

# Curating Research Assets: A Tutorial on the Git Version Control System



Matti Vuorre<sup>1</sup> and James P. Curley<sup>1,2</sup>

<sup>1</sup>Department of Psychology, Columbia University, and <sup>2</sup>Department of Psychology, University of Texas at Austin

Advances in Methods and  
Practices in Psychological Science  
2018, Vol. 1(2) 219–236  
© The Author(s) 2018  
Reprints and permissions:  
sagepub.com/journalsPermissions.nav  
DOI: 10.1177/2515245918754826  
www.psychologicalscience.org/AMPPS



## Abstract

Recent calls for improving reproducibility have increased attention to the ways in which researchers curate, share, and collaborate on their research assets. In this Tutorial, we explain how version control systems, such as the popular Git program, support these functions and then show how to use Git with a graphical interface in the RStudio program. This Tutorial is written for researchers with no previous experience using version control systems and covers both single-user and collaborative workflows. The online Supplemental Material provides information on advanced Git command-line functions. Git presents an elegant solution to specific challenges to curating, sharing, and collaborating on research assets and can be implemented in common workflows with little extra effort.

## Keywords

reproducibility, version control, Git, research methods, open science, open materials

Received 6/26/17; Revision accepted 11/14/17

Lack of reproducibility is increasingly being recognized as a problem across scientific disciplines, and journals in a wide range of research areas, including biology (Markowitz, 2015), ecology (Ihle, Winney, Krystalli, & Croucher, 2017), neuroscience (Eglen et al., 2017), and psychology (Munafò et al., 2017), have published calls for changing the scientific workflow to enhance reproducibility. Studies suggest that one specific challenge to reproducibility is the ways in which researchers organize, curate, share, and collaborate on their research assets (Vanpaemel, Vermorgen, Deriemaeker, & Storms, 2015; Wicherts, Borsboom, Kats, & Molenaar, 2006). By *assets*, we mean, for example, stimuli, data, and code used to support a research article's conclusions.

Fortunately for the empirical sciences, challenges related to curating materials across time, space, and collaborators have been solved to a high standard by software referred to as version control systems (VCSs). In this Tutorial, we introduce a popular VCS called Git and illustrate its use in the scientific workflow with a hypothetical example project. We show how to use Git with a graphical user interface (GUI) in the RStudio program (RStudio Team, 2016). We also show how to use Git with the online service GitHub for collaborative

workflows. Using Git (and GitHub) will streamline workflows and help researchers stay better organized, and thereby facilitate reproducibility.

## Version Control Systems

Consider a scenario in which several researchers are collaborating on a manuscript that reports results from an analysis of a data set. In a typical workflow, one person might format the raw data in a specific way to fit a particular statistical model and then write a draft of the manuscript, in the process creating three files: a spreadsheet with the data, a file with the computer code for the analysis, and the manuscript. If a collaborator then decided to use another statistical model, which required the data in a different format, and then edited the manuscript, he or she would create three more files: new data spreadsheet, new file with the code for the analysis, and revised manuscript. This cycle

### Corresponding Author:

Matti Vuorre, 406 Schermerhorn Hall, 1190 Amsterdam Ave. MC 5501,  
New York, NY 10027  
E-mail: mv2521@columbia.edu

**Box 1.** How Git Facilitates the Scientific Workflow

- Collaborators can share work easily, safely, and in an organized manner.  
Git enforces a common organizational 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. Git projects are shared as a whole, so complex projects with multiple files linking to each other are easy to share. Sharing projects with other researchers is built into Git and can be facilitated with online services such as GitHub. A project that uses Git can be easily copied to GitHub, and other researchers can download the entire project from GitHub onto their local computers.
- Users can try different ways of visualizing data while keeping track of the variants.  
Git saves a file's current version whenever that is requested, so new features can be tested without losing previous versions or increasing the number of files in a project's directories. Any past version can be retrieved from Git's history.
- Multiple collaborators can work on the same files at the same time.  
Collaborators work on the project locally, sending material to and receiving material from a central copy of the project. Git keeps track of who has done what, when, and why (if users add a commit message). Git never loses information or overwrites work unless asked to do so, but allows for collaborators to work on the same ideas simultaneously.

would then be repeated as many times as required; each time more files would be created, and it would become increasingly difficult for the authors to remember which data were paired with which analysis, and which analyses were reported in which version of the manuscript.

With a VCS, the project would contain three files only (one each for the data, analysis, and manuscript). The VCS would keep track of changes to the files and their different versions, therefore removing the need for new files for every new idea or edit. This lack of duplication, in turn, would likely reduce errors in remembering which data file was linked with which analysis, which manuscript version had the correct numerical results, and so forth. Box 1 shows additional examples of how VCSs can improve the scientific workflow. In the Discussion section, we compare Git, as a representative VCS, with other common workflow tools.

VCSs were initially developed for writing code collaboratively,<sup>1</sup> but are increasingly being adopted to enhance workflows outside computer science. To understand why, it is helpful to think of *code* more broadly as any text written on a computer: Manuscripts, statistical analysis scripts, source code of computerized experiments, and even data files are code, or have source code. Code is just plain text written on a computer, and 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). Furthermore, metadata, stimuli, study notes, and other research documents often go through multiple revisions and could therefore benefit from being version controlled.

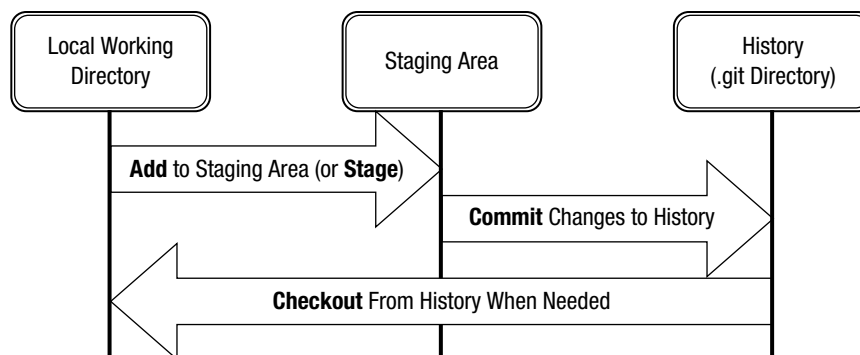
For example, a computerized experiment's source code (text written by humans but interpreted by computers to, e.g., display stimuli to participants) may have multiple authors and go through multiple versions. The problems inherent to keeping track of these versions

and changes, and allowing many authors to contribute (without introducing errors to the experimental program's code), are problems that VCSs were specifically designed to handle. Writing manuscripts collaboratively is quite similar, at least as far as the computer is concerned: Multiple authors write multiple versions of a text document, sometimes needing to inspect previous versions, and the text needs to be merged across these authors. Even data sets can be considered as plain-text code: In most computerized experiments, the output data are numbers and text written into a text file. Researchers would usually not want to see that their raw data files were changed after they were created, and VCSs make it possible to verify that they have not changed because their history is preserved.

The core concept of version control is that contributors to a project keep track of the changes they make to the source code by saving the files they have made changes to and then saving those revised files to the VCS's history. (This process is analogous to saving an intermediate version of a file on a computer's hard disk. Indeed, in order to submit a file to a VCS's history, one must first save it on disk.) The VCS maintains a history of changes to the code between the various versions of the files that were saved to its history and therefore allows users to return to any earlier version by browsing the history. Figure 1 illustrates these concepts by outlining the typical Git workflow. Git and GitHub terminology is explained in more detail in Box 2.

## 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 (<https://subversion.apache.org/>), Mercurial (<https://www.mercurial-scm.org/>), and Git (<https://git-scm.com/><sup>2</sup>). In this Tutorial, we focus on Git because it is increasing in popularity



**Fig. 1.** A diagram illustrating the typical Git workflow. Verbs in boldface are Git operations. In brief, this workflow begins with working in a local directory to make changes to a file in a project (e.g., editing a manuscript) and then adding the changed file to the staging area. Many files can be added to the staging area, if desired. The user can then commit the changes to Git's history. A commit can be accompanied by a short commit message that describes the changes made. When required, files' older versions can be retrieved by checking them out from Git's history. These terms and operations are explained in more detail in the main text. This figure was adapted from Figure 6 at <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>.

within the scientific community and is especially good for scientific collaboration because of its online tools that allow seamless collaboration even for very large research teams. Further, Git is free and open source, and it works on Windows, Mac, and Linux operating systems (among others).

Most computer users are already familiar with creating, copying, and deleting files and folders on their computers using the operating system's default file viewer (Mac's Finder, Windows' File Explorer). Git adds functionality to the computer's file system by making

available a set of commands—executed either from a point-and-click GUI or from the computer's command line—that allow users to keep track of files and their history and to distribute files across multiple computers and users. Git does not move or change the files or folders in any way: Users interact with their files as they would without Git, but instead of creating a new file each time they make important changes, they can use Git to save the file's current state to Git's history; subsequently, they can retrieve each of these versions of the file as needed.

## Box 2. Main Git and GitHub Operations and Terms

**Git repository:** a folder whose contents are tracked by Git. Changes within this folder can be saved to Git's history. Git repositories are located on users' computers and are therefore also called local repositories.

**GitHub repository:** a Git repository hosted on GitHub. GitHub repositories can be set to receive changes from local repositories, so that multiple users can work on the same project by connecting their local repositories to the GitHub repository, which is also called a remote or central repository.

**Initialize a local repository:** an operation that creates a local Git repository on a computer.

**Clone a remote repository:** an operation that creates a local Git repository by copying a remote repository (e.g., one hosted on GitHub).

**Add changes:** an operation that adds a changed version of a file to Git's staging area. The fact that only changes that are added to the staging area can be committed allows for control over what is saved in Git's history. For instance, if a user wants to make changes to file X, but not changes to File Z, part of a project's history, X can be added to the staging area without adding Z.

**Commit changes:** an operation that creates a snapshot of the project's current state by saving changes from the staging area to Git's history. A commit can include a short message describing the commit's purpose.

