (including oneself in the future) to quickly and reliably find the correct files. Some bad practices (left panel) include having multiple versions of each file, increasing the potential for e.g. accidentally using the wrong data for the analysis, or forgetting which version was used in the analysis. Better practices (right panel) organize the data to subfolders with meaningful names, and have one file per purpose. These files are then versioned using Git, eschewing the need for multiple files, thereby reducing potential for mistakes.

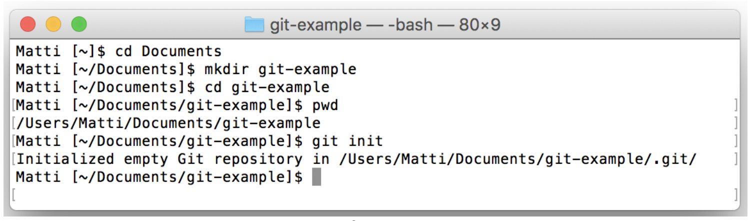
project's life cycle, but to most clearly show its use, we begin with an empty project with nothing in it.

Organizing Files and Folders

Implementing reproducibility into the scientific workflow is less time-consuming and effortful if it is planned from the onset of the project, rather than added to the project after all the work has been completed. It is therefore important to organize the project keeping a few key goals in mind (here, we follow guidelines such as the Project TIER recommendations[12]): The files and folders should have easy to understand names (avoid idiosyncratic naming schemes), and the names should indicate the purposes of the files and folders. We illustrate some good and not-so-good practices in Figure 4.

The first step is to create a home folder for the project. This folder should have an immediately recognizable name, and should be placed somewhere on your computer where you can find it. We call the example project `git-example`. All the materials related to this project will be placed in subfolders of the home directory,

[12] http://www.projecttier.org/tier-protocol/specifications/



```
Matti [~]$ cd Documents
Matti [~/Documents]$ mkdir git-example
Matti [~/Documents]$ cd git-example
Matti [~/Documents/git-example]$ pwd
/Users/Matti/Documents/git-example
Matti [~/Documents/git-example]$ git init
Initialized empty Git repository in /Users/Matti/Documents/git-example/.git/
Matti [~/Documents/git-example]$
```

Figure 5. Creating and navigating to a folder, and initializing it as a Git repository.

but for now `git-example` is just an empty folder waiting to be filled with data, code, and documents. Because the folder structure on a computer is easy to think of as a tree, a project's home folder—or any folder that has subfolders—is also known as the *root* directory. In what follows we use the terms home directory and root (directory) interchangeably.

Initializing a Git Repository

Next, you need to create a new folder for the Git repository. First, choose an appropriate folder on your computer (such as `User/Documents/`) where you'd like to create the project. You can either use the system's file navigator (Finder / File Explorer) to create this folder, or use the command line: Navigate to the desired folder by using `cd Documents` to move into the `Documents` folder (assuming it exists in the folder where you currently are). Use `cd ..` to move out of a folder (to its containing folder), if needed. Once you are in the folder where you want to create the project, type `mkdir git-example` to make the `git-example` directory, then `cd git-example` to move into it. You can, of course, also use the point-and-click interface (Finder or File Explorer) to create folders, instead of the `mkdir` command. Once you are in the project's home folder (you can verify where you are by typing `pwd`), you want to turn the folder into a project by initializing Git with the `git init` command. These commands are shown in 5

Instead of screenshots, for the rest of the tutorial we present the commands as follows:

```
$ git init
```

In these code listings, each command is preceded by a `$` symbol. To help remember the commands, we recommend simply typing them out, instead of copy-pasting. The `git init` command initializes the folder as a Git repository, and the only change so far has been the addition of a hidden `.git` folder inside `git-example` (and possibly a `.gitattributes` file. Users can ignore these hidden files and folders, however they can be shown with the `ls -la` command.) Now that the folder is initialized as a Git repository, Git monitors any changes within it, and allows you to add and commit these changes (Figure 3.)

Adding a File to Git

Every project (repository) should contain a brief note explaining what the project is about and who to contact. This note is usually called a readme, and therefore our first contribution to this project will be a README file (this file is so important that it has become standard practice to write it in capital letters). The README file should be a plain text `.txt` file (i.e. not created with Microsoft Word) that can be read with a simple text editor[13]. You can create this file with any text editor (e.g. the TextEdit and Notepad apps which allow for creating plain text files are installed by default on Mac and Windows OS respectively). We created the file with some text in it, and it can now be seen in the `git-example` folder, either by using the system's graphical file viewer or with the `ls` command. Because we added this file to a Git repository, Git is also aware of it. To see what files have changed since the last status change in the repository (there clearly has been only this one), you can ask for Git's **status** (in subsequent code listings, output is printed without preceding characters):

```
$ git status
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README
nothing added to commit but untracked files present (use "git add" to track)
```

The relevant output returned from executing this command is the "Untracked files:" part. There, Git tells the user that there is an untracked file (`README`) in the repository. To start tracking changes in this file, we **add** it to Git's staging area (Figure 3) by using the command `git add` followed by the filename (`README`), or `.` which is a shortcut for adding all files with changes.

```
$ git add README
```

We have now added this file to the staging area, and if we are happy with changes to the file's status, we can **commit** the file to Git's history. Commits are essential Git operations: They signify meaningful changes to the repository, and can be later browsed and compared to one another. As such, it is helpful to attach a small message to each commit, describing why that commit was made. Here, we've created a README file, and our commit command would look as follows:

[13]Plain text has many advantages over proprietary file formats such as Microsoft Word's .docx files. Briefly, plain text is both human and computer readable (.docx are unreadable without a copy of Microsoft Word), is both forward and backward compatible (there will always, and has always been, software capable of reading it), and plain text takes very little space. The file extension of plain text doesn't matter much, but we recommend using either `.txt` because it is widely recognized, `.md` for markdown syntax, or in the case of a readme file, simply not using any file extension.

```
$ git commit -m "Add README file."
```

The quoted text after the `-m` argument is the *commit message*. Entering this command to the command line returns a brief description of the commit, such as how many files changed, and how many characters inside those files were inserted and deleted.