**Git history:** a list of all the commits made in the repository.

**Checkout:** switch to an earlier version of the project by "checking it out" from Git's history.

**Push committed changes:** an operation that sends changes made on a user's local repository to the central (remote) repository.

**Pull changes:** an operation that brings changes from the central (remote) repository to a user's local repository, to keep it up-to-date with other collaborators' changes.

## Installing Git

Even if you already have Git installed on your computer, it is good practice to install the latest version (as of the writing of this article, Version 2.16.2), which can be downloaded from <https://git-scm.com/download>. Because Git is a stand-alone program, it is easy to install by simply downloading the installer and following the on-screen instructions. If you are a Mac user, the easiest way to install or update Git is to download the installer from <http://git-scm.com/download/mac> and install Git as you would any other program. Similarly, if you are a Windows user, you can download the Git software installer from <http://git-scm.com/download/win> and install Git as you would any other application. If you are a Linux (or other) user, you should look at the instructions on the Git Web site (<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>).

After the Git software has been installed, its functions are available to the computer's users through various Git clients. In this Tutorial, we show how to use Git with RStudio. We present instructions for using Git from the command line in the Supplemental Material available online.

## Disclosures

The Git repository for this manuscript, which includes its source code, can be accessed at <https://github.com/mvuorre/reproguide-curate>. The materials are also stored on the Open Science Framework, at <https://osf.io/txgn8/>. A Git repository for this Tutorial's hypothetical example project can be found at <https://github.com/mvuorre/git-example>. The Supplemental Material discusses using Git (and GitHub) from the command line and can be accessed at <https://github.com/mvuorre/reproguide-curate/blob/master/manuscript/supplement.pdf> or <http://journals.sagepub.com/doi/suppl/10.1177/2515245918754826>.

## Fundamentals of Git

The first operating principle of Git is that your work is organized into independent projects, which Git calls *repositories*.<sup>3</sup> A repository is a folder on your computer that is version controlled by Git,<sup>4</sup> and it can itself contain subfolders. 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 this full control, there is a small set of operations you need to know. As a new user of Git, you will find it helpful to tape a Git-command cheat sheet (<https://services.github.com/on-demand/resources/cheat-sheets/>) on your wall, but note that this sheet contains

many more commands than are needed for the basic use of Git in standard psychology studies.

Briefly, when you work in a Git repository (i.e., make changes to files within it), Git monitors the state of all the files, and when they change, Git knows that they differ from their previously logged state. If you are happy with the current changes, you *add* the changed files to Git's staging area (i.e., you *stage* them). 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 (Fig. 1). 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.

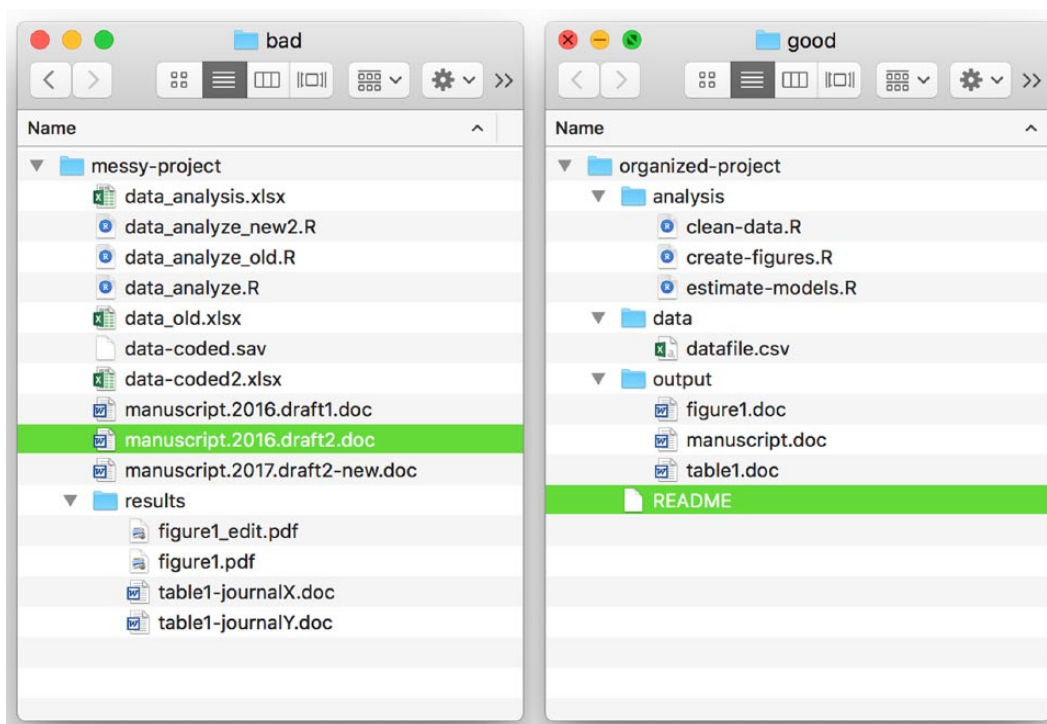
To explain the Git workflow in practice, we now turn to a practical example of working on a hypothetical project using RStudio. Git can be added to a project at any stage of the project's life cycle, but to most clearly show its use, we begin with an empty project.

## RStudio

RStudio (RStudio Team, 2016) is an integrated development environment (IDE) for the R programming language (R Core Team, 2017).<sup>5</sup> It is free and open source; works on Windows, Mac, and Linux operating systems; and can be downloaded from the project's Web site (<https://www.rstudio.com/products/rstudio/download/>). RStudio incorporates tools that are useful throughout the scientific research cycle, including tools for organizing projects, analyzing data, and writing manuscripts. To these ends, RStudio also includes a graphical interface for using Git.

## Creating the example project

Implementing reproducibility into the scientific workflow is less time-consuming and effortful if it is planned from the onset of a project, rather than added to the project after all the work has been completed. One of the early steps that can be taken to facilitate reproducibility is to organize and label files and folders as clearly as possible (here, we loosely follow guidelines such as the Project TIER, n.d., recommendations). For example, files and folders should have easy-to-understand names (i.e., idiosyncratic naming schemes should be avoided), and the names should indicate the purposes of the files and folders. Following these practices will help potential collaborators (and yourself) find files you need quickly and reliably. We illustrate some additional good and not-so-good practices in Figure 2.



**Fig. 2.** Examples of good and bad practices in organizing files and folders. Bad practices (left panel) include having multiple versions of a given file. When there are multiple versions, collaborators may, for example, accidentally use the wrong data for analysis or forget which version they used. It is better to organize a project in subfolders with meaningful names and to have only one file per purpose (right panel).

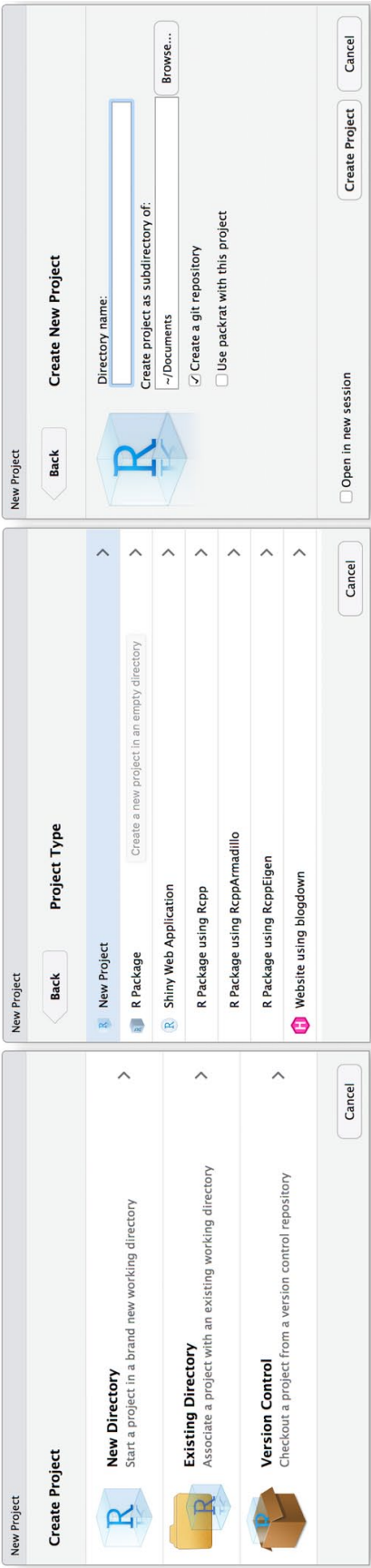
A project's folder should have an immediately recognizable name and should be placed somewhere on the computer where it can be easily found. We call the example project (and therefore its home folder) “git-example.” Because the folder structure on a computer is easy to think of as a tree, a project's home folder—or any folder that has subfolders—is also known as the *root* directory. To start the new project with RStudio, open the RStudio application, and click on “File” and then “New Project.”<sup>6</sup> This brings up a dialogue (left panel in Fig. 3) asking whether you want to create a project in a new directory, create a project in an existing directory, or retrieve (checkout) an existing project from a version control repository. We want to create our example project in a new directory, so after clicking on “New Directory,” we choose “New Project” (middle panel in Fig. 3; old RStudio versions may instead call this “Empty Project”). In the screen that follows (right panel in Fig. 3), we give a name to the project's home folder (“git-example”) and choose where to save it on the computer. We also check the “Create a git repository” box, which will automatically set up a new repository for the project. Clicking “Create Project” then creates the project's main folder in the specified location, as well as two files inside that folder. One of these

files is `.gitignore`, which we discuss in more detail later. The other file is `git-example.Rproj`; the extension indicates that the folder is the home folder for an R (RStudio) project. Users will not interact with this file directly, but it is a plain-text file containing the project's settings (these can be modified by clicking on “Tools” and then “Project Options” in RStudio).

Now that the R project has been created, RStudio has a Git panel in the top right portion of the GUI (Fig. 4). This panel shows the two new files in the repository and buttons for the main Git commands. Because the project's main folder has been initialized as a Git repository, Git will monitor any changes within the folder and allow you to add and commit these changes.