Keeping Track of Changes with Git

The `git-example` project (or rather, Git) now keeps track of all and any changes to README. To illustrate, you can change the text in the README file with a text editor, save the file, and then ask for `git status` on the command line:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   README
no changes added to commit (use "git add" and/or "git commit -a")
```

Git can tell that the README file has been modified since the last commit. It is often useful to know exactly *how* a file has changed, before committing it. To view differences to a file not yet committed, use `git diff file`. It shows changes within `file`, line by line, highlighting removals with red and added lines with green. Of course, if you have just been editing a file, you know what the changes are. Once you are happy with the changes, you can repeat the add and commit steps from above to permanently record the current state of the project to Git's history (below we use the `.` shortcut for adding all files with changes[14]):

```
$ git add .
$ git commit -m "Populate README with project description."
```

What Does Git Know?

The real importance of these somewhat abstract steps becomes apparent when we consider the Git **log**. To reveal the commit log of your repository, call

[14]For the `.` shortcut to work, you must currently be in the project's root directory. If you are not in the root directory, you can use `-A` shortcut instead; however, we recommend that you always ensure that you are in the project's root directory when running any Git commands.

```
$ git log
```

The main outputs of this command show that each commit is identified with a unique hash code (long alphanumeric string), which we can use to call for further information (see below); an author; a date and time; and a short commit message. Executing `git log` on our example project at this stage returns this:

```
commit 60cbe5c9b4a78e500314f791080381030577a035
Author: Matti Vuorre <mv2521@columbia.edu>
Date:   Tue Jun 13 17:20:27 2017 -0400
    Populate README with project description.

commit 16c475023ecbc99446164187eeaaab10647ac550
Author: Matti Vuorre <mv2521@columbia.edu>
Date:   Tue Jun 13 17:14:14 2017 -0400
    Add README file.
```

To see what exactly changed in the last commit (latest commits are at the top), you can call `git show` with the commit's hash code (only relevant parts of output shown below):

```
$ git show 60cbe5c9b4a78e500314f791080381030577a035
Author: Matti Vuorre <mv2521@columbia.edu>
Date:   Tue Jun 13 17:20:27 2017 -0400
    Populate README with project description.
diff --git a/README b/README
--- a/README
+++ b/README
@@ -0,0 +1,2 @@
+# Example Git Project
+This example project illustrates the use of Git.
```

This output is a detailed log of all changes in that commit. From top, it lists the author, the message, and then the commit's "**diff**" (i.e. what differs in the new version vs the old version of the file.) The current diff shows that one file received two additional lines of text (the part wrapped in @ symbols), and then the additions themselves (the lines prepended with +s).

Although you have now seen the fundamentals of using Git to track the states of (and therefore changes to) a repository, this contrived and overly simplistic example doesn't allow full appreciation of the benefits of using Git for version control. To better illustrate Git's functioning, we now fast-forward in the hypothetical example project to a stage in the project where more files and materials have been created.

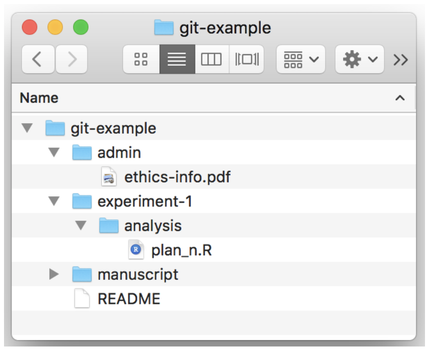


Figure 6. A project with more components.

(Slightly More) Advanced Git

For example, after working for a while on the project, you could have added two files inside well-organized subfolders of the project. At this point, the project could look like Figure 6.

Running `git status` in this folder tells that there are two new files (possible empty folders are ignored). That is, since the previous commit, two files have been added to the project; one a .pdf file with some administrative information (`ethics-info.pdf`), the other one is an R (R Core Team, 2017) script file for a pre-planned power analysis (`plan_n.R`). You would probably like to track any changes to the power analysis script, but the ethics information file isn't something that you need to keep track of—at least for this example. It would be laborious to constantly keep ignoring it when committing changes to Git (especially if you'd like to add multiple files simultaneously with `git add .`) Git has an elegant solution to specifying which files to keep track of. Because by default all files are monitored, Git uses a special file for instructing which files are to be *ignored*.

Make Git ignore files. To make Git ignore files, you simply add a plain text file called `.gitignore` to the home folder of the repository. You can use any text editor to create this file. Each row of this file should specify a file or a folder (or a regular expression) that Git should ignore. In the current example you could make Git ignore the `admin/` folder entirely, and any file with the .pdf extension inside `manuscript/`. The example `.gitignore` file would look like this (lines beginning with # are comments, and only read by humans):

```
# Ignore everything inside the 'admin' folder
admin/
# Ignore all .pdf files in the 'manuscript' folder
manuscript/*.pdf
```

Re-running `git status` now only shows the `plan_n.R` file (and the newly created `.gitignore` file, which is also under version control, naturally.)

Because there are now two untracked files that are not specified to be ignored in `.gitignore` (`.gitignore` and `plan_n.R`), and you usually should aim to maintain a clean commit history for the project, you can create two separate commits: One for the `.gitignore` file, and one for the power analysis file.[15]