### ***Adding files to Git***

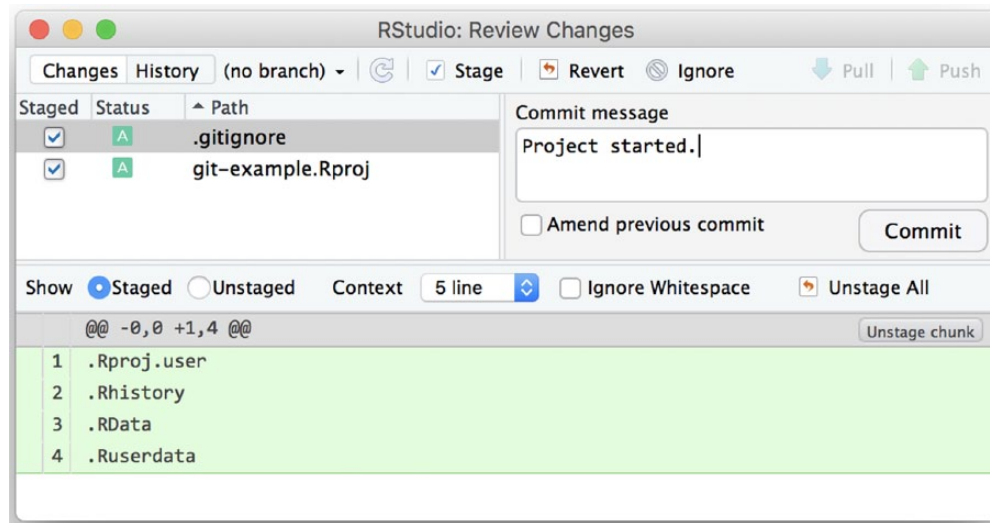
To mark this milestone in creating the project, you can commit all the changes so far to Git. To do this, click on “Commit” in the Git panel. This will bring up another window (Fig. 5), where you can select files to add to the staging area. The two new files in the repository are initially flagged by yellow question marks in the Status column. To add these new files to Git's staging area, select the radio buttons in that column. This files'



**Fig. 3.** Creating a project in RStudio. To create a new project in a new directory, click on “New Directory” (left panel) and then “New Project” (middle panel). The final step is to name the directory and indicate where it should be saved on the computer (right panel). There is also an option to create a Git repository for the project.



**Fig. 4.** Screenshot of RStudio's Git panel for the newly created example project. This panel shows the two files that were created when the project was started and the buttons for the “Commit” and “Diff” commands (“Diff” is used to identify the differences between two versions of a file).



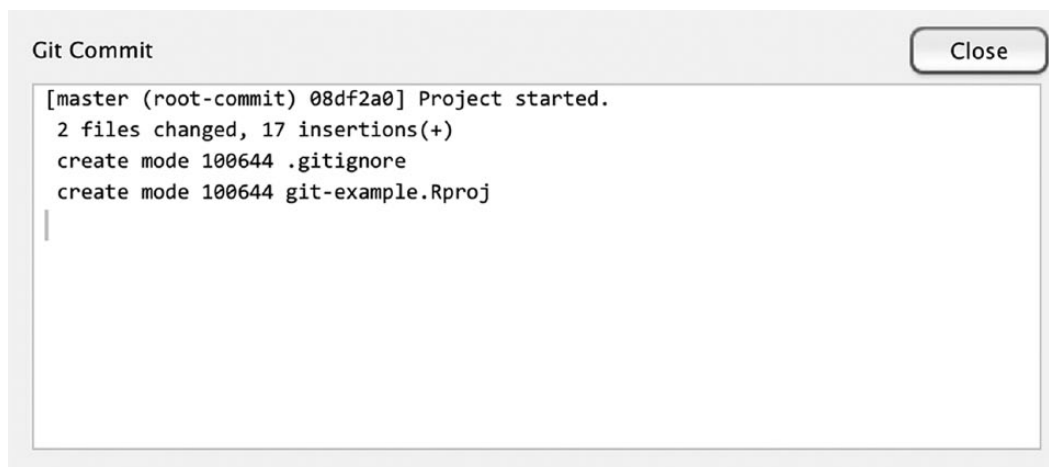
**Fig. 5.** Screenshot showing RStudio's Git panel for the example project after the first two files have been added to the staging area. Note that the bottom section of the window displays the changes in the `.gitignore` file, because it has been selected. An explanatory message has been typed in the "Commit message" box, in preparation for committing the files.

status is now shown as an "A" on a green background (see Fig. 5). The screenshot in Figure 5 shows how this window looks if you then select the `.gitignore` file. The text in the pale-green section at the bottom of this window indicates the lines of text that were changed in the selected file. All the lines of text in this file show up in this section because the file is new to Git and therefore each line is an addition (change) to the file (we discuss the contents of this file in more detail later).

You have now added the files to Git's staging area and can commit the files to Git's history. Before clicking on "Commit," write a short message in the "Commit

message" box, describing what changed and why. These messages will be important when you later browse the history of your repository. After you click on the "Commit" button, a window summarizing the commit's changes will pop up (Fig. 6).

Every project (repository) should contain a brief note explaining what the project is about and whom to contact for more information. This note is usually called a readme file, and therefore your first proper contribution to this project will be a README file.<sup>7</sup> Although it might seem odd to document such trivia when working solo on a project, it will be easier for you and other people



**Fig. 6.** Illustration of Git's description of changes within a commit. This commit (from Fig. 5) changed two files and inserted a total of 17 lines of text. The rest of the information can be safely ignored for the purposes of this example. Click on "Close" to proceed.



(e.g., someone else continuing your project later on) to recall what the project was about if this information is available.

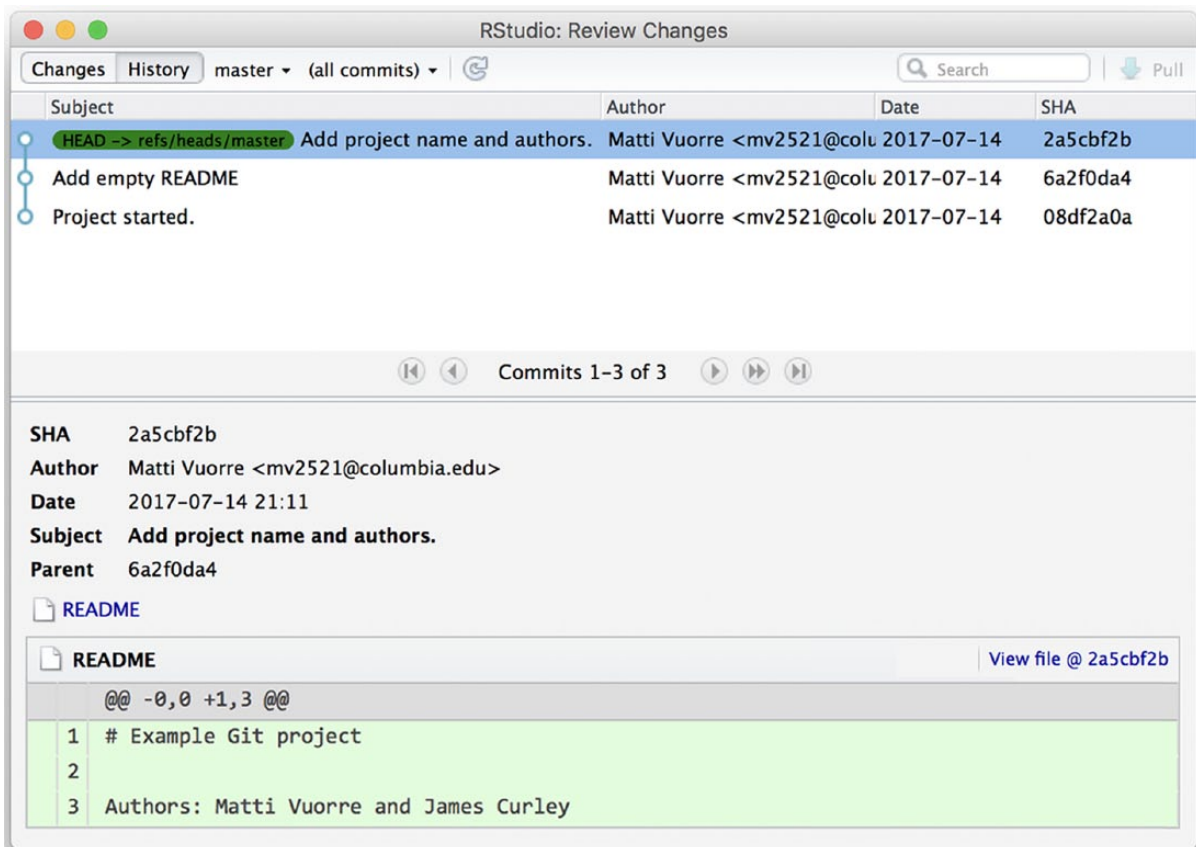
The README file should be a plain-text (.txt) file (i.e., not created with Microsoft Word or similar software) that can be read with a simple text editor.<sup>8</sup> Create this file with RStudio's text editor (click on "File," then "New File," then "Text File"), and save it to the project's root folder. Once you have saved the file in that folder, it will be visible in RStudio's Git panel (with yellow question marks indicating that it is a new file). Once you are happy with the README file's contents, stage the file by checking the radio button, write a commit message, and click on "Commit." This initial version of the README file is now saved in Git's history and can be retrieved later. Notice also that after the commit, the Git panel in RStudio is empty; there are no changes to the repository since the last commit.

### Keeping track of changes with Git

Git now will keep track of all and any changes to the README file and the two other files in the git-example project. For example, if you change the text in the README file with RStudio's text editor and save the

file, RStudio's Git panel will show the README file with a blue "M" (for "modified") flag: Git knows 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, click on "Commit" in the Git panel and select the appropriate file. The bottom section of the panel will then display the new lines of text on a green background and the old lines of text on a red background. That is, if you make changes to a given line, the old version of that line will be shown on the red background, and the new version will be shown on the green background. Once you are happy with the changes, you can repeat the add and commit steps to permanently record the current state of the project to Git's history.

The real importance of these somewhat abstract steps becomes apparent when we consider Git's history for a project. The history contains the exact state of the project at each commit and allows retrieval of previous versions of files. To view a project's Git history in RStudio, click the "History" button in the Git panel. For this example, suppose we have added some information to the README file. The top section of the following screen (Fig. 7) shows each commit's message, author, date and SHA key (a hash code that uniquely



**Fig. 7.** RStudio display showing the Git history for the example project after changes to the README file have been committed. The top section of the display summarizes the history, and the bottom section details the changes in the selected commit.



identifies the commit). The bottom section of the screen shows more details about the selected commit, including the actual changes made to files in that commit. In the current example, the README file received three new lines, shown in the green background in Figure 7.

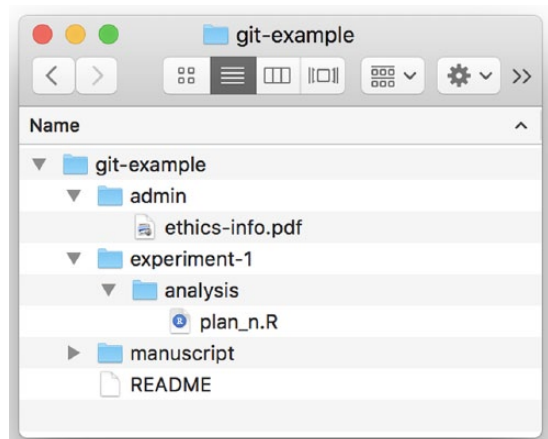
Although you have now seen the fundamentals of using Git to track the states of (and therefore changes to) a repository, this overly simplistic example does not allow full appreciation of the benefits of using Git for version control. To better illustrate Git's functioning, we now fast-forward the hypothetical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

### (Slightly More) Advanced Git

Suppose that after working for a while on the project, you have added two files to it, and the project's home folder looks like Figure 8. Viewing the Git panel in RStudio would reveal that there are two new files (possible empty folders are ignored): One is a .pdf file with some administrative information (ethics-info.pdf), and the other is an R script file for a prospective power analysis (plan\_n.R). Suppose further that you would like to track any changes to this script, but do not need to keep track of the ethics file. Git has an elegant solution to specifying which files to keep track of. Because by default all files are monitored, a special file tells Git which files are to be *ignored*.

#### Telling Git to ignore files

Git uses a special plain-text file in the home folder of the repository to control which files are to be ignored by Git. RStudio creates this file, which is called .gitignore, automatically when you create a new project with Git



**Fig. 8.** Screenshot of the root folder of the example project after an ethics file and script for a power analysis have been added. Note that the .gitignore and .Rproj files are not shown here.

enabled, and you can use a text editor to edit it (i.e., add or remove components to be ignored).<sup>9</sup> Note that the file is *hidden* (by default, not visible in the operating system's file viewer), but can be seen in RStudio's Files panel. Each row of text in this file specifies a file or a folder (or a pattern of characters, referred to as a *regular expression*) that Git should ignore. In the current example, suppose you want Git to ignore the admin folder entirely and also to ignore any file with the .pdf extension inside the manuscript folder. The example file would look like this:

```
.Rproj.user
.Rhistory
.RData
.Ruserdata
admin/
manuscript/*.pdf
```

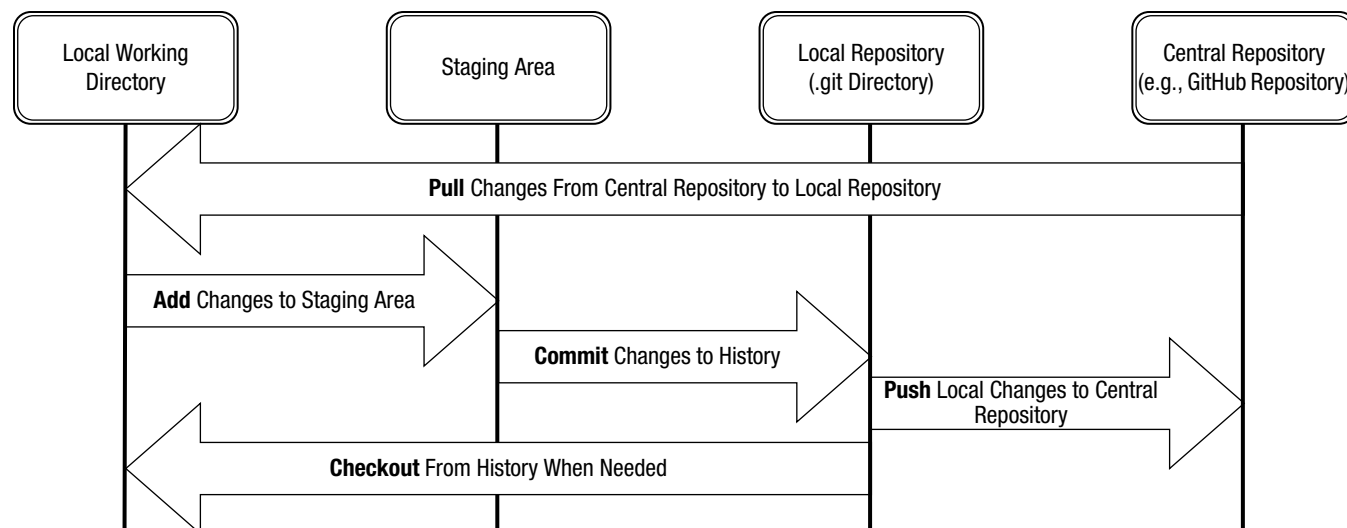
The first four lines are automatically written by RStudio when the project is created, and the fifth and sixth lines specify the files you want to ignore.

After you save these changes to the .gitignore file, RStudio's Git panel shows that the .gitignore file has changed and that there are changes in the experiment-1 folder. To find out which files have changed in that folder, click on the "Staged" radio button next to that folder (see the example of the Git panel in Fig. 4), and the display will show that the only changed file is plan\_n.R. Because there are now two files that are not specified to be ignored in .gitignore (i.e., .gitignore and plan\_n.R), and because you usually should aim to maintain a clean commit history for a project,<sup>10</sup> you should now create two separate commits: one for the .gitignore file and one for the power-analysis file.

After committing the plan\_n.R file to Git's history, you can at any time use Git's history to return to this file and see what was inside at an earlier time. 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, and then add and commit the changes to Git with a helpful message that logs this important event in Git's history. Afterward, you can inspect the file as it was before those changes were made, if required.

#### Accessing files' past versions

This possibility of "rewinding history" is especially useful for files that might undergo multiple revisions (e.g.,



**Fig. 9.** A diagram illustrating the typical collaborative Git workflow with a central repository (e.g., GitHub). As when a central repository is not involved, users work in their local repositories, making changes to files and viewing prior versions as needed. In addition, they push changes from their local repositories to the central repository, so that collaborators have access to them, and pull changes from the central repository to their local repositories, to see their collaborators' work. Verbs in boldface are Git operations and explained in the main text.

manuscripts, analysis files), or if you are interested in when the files were created, the order in which they were created, and who created or changed them. To see an earlier version of a file, click the “History” button in the Git panel<sup>11</sup> and select the commit that contains the version of the file that you would like to see. Then, in the lower section of the screen, click “View file @ [the commit’s SHA code].” This will reveal the file exactly as it was at that point in history.

Currently, RStudio’s Git panel offers limited functionality in rewinding history, and to our knowledge, the best tools for accessing old versions of a project are Git’s command line functions. In the Supplemental Material, we discuss in detail how to retrieve old versions of files, and even old versions of an entire project, using these functions.

## Collaborating

The true advantages of using Git become apparent when one considers projects with more than one contributor. For example, consider a project in which data are 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 (e.g., the popular Dropbox service). If two or more sites accidentally save a data file with the same name to the server (or service), and these changes are then automatically merged, whichever file was later will simply overwrite the earlier file. Disaster! Alternatively, consider a data analysis involving two or more people who work simultaneously on some complicated analysis script and

share their work using a central server or a system such as Dropbox. If user A and user 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) multisite collaboration on complex projects. For example, Microsoft Windows is developed collaboratively on a Git platform by about 4,000 engineers (Harry, 2017). We believe that Git can be especially helpful in scientific collaboration.

There are many ways in which a team could collaborate on a Git project; here we focus on a common one, called *centralized workflow*. In this workflow, a central (also called remote) repository is created on an online platform. Individual users work in their local Git repositories as we have already described, but can also send and receive changes from the central repository (see Fig. 9).

In brief, once a user wishes to collaborate on a project, the user creates a central repository, and connects his or her local repository to it. Other users can then clone their own local repositories from the central repository. The central repository exists on an online platform (e.g., GitHub) or, for example, on a private server, and the local repositories are on the collaborators’ own computers. Contributors, including the one who created the central repository, continue to work on their local repositories. After committing changes in their local repositories, they *push* their changes to the central repository, to make the changes available to other users. To get changes from the central repository, contributors *pull* changes from it to their local repositories.

Central Git repositories can be set up relatively easily on research teams' private servers, but because the details vary from team to team, here we illustrate the centralized workflow using GitHub.

## Collaborating With GitHub

GitHub is one of the 100 most popular Web sites worldwide. As of February 2018, it hosted more than 79 million software projects with a total of more than 28 million users. For this Tutorial, we chose to focus on GitHub for collaboration because it is already popular among scientists who use VCSs, it offers free private repositories to some users,<sup>12</sup> and repositories from GitHub can easily be connected to projects hosted on the Open Science Framework (OSF; <https://osf.io>). There are many alternatives to GitHub, such as Bitbucket (<https://bitbucket.org/>) and GitLab (<https://gitlab.com>), both of which provide free private repositories and function very similarly to GitHub. Thus, although we focus here on GitHub as an example, switching to another service would be relatively straightforward.