```
$ git add .gitignore
$ git commit -m "Added .gitignore file"
$ git add .
$ git commit -m "Completed power analysis"
```

After this last commit you can, at any time in the future, come back to this commit with `git log` or `git show` and see what was inside the newly created power analysis file when it was first created. For instance, if new information suggests that you should change the assumed effect size in the power analysis, you can simply edit and save the file, then add and commit the changes to Git with a helpful message that logs this important event in Git's history.

This possibility of "rewinding history" is especially useful for files that might go under multiple revisions (manuscripts, analysis files), or if you are interested when and in what order the files were created—and who created or changed them. One might consider committing a power analysis file to Git as a small personal pre-registration of a part of the research plan.

Try a new feature. We often find that making some changes to a project didn't have the desired effect: The manuscript ended weaker or the analysis didn't work anymore. Git allows great flexibility in trying new features, then undoing the changes[16]. Starting with an empty staging area, you could start modifying a file (e.g. `plan_n.R`), and after a while realize that the changes were not good. At this point it is common to press "Undo" in the text editor, but if the file has been saved multiple times or multiple files have been changed, it is difficult to get to the starting point by simply using the "Undo" button. Instead, with Git you can **checkout** the file's previous version from history. To undo all changes to `plan_n.R` (since the last commit), run

[15]It is entirely up to the user to decide what to commit and when. However, it is best practice to commit often while making incremental changes. Each commit should aim to solve one problem, introduce one new idea, or—more generally—do one thing. This way, when the commit history is reviewed later, it is easy to find and come back to a specific change.

[16]A particularly powerful approach for trying new features is **branching**: The project can be duplicated to a new branch and modified, then merged back to the main branch after work on the new feature is complete—or the new branch can be discarded if the work ended unsatisfactory. Branches are outside the scope of this tutorial, for more information see the Git website (https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell).

```
$ git checkout experiment-1/analysis/plan_n.R
```

Notice that you have to write the full path of the file (relative to the project's root, see Figure 6) so that Git knows precisely which file you want to checkout from history. With these example operations, we have discussed the main Git operations as outlined in Figure 3: Make changes to files, **add** them to the staging area, **commit** to history; **checkout** from history to undo changes. Although these operations are enough to maintain a Git repository, we have only scratched the surface of Git's functionality.

Undo committed changes. Another common scenario is one where a user makes changes to a file, adds the changes to the staging area, commits them to Git's history, and only then realizes that the changes weren't good. If you have committed changes to a file, and would like to revert back to an older version of the file, you could **checkout** the file's earlier version, and then commit the older version.[17] For example, suppose you have made bad changes to a file called `file.txt`, and committed the changes to history, and would then like to undo the bad changes by reverting to an older version of the file. First, view the history with `git log` (here we use the `--oneline` argument to make the output more concise):

```
$ git log --oneline
4c64f11 Bad changes to file.txt
039d6ff Good changes to file.txt
a73f2ec Add file.txt
```

Recall that `git log` returns the most recent changes at the top, and notice that the `--oneline` argument has also made the commit hash codes shorter and thus easier to read and write. Here we can see that commit `039d6ff` has a good version of the file, and subsequent changes in commit `4c64f11` were bad (you would of course not commit "bad changes", but here the message is informative for clarity). To revert `file.txt` to it's good state in commit `039d6ff`, you can use use `git checkout hash filename`, which here would be:

```
$ git checkout 039d6ff file.txt
```

Now asking for `git status` reveals that `file.txt` has been modified in the working directory (its current state is as it was in commit `039d6ff`). You can now add and commit these changes with `git add file.txt`, then `git commit -m "Undid bad changes to file.txt"` (type a commit message suitable for your situation).[18] `git log --oneline` would then show:

[17]For more information on undoing changes, see https://www.atlassian.com/git/tutorials/undoing-changes.

[18]If, for some reason, you preferred the latest version after all, you can undo the revert process by `git checkout HEAD file.txt`, instead of adding and committing the older version. This function checks out the current state ("`HEAD`") which contained the bad changes.