If you are a new GitHub user, you must first create a free user account at <https://github.com>. You must create the GitHub account with the same e-mail address that you used when configuring your local Git (or reconfigure your local Git to use the e-mail address that you used to register for GitHub). GitHub will use your e-mail address for authorization purposes and will use your username to identify you as the source of your commits. The Supplemental Material shows how to use the command line for configuring Git.

### ***Creating a new GitHub repository and linking it with a local Git repository***

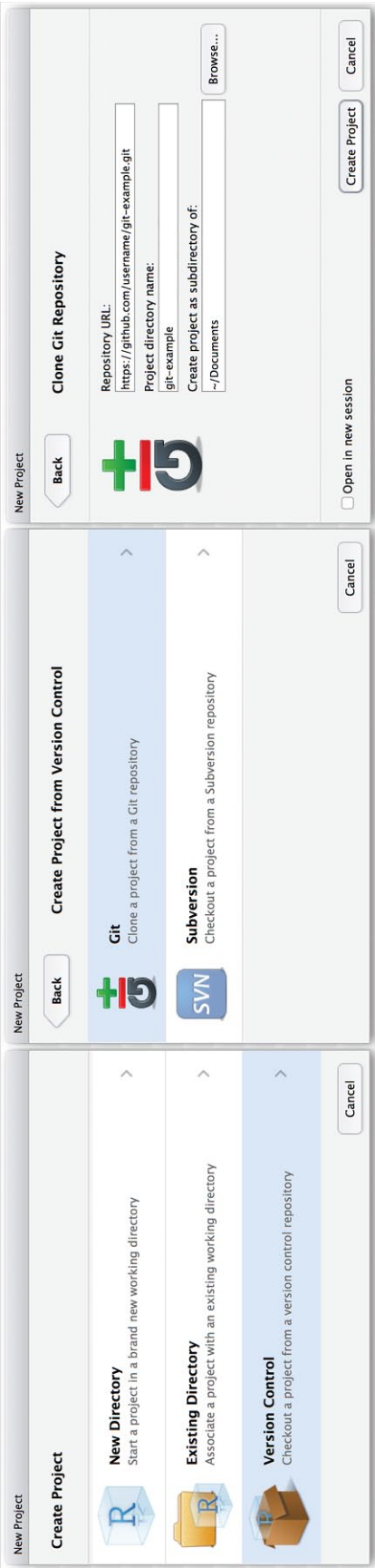
To start collaborating on a project, you first need to create a repository on GitHub. After you have created a GitHub account, log into your account and click the green "New repository" button on GitHub's main page. GitHub will first ask for a name for your GitHub repository. The name can be anything, but to continue with our example project, call the GitHub repository `git-example`. After indicating the repository's name, you can choose whether the repository should be public or private; we discuss this choice in more detail later, but for now, choose "Public." At this point, you can allow GitHub to automatically create README, `.gitignore`, and license files for your repository, but for this example, skip this step because these files have already been created in the local repository.

After you click on "Create repository," the next step is to link the new GitHub remote repository to a local

Git repository, by either creating a new local repository from the remote repository (this is called *cloning*) or connecting an existing local repository to the newly created remote repository. Currently, RStudio allows you to link a local Git repository to a remote repository only when you are creating a new project in RStudio; you cannot connect existing local Git repositories to GitHub with RStudio. (You can connect an existing local repository to a GitHub remote repository with two short lines of code in the computer's command line, and we show how in the Supplemental Material.) Therefore, we show here how to use RStudio to create a new project that connects to the GitHub remote repository you just created (Fig. 10). You need the remote repository's URL to set up the new project in RStudio. This URL is visible on GitHub on the page that appeared after you created the new repository. Copy the repository's URL from the address box on GitHub (the URL will end with `.git`, e.g., <https://github.com/username/reponame.git>). Then, in RStudio, click on "File" and then "New Project," and select "Version Control" (Fig. 10, left panel). This option allows you to create a new local repository by cloning the GitHub remote repository. Click on "Git" (Fig. 10, middle panel), then paste the GitHub URL in the "Repository URL" box, and choose an appropriate location on your computer for the project (Fig. 10, right panel). Note that the full address, including "https://," should be pasted in the box for the repository's URL.

You have now associated the empty remote (GitHub) repository with a local folder on your computer. Usually, you would connect the local and remote repositories as soon as a project is started. However, in this example, you created an empty GitHub repository but had a local repository for a project that was already started on your local computer, so you need to copy the contents of the old folders and paste them in the new local repository, so that you can continue with the same materials.

Next, make sure that the contents of the old `.gitignore` file have been copied to the new file. Otherwise you will be committing files that you would rather ignore into Git's history. You can then commit all the files to the new local Git repository using the steps detailed earlier in this Tutorial. After adding and committing some files locally, you will have two new buttons in RStudio's Git panel: The local repository can now send (push) changes to and receive (pull) changes from the remote repository. Click on "Push" and use a Web browser to navigate to the repository's GitHub page and refresh the page. You will then see all the committed files and folders on GitHub. You can browse and view the project's files, and even make changes to them, on the GitHub page.



**Fig. 10.** Creating a new local repository by checking out a project from a GitHub repository. Click on “Version Control” (left panel) and then “Git” (middle panel). The final step is to paste the URL for the GitHub repository in the appropriate box and indicate where the project should be saved on the local computer (right panel).

## Contributing to a central (GitHub) repository

Now that you have created a GitHub repository, other team members need only set up Git on their own computers and sign up for GitHub in order to clone the remote repository onto their local computers, just as you did. These users can find out the repository's URL by navigating their Web browsers to the repository's GitHub address (e.g., <https://github.com/mvuorre/reproguide-curate> for this Tutorial's repository; the repository's URL is the GitHub address with a `.git` extension) and clicking on the big green "Clone or download" button; the complete address will then appear in the text box, shown in Figure 11. Again, the full address, including "https://," must be pasted in the box for the repository's URL. After creating clones on their own computers, new contributors can work on their local copies of the project as detailed in earlier parts of this Tutorial. After committing their changes, they can update the status of the central repository by pushing their changes to it. To push changes, they simply press the "Push" button in RStudio's Git panel.

Note that you can also clone GitHub repositories if you are not aiming to contribute to a project. Hosting projects on GitHub is useful because other people can easily clone (download) your work to build on it.

## Obtaining other people's changes from the central repository

Just as you must manually push your own local changes to the remote repository, you must also obtain other contributors' changes by pulling them from the central repository. Pulling is indicated in the collaborative workflow in Figure 9 because it is important that you start working on the most up-to-date version of the

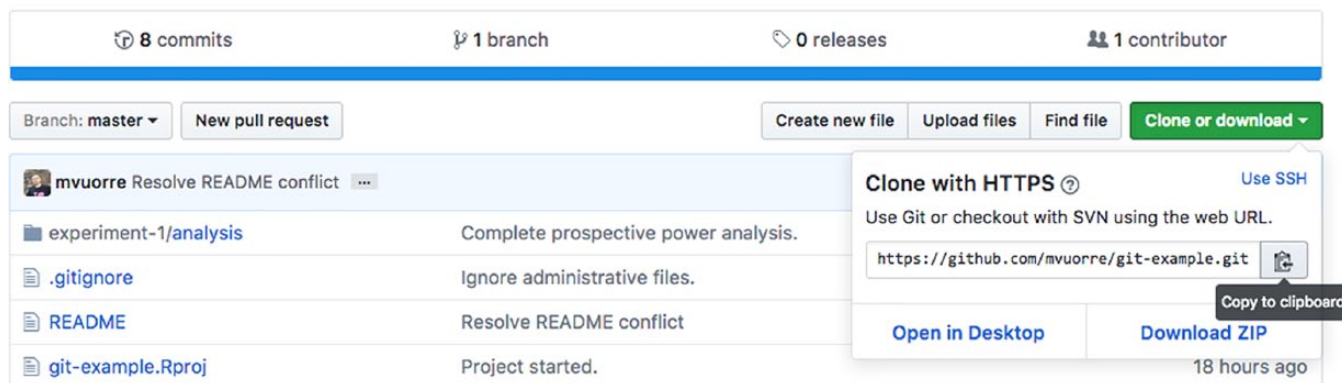
project (e.g., you do not want to redo work that has been completed successfully or make unnecessary conflicting changes). Before starting to work on potential changes, always pull the remote changes by pressing the "Pull" button in RStudio's Git panel.

## Resolving conflicts in GitHub

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 operations allows for complex workflows without any important code (data, ideas in a manuscript, analysis code) ever being overwritten. One possible concern is that two or more users may have worked on the same code and then attempted to push conflicting changes. There is no automatic way for a computer to tell what changes to prioritize, but if a conflict occurs, there is no need to worry; you simply need to know how to resolve it.

Many different types of conflicts may appear in collaborative work. For example, multiple users may create files with the same name but different content, or multiple users working on the same code may create changes that conflict with each other. Typically, this latter type of conflict occurs when the users have made edits to the same line of text in the same document. We use this 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 and work in local repositories). At some point, they might be working on the same file (e.g., they might be writing a manuscript together) and find that they have made changes that conflict with each other. More specifically, let us assume that both users are making changes to a file, and user B



**Fig. 11.** Screenshot illustrating how to find a remote repository's URL on a project's main GitHub page. When users navigate their Web browser to a repository's GitHub address and click on the big green "Clone or download" button, the complete URL appears in the text box.

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 incompatible changes to the central repository, a conflict occurs. 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, so that it reflects both collaborators' edits, and then push the new merged version to the central repository. Let us look at what this workflow entails in a little more detail.

For this scenario, assume that user B has made changes to the README file in the git-example project and pushed the changes to the central repository. When user A attempts to push conflicting changes to that repository, RStudio will display a Git error message indicating that the push would create a conflict (Fig. 12). User A sees that she needs to pull the most current state of the remote repository by clicking on "Pull."

After user A pulls the changes from the remote repository, Git will automatically merge the two conflicting versions of the README file into one file in the local repository. (Git will indicate in which file the conflict occurred; see Fig. 13.) Git will not remove anything, and therefore user A will need to decide which lines of the changed file to save and which ones to discard. User A can open the README file with RStudio's text editor, and it might look like this:

```
# Example Git Project

<<<<<<< HEAD
```

```
[user A's proposed version of the
text]
```

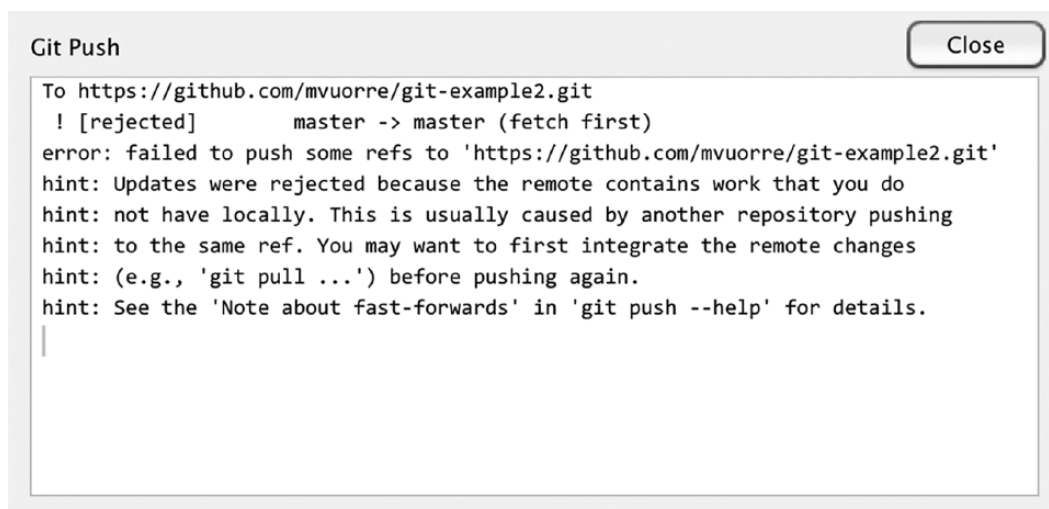
```
=====
```

```
[user B's proposed version of the
text]
```

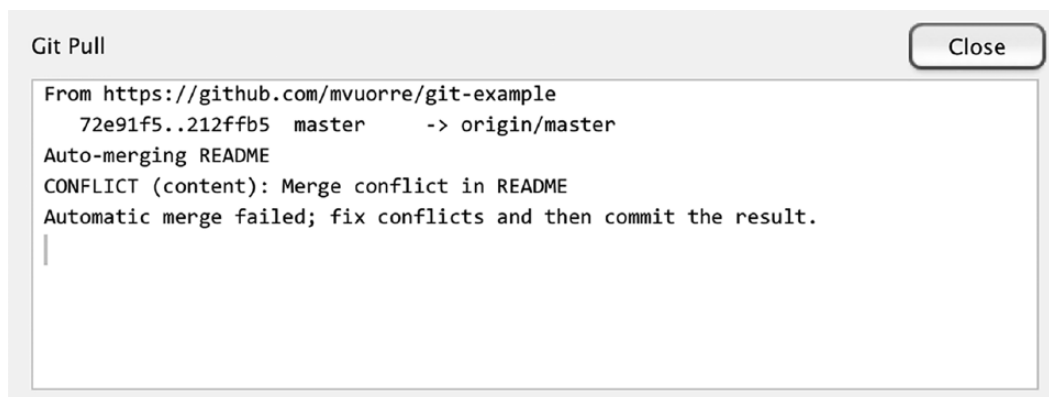
```
>>>>>> 212ffb5de589755ae4fda57fb5af
60194283dae8
```

This passage indicates that the first line of the file is identical in the two users' versions, but after the first line, "<<<<<<< HEAD" indicates that what follows are the to-be-integrated lines of text. First, user A's version is presented, and then, after "=====", user B's changes from the remote repository are presented, followed by the SHA key for those changes. User A can then edit this file however she chooses and then add the changes to Git's staging area and commit the changes. Once the commit is done, the conflict has been resolved, and user A can push the changes to the GitHub remote repository.

The options for dealing with conflicts within RStudio's Git panel are somewhat limited, and we present more detailed information on managing conflicts with Git's command-line tools in the Supplemental Material. There are additional kinds of conflicts that depend on how users collaborate with one another; however, a detailed explanation of all potential scenarios is outside the scope of this Tutorial.<sup>13</sup> Most important, 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.



**Fig. 12.** Screenshot showing how RStudio displays a Git warning if a user attempts to push conflicting changes to the remote repository.



**Fig. 13.** Screenshot from RStudio showing an error message for a conflict in the README file. Whenever a user pulls changes from the central repository and there is a conflict with the local repository, a conflict warning will indicate the file (or files) with a conflict.

### ***Private or public collaboration?***

By default, all GitHub repositories are public: Anyone with an Internet connection can use his or her Web browser to inspect the contents of your repository, or even clone it to his or her 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 writing this Tutorial—we see very few downsides to working “in the open.”

When collaborators wish to work privately, they have the option of placing the central repository for a Git project on the research team’s private server instead of GitHub, but it is also possible to make the repository private on GitHub (this can be done when the repository is first created or afterward). Private GitHub repositories, and their contents, are accessible only to invited team members and are therefore ideal when a team would like to work without revealing their work to the public. For example, you might want to work in a private repository initially and make it public only once you feel the material is mature enough for public consumption. Note, however, that making a repository public makes all of its contents public, including its Git history.

To make a GitHub repository private, navigate to its Web site with a Web browser and click on “Settings” and then “Make this repository private.” Once one user has set the GitHub repository to private mode, anyone wishing to clone or view the repository, push changes to it, or pull changes from it must provide his or her GitHub username and password. Only if they match an invited team member’s username and password can the user access the repository. At the time of this writing, GitHub users can have up to five private repositories for free (see note 12).

### ***Connecting a GitHub repository to other services***

OSF is designed for organizing and communicating research materials and is quickly becoming a popular service for sharing data sets and stimuli, among other content. Researchers can easily link GitHub repositories to their OSF projects by following on-screen instructions on the OSF Web site (<http://help.osf.io/m/addons/1/524148-connect-add-ons>).

Hosting a research project’s materials online on GitHub also makes the materials themselves citable. To facilitate citation, researchers can connect their GitHub repositories to Zenodo, an archiving Web site that will assign DOIs (digital object identifiers) to the repositories. Instructions for obtaining a DOI for a GitHub repository can be found online (<https://guides.github.com/activities/citable-code/>).

## **Discussion**

In this introductory Tutorial, we have explained how to use the Git VCS for curating and collaborating on research assets in behavioral sciences. The essential Git workflow includes adding and committing incremental changes to a version controlled repository, which can be worked on collaboratively by many researchers through the online GitHub platform.

Although we have advocated the use of Git and GitHub, we do not intend to suggest that it is the only, or always the best, method for curating and collaborating on research materials (see Box 3 for other common misconceptions related to Git). For example, hosting a text document on a service such as Google Docs or Dropbox allows collaborators to instantly see each other’s saved edits. Both of these alternatives also allow



**Box 3.** Responses to Common Misconceptions About Git

Misconception: It seems that it would be easy to lose my work with Git.

Response: Git does not change current files or folders unless asked to do so, and therefore using Git does not affect the likelihood that something will happen to them (e.g., because of hardware failure). Essentially, Git saves the history of a file's contents in a hidden .git folder inside the repository; if users tamper with this folder or run Git commands without knowing their consequences, it is possible to corrupt or lose parts of this history. Further, current uncommitted changes are discarded when a user retrieves (checks out) an older version of the repository. Therefore, users should ensure that important changes are committed before they retrieve an old version.

Misconception: I already back up my work, so I have no need for Git.

Response: Git is a tool for keeping track of changes to projects, not for backing up projects. However, connecting one's local Git repository to GitHub creates a cloud copy of it (provided one keeps the GitHub repository up-to-date).

Misconception: Saving a file is pretty much the same as a Git commit.

Response: Saving a file overwrites the file's immediately previous version on the computer's hard drive. Without Git, the file's previous versions can rarely be retrieved without a specialist's intervention. With Git, a file's previous committed versions can be retrieved from Git's history.

Misconception: I do not use Git because I want my work in progress to be private.

Response: Any Git repository created on a local computer is private. Even when a local Git repository is connected to a remote online repository, users must perform a separate operation (push) to send the local contents to the online repository. Further, users can set their online repositories to be either public (accessible by anyone) or private (accessible only with a password).

users to access files' history (Dropbox's history is limited to the past 30 days). If documents are worked on in Word with "Track Changes" enabled, collaborators can also see who has edited what and can revert changes in the text. In comparison, Git requires additional operations to send changes to and receive changes from collaborators, which may not be the best workflow for simple collaborative projects, such as editing manuscripts. Online services such as Overleaf (<https://www.overleaf.com/>) and Authorea (<https://www.authorea.com/>) also seem potentially useful for collaborative writing, and can be easier to use than Git.

However, for more complicated projects, VCSs, such as Git, have many benefits over these alternatives. For example, we believe it is helpful to think of many manuscripts in the context of a greater project that contains other materials, such as data, analysis scripts, and stimuli. In such cases, collaborators may wish to change not only the text of the manuscript, but also, for example, the associated statistical analyses. Git also helps collaborators to stay up-to-date with all of a project's components, thus possibly reducing room for error due to, for example, using outdated data or analysis files: Without Git, collaborators must create a different file for each version of the analysis, which can lead to an unwieldy project and confusion as to which version is the most up-to-date or appropriate. Further, because Dropbox and other similar services automatically synchronize files across collaborators, it is sometimes difficult to collaborate on the same file simultaneously using these services. Git allows multiple people to work on the same files simultaneously because the

saving and synchronizing steps are separated from each other.

Using Git with RStudio is an especially attractive option for psychologists because the RStudio IDE, with the R Markdown and knitr packages (Allaire et al., 2016; Xie et al., 2016), offers a complete environment for project management, data analysis, and manuscript preparation. Psychologists will also be interested in the papaja package for creating manuscripts formatted according to American Psychological Association style (Aust & Barth, 2018; this manuscript was prepared with the papaja package).