```
$ git log --oneline
bcbb123 Undid bad changes to file.txt
4c64f11 Bad changes to file.txt
039d6ff Good changes to file.txt
a73f2ec Add file.txt
```

This operation of checking out earlier versions is very useful not only for undoing changes, but for viewing older versions of files as well. However, if you would only like to view past states of the project, instead of reverting / undoing to earlier states of particular files, you can checkout an earlier version of the entire repository, as explained below.

Going to an earlier version of the project. To return to an earlier state of the project, you can use the `git checkout` command. For example, the display above shows that in commit `a73f2ec`, you had added `file.txt`. If you would like to see the project at that commit, type `git checkout a73f2ec`. This command instantly checks out the file(s) at that point in history, and places them in the working directory where you can view them. This is very helpful if, for example, you would like to quickly run an earlier version of a statistical analysis, which may depend on multiple files. After you have viewed this old version of the project, you can return to the current version with `git checkout master`.

Both of these operations—checking out an earlier version of a file or of the entire project—are "safe" in the sense that your project's history won't be affected. However, checking out an earlier version of a specific file changes the current state of the project (the current version of the file is temporarily overwritten with the old version), so it is good practice to carefully keep track of the current version of your file before making further commits.

Collaborating

The true advantages of using Git become apparent when we consider projects with more than one contributor. For example, consider a project where data is collected at multiple sites, and the data files are then saved onto a central server, or shared through a service that automatically merges files from multiple sources (such as the popular Dropbox service). If two or more sites accidentally save a data file with the same name (e.g. `data-001.csv`), and these changes are automatically merged, the later file will simply overwrite the earlier file. Disaster!

Alternatively, consider a data analysis where two or more people work simultaneously on some complicated analysis, and share their work using a similar system as in the above example. If user A and B are making changes to the same file and user B saves the file, user A's version of the file will be overwritten. Disaster!

Git and other VCSs, on the other hand, were specifically designed to allow (and facilitate) multi-site collaboration on arbitrarily complex projects: Microsoft Windows—which we assume is a fairly complex and collaborative project—is developed

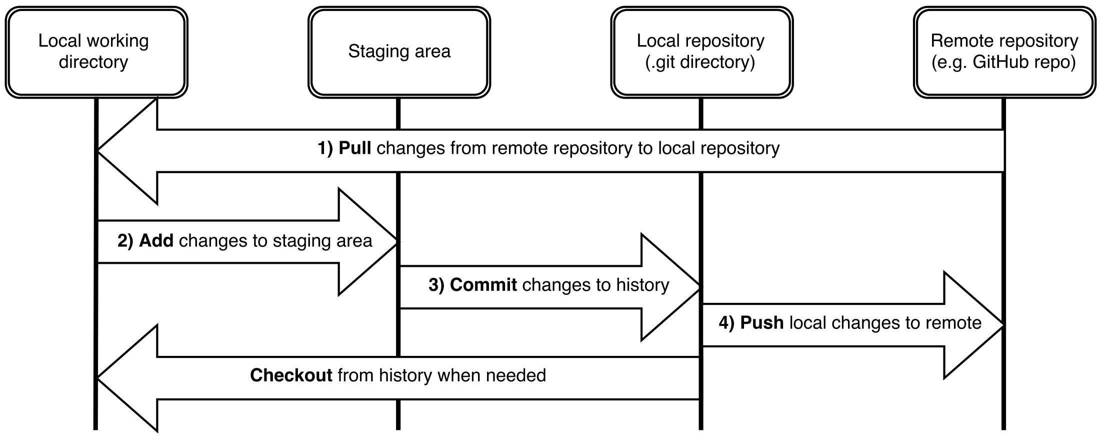

Local working directory
Staging area
Local repository (.git directory)
Remote repository (e.g. GitHub repo)
1) Pull changes from remote repository to local repository
2) Add changes to staging area
3) Commit changes to history
4) Push local changes to remote
Checkout from history when needed

Figure 7. A diagram illustrating the typical collaborative Git workflow with a remote repository (e.g. GitHub). Verbs indicated in bold text are Git operations and explained in detail in the main text.

on a Git platform (Harry, 2017). We believe that Git can be especially helpful in scientific collaboration, a topic to which we turn next.

There are many ways in which a team could collaborate on a Git project; here we focus on a common one, called "Centralized Workflow", where one virtual copy of the project is considered central, and all contributors interact with this central copy, instead of directly with each other.

Overview of the Centralized Git Workflow

In this workflow, a central project hosted on a research team's server, or an online service like GitHub (https://github.com/), is used to send and receive information from and to local projects. Because the word "project" can now refer to virtual copies of projects, they are instead commonly referred to as **repositories**. First, one user creates this central repository, then other users **clone** local copies of it (see Table **??**). Contributors, including the one who created the central repository, then work on their local projects as detailed above, and after committing, **push** their changes to the central repository. To get changes which other users have pushed to the central repository, contributors **pull** changes from it.

Setting up a centralized Git repository on a research team's private server is relatively straightforward, but because the details vary from team to team, here we illustrate the Centralized Workflow using GitHub.

GitHub

GitHub is one of the 100 most popular website worldwide, and hosts over 60 million software projects with a total of over 20 million users (https://github.com/about). We chose GitHub for collaboration because it is already the de-facto standard in VCS collaboration among scientists, it offers free private repositories (see below), and repositories from GitHub can easily be connected to projects hosted on the Open Science Framework (https://osf.io).

Anyone wanting to use GitHub must first create a free user account at https://github.com. After the user account has been set up, and the user has logged in, creating a new repository at GitHub is self-evident (there is a big green button labeled "New repository"). You must create the GitHub account with the same email address as you used above when configuring your local Git (or re-configure your local Git to use your the email address that you used to register for GitHub).

Create a new GitHub repository. Although we describe how to create a new repository, the goal here is to create an online "remote" repository for the running example within this tutorial, so that you can "push" your current local repository to the GitHub remote. Because of this, on GitHub, do not check the "Initialize this repository with a README" (you already created one in the local repository if you followed the steps above on your own computer), and do not add a .gitignore or a license. For the current example, after clicking "New repository", you are asked to enter a name for the repository, which can be anything. For consistency we call the GitHub repository `git-example`—the same name as our local project folder. After entering the name, click "Create repository". The next step is to link the new remote to the local repository, using the following commands (they are also visible on the website that opens after you have created the repository on GitHub):

```
$ git remote add origin https://github.com/username/reponame.git
```

Where `username` and `reponame` are the user's GitHub username, and the GitHub repository name. The correct address is visible in the GitHub page that opens after creating the repository. Once the connection is set up, we can **push** local changes to the GitHub remote:

```
$ git push -u origin master
```

The `-u origin master` arguments are only required for the first push, as they set up the connection. Running this command will send your local repository to the GitHub repository. For pushing changes collowing this initial push, simply type `git push` after adding and committing locally. You have now created the remote central repository, and other users can start contributing to it.

Contributing to a central (GitHub) repository. Once another team member has set up Git on their own computer, and signed up for GitHub, they need to **clone** the remote GitHub repository onto their local computer. To clone a repository, the new user must first navigate to an appropriate location on the computer where they would like to save the project on their computer, for example their `User/Documents` folder. Once the appropriate location is found, cloning will create a new folder for the repository inside this folder. Use `git clone` to clone a repository from a URL:

```
$ git clone https://github.com/username/reponame.git
```

To find out the correct URL to enter to `git clone`, you can navigate your web browser to the repository's GitHub address (e.g. https://github.com/mvuorre/reproguide-curate for this tutorial's repository) and click the big green "Clone or download" button; the complete address is in the text box.

New contributors can then work on their local copies as detailed in earlier parts of this tutorial; making changes, adding, committing, and pushing. After committing their changes, they can update the status of the central Git repo by pushing their changes to it:

```
$ git push
```

You can also clone GitHub repositories if you are not aiming to contribute to a project. Hosting projects on GitHub is attractive because other people can easily clone (download) your work to build on, and contribute to, your work.

Obtaining other's changes from the central repo. Just as you must manually push your own local changes to the remote repository, you must also obtain others' changes by **pulling** them from the central repo. Pulling is indicated as the first step in the collaborative workflow in Figure 7, because it is important that you start working on the most up to date version of the project (e.g. you don't want to reinvent the wheel or make unnecessary conflicting changes). Before starting to work on your proposed changes, pull the remote changes with:

```
$ git pull
```

The way in which users and their local repositories interact with the central repository by pushing and pulling is the cornerstone of collaboration on GitHub, and thoughtful use of these commands allows for complex workflows without any important code (data, ideas in manuscript, analysis code) ever being overwritten. However, there is no automatic way for a computer to tell what changes to prioritize: If two or more users have worked on the same code and then attempt to push their changes, it is possible that they have made changes which conflict with each other. If this happens, there is no need to worry; you simply need to know how to resolve the conflict.

Resolving conflicts in collaborative work

Many different types of conflicts may appear in collaborative work, such as multiple users creating files with the same name but different content, or multiple users working on the same code (recall that we use the word code to mean any text written on a computer, e.g. text in a manuscript) and creating changes that conflict with each other. We use this latter situation as an example to explain how to resolve conflicts in collaborative work.

Consider the following scenario. Two collaborators, User A and User B, are working on the same project (they collaborate on a repository on GitHub, with local repositories as detailed above.) At some point, they might be working on the same file—such as writing a manuscript together—and find that they have made changes that conflict with each other. More formally, let's assume that both users are making changes to a file, and User B happens to add and commit his changes locally and push them to the central repository before User A does. When User A then attempts to push her changes to the local repository—and the changes are incompatible with User B's changes—a **conflict** will happen. That is just a natural consequence of two individuals working simultaneously on the same idea, and then writing different code in the same location in the file. When this happens, user A needs to first integrate the latest version of the project from the central repository to her local project, such that it reflects both collaborators' edits, and then push the new "merged" version to the central repository. Let's look at what this workflow entails in a little more detail.

First, let's assume that a collaborator (User B) has made changes to the `README` file in the `git-example` project and pushed the changes to the central repository. For brevity, we only show the first few lines of this file:

```
# Example Git Project
Hello world!
This example project illustrates the use of Git.
```

At the same time, User A might have changed her local version of the `README` file to look like this:

```
# Example Git Project
Here are some changes.
This example project illustrates the use of Git.
```

If User A now commits the changes locally, and attempts to push the changes to the central repository, an error will appear

```
$ git add .
$ git commit -m "Some meaningful changes"
$ git push
```

```
error: failed to push some refs
hint: Updates were rejected because the remote contains work that you
do not have locally. This is usually caused by another repository
pushing to the same ref. You may want to first integrate the remote
changes (e.g., 'git pull ...') before pushing again.
```

As is usually the case, Git also returns a hint on what to do, which you can follow to succesfully resolve the conflict. Git suggests that User A first integrates the remote changes: She needs to first obtain the latest version of the file(s) from the central repository, add the proposed changes to the latest version of the file (the one containing User B's changes), and push that version. She can first obtain the latest changes from the central repository:

```
$ git pull --rebase
```

You can `git pull` without the `--rebase` argument, but that would create an unnecessary commit message and lead to a slightly different workflow[19]; we recommend using the `--rebase` argument. At this point, Git is in rebasing mode, allowing User A to resolve the conflict before pushing her changes. Running `git status` returns (only relevant output shown):

```
$ git status
rebase in progress; onto bada506
You are currently rebasing branch 'master' on 'bada506'.
  (fix conflicts and then run "git rebase --continue")
Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
    both modified:   README
no changes added to commit (use "git add" and/or "git commit -a")
```

Git again shows a helpful (if not rather jargony) message; the relevant point is that the repository is in "re-basing" mode (technically, you are re-basing your current version (`master`) onto User B's latest contribution, identified with `bada506`). The next step is to manually edit the conflicted file to reflect both User A's and User B's changes. Of course, the user who is doing the re-basing (i.e. merging or resolving the conflict) may simply decide to edit the file to reflect only her changes, completely rejecting User B's changes. When viewed with a text editor, the conflicting `README` file now looks like this:

[19]For more information, see https://www.atlassian.com/git/tutorials/comparing-workflows.

```
# Example Git Project
<<<<<<< HEAD
Hello world!
=======
Here are some proposed changes.
>>>>>>> Some meaningful changes
```

The first line of the file was identical across the two Users' versions of the file, and therefore remains the same. However, after the first line, there is first a line (`<<<<<<< HEAD`) indicating that what follows are the to-be-integrated lines of text. Anything after this line up to the `======` line is the to-be-integrated text from the central repository. We can see that this is simply the line of text that User B created ("Hello world!"). After the separating line (`======`) are User A's local changes, followed by those changes' commit message ("Some meaningful changes") prepended with a `>>>>>>>`. User A can then edit this file however she chooses, using the tags to help her see which are her lines of code (text), and which are User B's. For example, the result might look as follows:

```
# Example Git Project
Here are some proposed changes: Hello world!
This example project illustrates the use of Git.
```

Then, User A can save the file and add the changes with `git add README`. Importantly, Git is in rebasing mode (which can be aborted with `git rebase --abort` to reject the central repository's changes and return User A back to her latest local version), and therefore User A should not commit, but instead complete the "re-basing" with `git rebase --continue`.