Although this Tutorial includes enough material for readers to get started, Git (and GitHub) is a vast ecosystem with great opportunities, some of which are discussed further by Perez-Riverol et al. (2016; see Box 4 for additional resources on using Git and GitHub). For example, the concept of "born-open data" (i.e., research data that are automatically posted online upon collection) can be implemented easily with the Git + GitHub workflow (Rouder, 2016). The challenges to reproducibility are many, and they have only recently received the targeted attention they deserve in the collaborative effort to improve the reliability of empirical sciences. Curating research assets and focusing on the practical aspects of the scientific workflow is important for ensuring the continuity of one's work, and for efforts toward a cumulative and reliable science.

**Action Editor**

Alex O. Holcombe served as action editor for this article.

**Box 4.** Additional Resources for Learning About Git

- The Basic Workflow of Git (an infographic explaining how Git's version control system works): <https://www.git-tower.com/blog/workflow-of-version-control>
- Git + GitHub (information on using Git and GitHub in an R programming context): <http://r-pkgs.had.co.nz/git.html>
- GitHub's Git cheat sheets (reference sheets on the most commonly used Git commands; available in multiple languages): <https://services.github.com/resources/cheatsheets/>
- GitHub Glossary (a glossary of Git and GitHub terminology): <https://help.github.com/articles/github-glossary/>
- *Pro Git* (Chacon & Straub, 2014; a complete manual of Git): <https://git-scm.com/book/en/v2>
- tryGit (an interactive Web site for learning the basics of Git): <https://try.github.io>

**Author Contributions**

M. Vuorre and J. P. Curley designed the format of this Tutorial. M. Vuorre drafted the manuscript, and J. P. Curley provided critical revisions and comments. Both authors approved the final version of the manuscript for submission.

**Acknowledgments**

We thank Tom Hardwicke, Travis Riddle, Judy Xu, and two anonymous reviewers for feedback on earlier drafts of this manuscript.

**Declaration of Conflicting Interests**

The author(s) declared that there were no conflicts of interest with respect to the authorship or the publication of this article.

**Funding**

This work was supported, in part, by Institute of Education Science Grant R305A150467. The authors are solely responsible for the content of this article.

**Supplemental Material**

Additional supporting information can be found at <http://journals.sagepub.com/doi/suppl/10.1177/2515245918754826>

**Open Practices**

All materials have been made publicly available via the Open Science Framework and can be accessed at <https://osf.io/txgn8/>. The complete Open Practices Disclosure for this article can be found at <http://journals.sagepub.com/doi/suppl/10.1177/2515245918754826>. This article has received the badge for Open Materials. More information about the Open Practices badges can be found at <http://www.psychologicalscience.org/publications/badges>.

**Notes**

1. The software we discuss in this article (Git, GitHub) is used by major software developers such as Microsoft, Google, and Facebook on code bases with hundreds of contributors.
2. The creator of Git, Linus Torvalds (who also is the principal developer of the Linux operating system), says he named

Git, which is British slang for “a rotten person,” after himself (McMillan, 2005).

3. Git *submodules* allow more advanced users to link projects to each other and to organize a complex project into subprojects (<https://git-scm.com/book/en/v2/Git-Tools-Submodules>).

4. There are no visible changes to a folder once it is tracked by Git. After 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.

5. Although we refer to the R programming language in the example that follows, Git can be used through the RStudio GUI even if you do not use R (or any other programming language).

6. For a video tutorial showing how to set up Git with RStudio, see <https://pagepiccinini.com/r-course/lesson-0-introduction-and-set-up/>.

7. The README file is so important that it has become standard practice to write this file name in capital letters. We follow this tradition here, but note that the capital letters are a tradition, not a requirement.

8. Plain text has many advantages over proprietary file formats, such as Microsoft Word's `.docx` format. Briefly, plain text is both human and computer readable, is both forward and backward compatible (there will always be, and has always been, software capable of reading it), and takes very little memory. The file extension of a plain-text file does not matter much, but we recommend using the widely recognized `.txt` extension or, in the case of markdown syntax, `.md`. For README files, no file extension is needed.

9. Files specified in `.gitignore` are only ignored by Git; they will still behave just like any other file in your local computer's hard drive.

10. 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. Ideally, each commit should 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 a specific change.

11. RStudio also has a History panel, which is related to R command history, and should not be confused with the “History” button in the Git Commit panel.

12. To obtain the free repositories, fill out the request form at [https://education.github.com/discount\\_requests/new](https://education.github.com/discount_requests/new). Students with `.edu` e-mail addresses can obtain unlimited free repositories at <https://education.github.com/pack>.

13. Although the instructions we provide will help in the most commonly encountered scenarios, readers can refer to the

following Web sites for more information: <https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/> and <https://www.atlassian.com/git/tutorials/comparing-workflows>. You can also resolve conflicts directly on GitHub (<https://help.github.com/articles/resolving-a-merge-conflict-on-github/>). Note that GitHub's customer service (<https://github.com/contact>) is very responsive to users' help requests.

## 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) [Computer software]. Retrieved from <https://cran.r-project.org/web/packages/rmarkdown/index.html>
- Aust, F., & Barth, M. (2018). *papaja: Prepare reproducible APA journal articles with R Markdown* (R package Version 0.1.0.9709) [Computer software]. Retrieved from <https://github.com/crsh/papaja>
- Chacon, S., & Straub, B. (2014). *Pro Git* (2nd ed.). Retrieved from <https://git-scm.com/book/en/v2>
- 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, 770–773. doi:10.1038/nn.4550
- GitHub. (2018). ["About" page]. Retrieved from <https://github.com/about>
- Harry, B. (2017, May 24). The largest Git repo on the planet [Web log post]. Retrieved 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*, 28, 348–354. doi:10.1093/beheco/arx003
- Markowitz, F. (2015). Five selfish reasons to work reproducibly. *Genome Biology*, 16, Article 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/](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., Percie du Sert, N., . . . Ioannidis, J. P. A. (2017). A manifesto for reproducible science. *Nature Human Behaviour*, 1(1), Article 0021. doi:10.1038/s41562-016-0021
- Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., Leprevost, F. da V., . . . Vizcaino, J. A. (2016). Ten simple rules for taking advantage of Git and GitHub. *PLOS Computational Biology*, 12(7), Article e1004947. doi:10.1371/journal.pcbi.1004947
- Project TIER. (n.d.). *Specifications*. Retrieved from <https://www.projecttier.org/tier-protocol/specifications/>
- R Core Team. (2017). *R: A language and environment for statistical computing* (Version 3.4.3) [Computer software]. Retrieved from <https://www.R-project.org/>
- Rouder, J. N. (2016). The what, why, and how of born-open data. *Behavior Research Methods*, 48, 1062–1069. doi:10.3758/s13428-015-0630-z
- RStudio Team. (2016). *RStudio: Integrated development environment for R* (Version 1.1) [Computer software]. Retrieved from <http://www.rstudio.com/>
- Vanpaemel, W., Vermorgen, M., Deriemaeker, L., & Storms, G. (2015). Are we wasting a good crisis? The availability of psychological research data after the storm. *Collabra: Psychology*, 1, Article 3. 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, 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) [Computer software]. Retrieved from <https://cran.r-project.org/web/packages/knitr/index.html>

1 Supplement to Curating Research Assets in Behavioral  
2 Sciences

3 Matti Vuorre<sup>1</sup> & James P. Curley<sup>1,2</sup>

4 <sup>1</sup> Department of Psychology, Columbia University, New York, USA

5 <sup>2</sup> Department of Psychology, University of Texas at Austin, Texas, USA

6 Contents

7	<b>Git setup</b>	<b>2</b>
8	The command line . . . . .	2
9	Setting Git's user information . . . . .	2
10	<b>Using Git from the Command Line</b>	<b>4</b>
11	Organizing Files and Folders . . . . .	4
12	Initializing a Git Repository . . . . .	5
13	Adding a File to Git . . . . .	5
14	Keeping Track of Changes with Git . . . . .	6
15	What Does Git Know? . . . . .	7
16	(Slightly More) Advanced Git . . . . .	8
17	“Rewinding History” with Command Line Git Functions . . . . .	9
18	Tagging important commits . . . . .	11
19	<b>Collaborating</b>	<b>11</b>
20	Connect a Local Repository to a GitHub Remote Repository . . . . .	11
21	Cloning a Remote Repository . . . . .	12
22	Obtaining other's changes from the central repo . . . . .	12
23	Resolving conflicts in collaborative work . . . . .	12
24	Deleting a Git Repository . . . . .	15

---

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](mailto:mv2521@columbia.edu)

This is a supplemental file to “Curating Research Assets in Behavioral Sciences”. In this file, we show how to set up and use Git from the computer’s command line, and highlight some more advanced Git functionality. The commands in this supplemental file run approximately parallel to the Git operations executed through the R Studio GUI in the main text.

## Git setup

Once you have installed Git, you can use it from the computer’s command line or through a GUI. Although you can use Git with a GUI, 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. For some Git functions, such as identifying its user, you will need to use a few command line functions, shown below. Before getting to the commands, we provide a brief introduction on how to use the computer’s command line.

### 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). The most common command line functions are listed in Table 1.

First, you need 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 (Windows)). This is important because whenever you execute commands in the command line, they are executed in a specific directory. 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 usually 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 with the Mac Terminal application.

### Setting Git’s user information

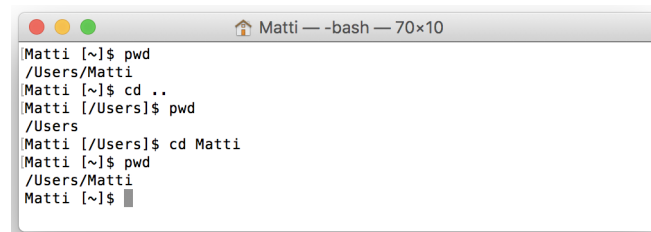
Some users may find using a text-based command line interface unfamiliar, but to get started with Git, there are two required configuration commands which you need to run

Table 1

*Basic command line functions.*

Command	Result
<code>pwd</code>	Show current directory.
<code>cd ..</code>	Move out of current directory (up one level).
<code>cd folder</code>	Move into folder (must be inside current directory).
<code>ls</code>	List files and folders in current directory.