```
$ git rebase --continue
Applying: Some meaningful changes
```

This command returns a reminder telling User A which local commit is being applied on top of the changes she pulled from the central repository, and then returns Git to it's normal mode from rebase mode. The final step is then to use `git push` to send the local changes to the central repository.

How these potential conflicts appear depends on how users collaborate with one another, and a detailed explanation of all potential scenarios is outside the scope of this tutorial[20]. Most importantly, even in the event of conflicts, all committed changes are saved in Git's history and can be retrieved, so experimenting with different approaches to resolving conflicts is safe.

[20]Covering all different types of file conflicts is outside the scope of this tutorial. Although the instructions provided herein will help in most common use case scenarios, readers can refer to the following websites for more information: https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/ and https://www.atlassian.com/git/tutorials/

Private or public collaboration?

By default, all GitHub repositories are public: Anyone with an internet connection can use their web browser to inspect the contents of your repository, or even clone it to their computer. This may sound unfamiliar to researchers used to working more privately, and clearly necessitates planning and thought with respect to issues such as data privacy and sharing sensitive materials. However, for many projects—including the writing of this tutorial—we see very few downsides to working "in the open".

There are two alternatives to working in a public GitHub repository: One, which we won't cover here, is to not use GitHub but instead place the central repository on the research team's shared but private server. The second option is to make the repository private on GitHub (this can be done when the repository is first created or afterwards by clicking Settings on the repository's website). Private repositories, and their contents, are only accessible to invited team members, and are therefore ideal for small teams who would like to work without revealing their master plans to the public just yet. For example, you might initially choose to work in a private repository, and only make it public once you feel the material is mature enough for public consumption.

To make a GitHub repository private, navigate to the repository's website with a web browser, and click "Settings", then "Make this repository private". Once one user has set the central GitHub repository to private mode, anyone wishing to clone, push, pull, or view the repo must provide their GitHub username and password. Only if they match an invited team members username and password can the user access the repository.

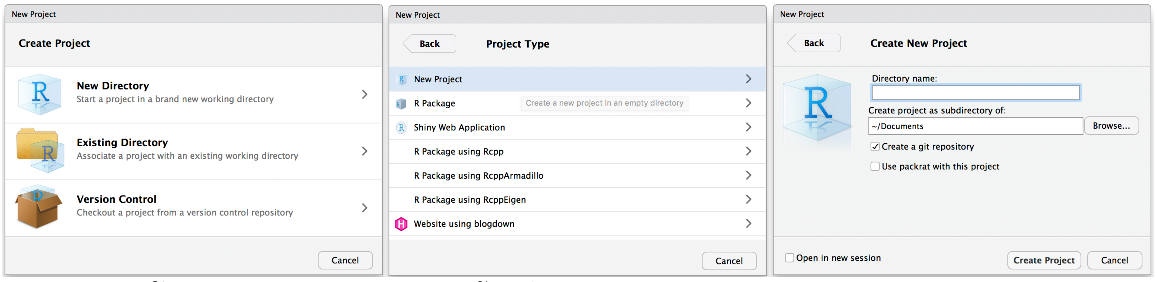
At the time of this writing, GitHub offers five private repositories for free[21]. To obtain more private repositories, you need to register for a paid account.

Using Git with a Graphical Interface

Above, we have detailed how to use Git from the computer's command line, but not all users are comfortable—although there is no reason not to be!—with this mode of interacting with their computers. We chose to introduce Git from the command line because eventually users might want to use it for more complicated operations (and it is required for setting the user's information). However, there are various graphical user interfaces (GUIs) available for Git which allow point-and-click interacting with Git.

comparing-workflows. You can also resolve conflicts on GitHub (https://help.github.com/articles/resolving-a-merge-conflict-on-github/), and the GitHub customer service is very responsive to users' help requests, which can include questions on code conflicts.

[21]To obtain the free repositories, fill out the request form at https://education.github.com/discount_requests/new.



Create project as subdirectory of: ~/Documents

✓ Create a git repository

Use packrat with this project

Figure 8. Creating a project in R Studio.

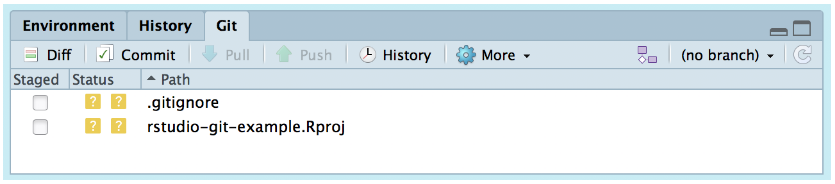
R Studio

In this section, we introduce how to use the popular R Studio IDE (RStudio Team, 2016) for the R programming language (R Core Team, 2017) for interacting with Git. An IDE is an interface that bundles together many necessary features of software development—some users may be familiar with R Studio for conducting statistical analyses. Version control is an essential feature of software development, and R Studio provides a good GUI within its IDE for controlling Git.

For this example, we will begin a completely new project using only R Studio, and assume that readers are familiar with the concepts from the above tutorial. R Studio is a free and open source IDE, works on Windows, Mac, and Linux operating systems, and can be downloaded from the project's website at https://www.rstudio.com/products/rstudio/download/. Once downloaded and installed, it can be used for accessing the R programming language, and for many operations involved in curating research assets, including a GUI for Git.

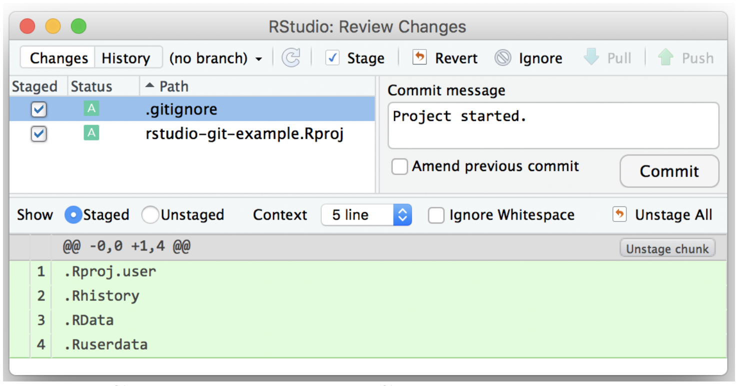
Managing projects (and Git repositories) with R Studio is centered on the idea of R Projects. To start a new project with R Studio, open the R Studio application, and click File -> New Project. This brings up a dialog (left panel in Figure 8) asking whether to create a project in a new directory, existing directory, or checkout an existing project from a version control repository. Here, we create a new project in a new directory, and choose "New Project" in the following screen (middle panel in Figure 8). Then we'll give a name to the project's home folder and choose where to save it on the computer. Importantly, we'll also check the "Create a git repository" box (right panel in Figure 8), which will automatically set up a new repository for the project (provided you have set Git up as detailed above). Clicking "Create project" creates the folder in the specified location, and two files inside the project's main folder.

One of these files is `.gitignore` which we covered above. The other file is an `.Rproj` file, which indicates that the folder is the home folder for an R (Studio) project. Users don't interact with this file directly, but if opened, it is a plain text file containing the project's settings (these can be modified through Tools -> Project Options in R Studio).



Staged	Status	Path
☐	? ?	.gitignore
☐	? ?	rstudio-git-example.Rproj

Figure 9. R Studio's Git tab for a newly created project.



Staged	Status	Path
✓	A	.gitignore
✓	A	rstudio-git-example.Rproj

Commit message: Project started.

```
@@ -0,0 +1,4 @@
.Rproj.user
.Rhistory
.RData
.Ruserdata
```

Figure 10. Creating a Git commit with R Studio.

Using Git with R Studio

Once the R Project has been created, R Studio has a "Git" tab in the top-right panel of the GUI. At first, this tab shows the two new files in the repository, and some buttons with familiar looking names ("Commit", "Push", etc.)

Add and commit changes. To mark this milestone of creating a project, commit all the changes so far (new files, new project) to Git. The workflow is exactly the same as when using Git from the terminal, but there are now some visually appealing helpers. To begin committing these changes, click on "Commit" in the Git tab. This brings up another window where you can select files to add into the staging area (the files in Figure 10 are staged by checking the boxes on the names' left). Once all the desired files (here we chose both) are staged, you can write a short commit message in the appropriate text box, and click the "Commit" button. The text highlighted in green in the bottom part of Figure 10 indicates the lines of text that were changed in the selected file (`.gitignore`). These lines are all green because the file is new to Git and therefore each change is an addition (the files are unimportant R project files which users can safely ignore.)

Add a remote GitHub repository. When creating the R Project, there was an option for creating the project from an existing Git repository. Because we didn't do this, we must now manually instruct Git that this project should have a remote GitHub repository. This is done exactly as above, using the command line functions. After you've created a new repository on GitHub, navigate to the project's folder with the command line shell (or you can click on "More" in R Studio's Git tab, and then "Shell", which opens a new shell window in the correct location), and execute the following commands (but replace `username` and `reponame` with the correct user and repository names):

```
$ git remote add origin https://github.com/username/reponame.git
$ git push -u origin master
```

After executing these commands, you can use all Git and GitHub features from R Studio. This is especially useful because the R Studio IDE offers a complete environment for project management, data analysis, and manuscript preparation with the R Markdown and knitr R packages (Allaire et al., 2016; Xie et al., 2016). Psychologists will be especially interested in the papaja package for creating APA formatted manuscripts (Aust and Barth (2016); the source code of this manuscript, which was prepared with the papaja package, can be found at https://github.com/mvuorre/reproguide-curate).

Git Summary

Table **??** gives the main Git commands and their purported use in roughly the order in which they would be used in a typical workflow. Help on how to use each command is available by appending `--help` to each command (e.g. `git commit --help`). When printed in the command line, some help pages run for several pages, press the spacebar to move to the next page, or `q` to quit looking at the help page. In Box 2 we give links to online materials that we have found particularly helpful in learning Git and teaching it to others.

```
## Error: 'data/git-commands.csv' does not exist in current working directory ('/User
```

```
## Error in eval(lhs, parent, parent): object 'commands' not found
```

Box 2. Further resources for learning Git

- Basic VCS workflow, an infographic explaining how VCS works (https://www.git-tower.com/blog/workflow-of-version-control).
- GitHub's Git cheat sheet is available in multiple languages and contains the most used Git commands (https://services.github.com/resources/cheatsheets/).
- TryGit, an interactive website for learning the basics of Git (https://try.github.io).
- Git + GitHub in an R programming context (http://r-pkgs.had.co.nz/git.html).
- Pro Git book, a complete manual of Git (https://git-scm.com/book/en/v2).

Deleting a Git Repository

Finally, users may sometimes choose to delete their Git repositories. Deleting a project is as simple as moving the containing folder(s) to Trash, which also deletes the Git project (the Git project is contained in a hidden `.git` file in the project's home folder.) If you wish to only delete the Git repository, but keep the project itself, you can delete the `.git` folder. Because the folder is hidden, it will not show up in the default graphical file explorer. You can remove this folder using the command line by navigating to the project's root folder, and using the following command:

```
rm -rf .git
```

We recommend caution with this operation, as it will permanently delete the `.git` folder, which may contain important information. To verify that the folder was deleted, you can list all the files and folders in the current working directory, including hidden ones, with the following command:

```
ls -la
```

To delete a repository on GitHub, use your internet browser to navigate to the repository's GitHub address, click Settings, then "Delete this repository". Be aware that if anyone has cloned this project to their computer, you cannot delete their cloned versions.

Discussion

We have presented an introductory tutorial on using the Git version Control System for curating and collaborating on research assets in behavioral sciences. Although this tutorial includes enough material to get started and manage most operations, Git (and GitHub) is a vast ecosystem with endless opportunities. For example, the concept of "Born open data", where research data is automatically posted online upon collection, is made very easy with the Git + GitHub workflow (Rouder, 2015). Another important feature of this workflow, which we did not cover, relates to disseminating scientific knowledge. The Git + GitHub workflow, especially when used from R Studio, makes it easy for researchers to create accessible documents in .pdf, website, blog post and other formats for facilitating communicating their research.

Some limitations of using Git (and GitHub) are that it requires some initial learning up front, and human intervention at various steps when changes are committed, pushed, etc. Other common collaboration and "versioning" systems, such as Dropbox seem to manage these operations automatically and without requiring any time investment in learning the tools. Why, then, should researchers spend their valuable time in learning to use Git (or other VCSs) instead of alternatives (e.g. Dropbox).

First, Dropbox doesn't save detailed information on who did what and when at each change point in a file's history: Although files' past versions are saved for 30 days (for free account users), it is impossible to tell what exactly changed between versions, and therefore finding the desired version of a file can be particularly difficult. With Git, you can easily `git log` back in time and see what happened at each change-point (especially if you took the time to write informative commit messages.)

Second, we offer that the investment—which may itself be smaller than many might think—to learning Git can end up turning into saved time for more complex projects, because Git never loses track of a project's history, and allows for multiple collaborators to seamlessly work together on the same files. Collaborators don't need to give a file to one person to work on while all others wait for these changes.

Third, the realities of scientific collaboration are too complex to easily narrow down to a model in which files are automatically updated across multiple computers, such as they are when using an "automatic" service like Dropbox. Sharing complex ideas and code across multiple collaborators will ultimately always require some form of human oversight and communication, and this insight has been recognized and implemented by software developers, and Git's tools for merging complex ideas from multiple sources.

Finally, it is important to note that Version Control and backups of one's work are not the same thing. Although Git allows for keeping track of the history of one's project, it is not a physical or virtual backup of that work. Researchers should also maintain best practices in backing up their research assets in multiple physical locations, and possibly online as well.

The challenges to reproducibility are many, and they have only recently received the targeted attention they deserve for the reliability of empirical sciences. Curating research assets and focusing on the scientific workflow is important for ensuring the continuity of one's work and improves efforts for a cumulative and reliable science.

References

Allaire, J. J., Cheng, J., Xie, Y., McPherson, J., Chang, W., Allen, J., … Hyndman, R. (2016). Rmarkdown: Dynamic Documents for R (Version 1.3). Retrieved from https://cran.r-project.org/web/packages/rmarkdown/index.html

Aust, F., & Barth, M. (2016). *Papaja: Create APA manuscripts with RMarkdown.* Retrieved from https://github.com/crsh/papaja

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature News*, *533*(7604), 452. doi:10.1038/533452a

Eglen, S. J., Marwick, B., Halchenko, Y. O., Hanke, M., Sufi, S., Gleeson, P., … Poline, J.-B. (2017). Toward standard practices for sharing computer code and programs in neuroscience. *Nature Neuroscience*, *20*(6), 770–773. doi:10.1038/nn.4550

Harry, B. (2017). *The largest git repo on the planet.* Retrieved June 20, 2017, from https://blogs.msdn.microsoft.com/bharry/2017/05/24/the-largest-git-repo-on-the-planet/

Ihle, M., Winney, I. S., Krystalli, A., & Croucher, M. (2017). Striving for transparent and credible research: Practical guidelines for behavioral ecologists. *Behavioral Ecology*. doi:10.1093/beheco/arx003

Jacobsen, J., Schlenker, T., & Edwards, L. (2012). *Implementing a Digital Asset Management System: For Animation, Computer Games, and Web Development.* CRC Press.

Kidwell, M. C., Lazarević, L. B., Baranski, E., Hardwicke, T. E., Piechowski, S., Falkenberg, L.-S., … Nosek, B. A. (2016). Badges to Acknowledge Open Practices: A Simple, Low-Cost, Effective Method for Increasing Transparency. *PLOS Biology*, *14*(5), e1002456. doi:10.1371/journal.pbio.1002456

Kitzes, J., Turek, D., & Deniz, F. (2017). *The Practice of Reproducible Research: Case Studies and Lessons from the Data-Intensive Sciences.* University of California Press. Retrieved from https://www.practicereproducibleresearch.org/

Leek, J. T., & Peng, R. D. (2015). Statistics: *P* values are just the tip of the iceberg. *Nature News*, *520*(7549), 612. doi:10.1038/520612a

Markowetz, F. (2015). Five selfish reasons to work reproducibly. *Genome Biology*, *16*(1), 274. doi:10.1186/s13059-015-0850-7

McMillan, R. (2005, April 20). After controversy, Torvalds begins work on "Git". *PC World*. Retrieved from https://www.pcworld.idg.com.au/article/129776/after_controversy_torvalds_begins_work_git_/

Munafò, M. R., Nosek, B. A., Bishop, D. V. M., Button, K. S., Chambers, C. D., Sert, N. P. du, … Ioannidis, J. P. A. (2017). A manifesto for reproducible science. *Nature Human Behaviour*, *1*, 0021. doi:10.1038/s41562-016-0021

Nuijten, M. B., Hartgerink, C. H. J., van Assen, M. A. L. M., Epskamp, S., & Wicherts, J. M. (2015). The prevalence of statistical reporting errors in psychology (1985–2013). *Behavior Research Methods*, 1–22. doi:10.3758/s13428-015-0664-2

Open Science Collaboration. (2015). Estimating the reproducibility of psychological.

Science, *349*(6251), aac4716. doi:10.1126/science.aac4716

Peng, R. D. (2011). Reproducible Research in Computational Science. *Science*, *334*(6060), 1226–1227. doi:10.1126/science.1213847

R Core Team. (2017). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Rouder, J. N. (2015). The what, why, and how of born-open data. *Behavior Research Methods*, 1–8. doi:10.3758/s13428-015-0630-z

RStudio Team. (2016). *RStudio: Integrated Development Environment for R*. Boston, MA: RStudio, Inc. Retrieved from http://www.rstudio.com/

Vanpaemel, W., Vermorgen, M., Deriemaecker, L., & Storms, G. (2015). Are We Wasting a Good Crisis? The Availability of Psychological Research Data after the Storm. *Collabra: Psychology*, *1*(1). doi:10.1525/collabra.13

Wicherts, J. M., Borsboom, D., Kats, J., & Molenaar, D. (2006). The poor availability of psychological research data for reanalysis. *American Psychologist*, *61*(7), 726–728. doi:10.1037/0003-066X.61.7.726

Xie, Y., Vogt, A., Andrew, A., Zvoleff, A., Simon, A., Atkins, A., … Foster, Z. (2016). Knitr: A General-Purpose Package for Dynamic Report Generation in R (Version 1.15.1). Retrieved from https://cran.r-project.org/web/packages/knitr/index.html