*Note.* Execute the commands in your command line shell application (e.g. Terminal (Mac) or Git Bash (Windows)) by pressing Return (Mac) or Enter (Windows).



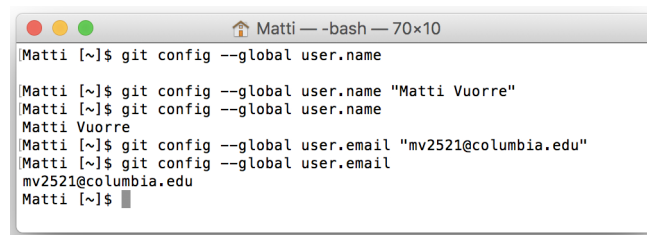
*Figure 1.* The Terminal command line shell. The functions are explained in more detail in the text. Each command (preceded with a \$ symbol) is followed by its output on the following line. To the left of the \$ symbol, this user's shell application displays the user's user name and the current directory in square brackets. Here, the outputs are directories separated with forward slashes, and top-most (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.

65 once, and the basic functionality requires using only a handful of commands<sup>1</sup>. The first step  
 66 in using Git is making it aware of who is using the computer. You need to set the user's  
 67 name and email address by entering a few basic commands in the command line. First, to  
 68 show the current user information, run the following command in the command line:

```
$ git config --global user.name
```

69 In these code listings, each command is preceded by a \$ symbol to indicate that they  
 70 are inputs to the command line. The command line typically displays this symbol (some  
 71 versions might use another symbol, such as >) on the current line where commands can  
 72 be entered. Do not type the \$ symbol as part of the command. To help remember the  
 73 commands, we recommend typing them out, instead of copy-pasting. Executing the above  
 74 command should not return anything, unless a previous user of the computer has set the  
 75 global Git user name. Each Git command starts with the word `git`, then a command (such as

<sup>1</sup>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).

76 `config`), and then arguments to the command, such as `--global` (for global configuration),  
 77 followed by variables, such as `user.name`. To ensure that Git knows who you are, execute  
 78 the following command (replacing `User Name` with your desired user name):

```
$ git config --global user.name "User Name"
```

79 This command maps `User Name` to Git's global `user.name` variable. If you now re-run  
 80 the first command (`git config --global user.name`), the command line will return the  
 81 user name you entered. The user name can be anything you'd like, but it is probably a  
 82 good idea to use your real name so that potential collaborators know who you are. The  
 83 second piece of information is your email address, which is entered by the following command  
 84 (where `email@address.com` is your email address):

```
$ git config --global user.email "email@address.com"
```

85 You can verify that the correct email address was saved with `git config --global`  
 86 `user.email`. These commands are shown as entered to the Terminal command line shell  
 87 application in Figure 1.

88 Once this information is entered, Git will know who you are, and is able to track who is  
 89 doing what and when within a project, which is especially helpful when you are collaborating  
 90 with other people, or when you are working on multiple computers. For detailed instructions  
 91 on how to use Git commands, you can type `git --help`. For help on how to use the `git`  
 92 `config` command, type `git config --help`. When printed in the command line, some  
 93 help pages run for several pages; you can press the space bar to move to the next page, or `q`  
 94 to quit looking at the help page.

## 95 Using Git from the Command Line

### 96 Organizing Files and Folders

97 To create a new folder for the Git repository, you first choose an appropriate folder on  
 98 your computer (such as `User/Documents/`) where you'd like to create the project. You can  
 99 either use the system's file navigator (Finder / File Explorer) to create this folder, or use  
 100 the command line: Navigate to the desired folder by using `cd Documents` to move into the  
 101 `Documents` folder (assuming it exists in the folder where you currently are). Use `cd ..` to  
 102 move out of a folder (to its containing folder), if needed.



```

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

### 103 Initializing a Git Repository

104 Once you are in the folder where you want to create the project, type `mkdir`  
 105 `git-example` to make the `git-example` directory, then `cd git-example` to move into  
 106 it. You can, of course, also use the point-and-click interface (Finder or File Explorer) to  
 107 create folders, instead of the `mkdir` command. Once you are in the project's home folder  
 108 (you can verify where you are by typing `pwd`), you can turn the folder into a Git repository  
 109 by initializing Git with the `git init` command. These commands are shown in Figure 3.

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

```
$ git init
```

111 The `git init` command initializes the folder as a Git repository, and the only change  
 112 so far has been the addition of a hidden `.git` folder inside `git-example` (and possibly a  
 113 `.gitattributes` file. Users can ignore these hidden files and folders, however they can be  
 114 shown with the `ls -la` command.) Now that the folder is initialized as a Git repository,  
 115 Git monitors any changes within it, and allows you to add and commit these changes.

### 116 Adding a File to Git

117 To see what files have changed since the last status change in the repository, you can  
 118 ask for Git's **status** (in subsequent code listings, command line input is prepended with \$,  
 119 and 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)

```

120 The relevant output returned from executing this command is the "Untracked files:"  
 121 part. There, Git tells the user that there is an untracked file (`README`) in the repository. To  
 122 start tracking changes in this file, we **add** it to Git's staging area by using the command  
 123 `git add` followed by the file name (i.e. `README`, in our example, the file doesn't have an  
 124 extension), or `.` which is a shortcut for adding all files with changes to the staging area.:

```
$ git add README
```

125 We have now added this file to the staging area, and if we are happy with changes to  
 126 the file's status, we can **commit** the file to Git's history. Here, we've created a README  
 127 file, and our commit command would look as follows:

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

128 The quoted text after the `-m` argument is the *commit message*. Entering this command  
 129 to the command line returns a brief description of the commit, such as how many files changed,  
 130 and how many characters inside those files were inserted and deleted. The distinction between  
 131 adding and committing in Git is important: The adding stage allows the user detailed  
 132 control of what to add to the staging area in preparation of a commit to Git's history. The  
 133 commit, then, commits the files from the staging area to history. These two operations can  
 134 be run simultaneously by including the `-a` option to `git commit` (e.g. `git commit -a -m`  
 135 `"Commit message"`), but it may be more difficult to control what gets committed with this  
 136 command, and we therefore recommend beginning users to do `git add` and `git commit` in  
 137 separate commands.

### 138 Keeping Track of Changes with Git

139 The `git-example` project (or rather, Git) now keeps track of all and any changes to  
 140 README. To illustrate, you can change the text in the README file with a text editor,  
 141 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")
```

142 Git can tell that the README file has been modified since the last commit. It is often  
 143 useful to know exactly *how* a file has changed, before committing it. To view differences to  
 144 a file not yet committed, use `git diff file`. It shows changes within `file`, line by line,  
 145 highlighting removed lines of text with red and added lines with green. Once you are happy  
 146 with the changes, you can repeat the add and commit steps from above to permanently  
 147 record the current state of the project to Git's history (below we use the `.` shortcut for  
 148 adding all files with changes<sup>2</sup>):

---

<sup>2</sup>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 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

```
$ git log
```

The output of this command shows 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 the changes in that commit. From top, it lists the commit's author, date, message, and then the commit's "**diff**". The diff is a detailed description of the changes introduced in that commit, explained in more detail below. The diff's first line tells that the following output is a git diff, and the origin of the changes was the README file (a/README), and the destination was the same file (b/README):

```
diff --git a/README b/README
```

163        These two file names could be different if the content from a file was moved to another  
 164        file with a different name, or if a file was moved. The next two lines expand the information  
 165        presented in the previous line:

```
--- a/README
+++ b/README
```

166        The first and second line in this output show the file which received deletions (lines were  
 167        removed, ---) and the file which received additions (lines were added, +++). Alternatively,  
 168        if a file was moved or renamed, these lines would indicate that. Following these lines, the  
 169        output shows where the changes were made:

```
@@ -0,0 +1,2 @@
```

170        The -0,0 indicates where in the initial version of the file the changes were made; the  
 171        first number indicates the first line of changes, the second indicates for how many following  
 172        lines the changes continued. This output is a little awkward, but because the file was initially  
 173        empty, the changes must have occurred on the 0th line. The second pair of numbers (+1,2)  
 174        indicates that the added text begins on line 1, and covers two lines of text. The last two  
 175        lines of the output indicated the changes in the text. The two added lines are prepended  
 176        with + symbols to indicate that these lines were added:

```
+# Example Git Project
+This example project illustrates the use of Git.
```

## 177 (Slightly More) Advanced Git

178        **Make Git ignore files.** To make Git ignore files, you simply add a plain text file  
 179        called `.gitignore` to the home folder of the repository. You can use any text editor to  
 180        create this file<sup>3</sup>. Notice that the file is *hidden* (by default, not visible in the OS's file viewer),  
 181        but can be seen in the command line with the command `ls -la`. Each row of this file should  
 182        specify a file or a folder (or a regular expression) that Git should ignore. In the current  
 183        example you could make Git ignore the `admin/` folder entirely (first line in the code listing  
 184        below), and any file with the `.pdf` extension inside `manuscript/` (second line). The example  
 185        `.gitignore` file would look like this:

```
admin/
manuscript/*.pdf
```

186        Re-running `git status` now only shows the `plan_n.R` file and the newly created  
 187        `.gitignore` file, which is also under version control, naturally. Because there are now

<sup>3</sup>You can also create this file in the command line by using the `touch` command: `touch .gitignore`. You can then open the file with a text editor to make changes to it.

two untracked files, which 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.<sup>4</sup>

```
$ 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. Below, we show some useful ways in which Git can be used to “rewind” the commit history.

## “Rewinding History” with Command Line Git Functions

**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<sup>5</sup>. Starting with an empty staging area, you could start modifying a file (e.g. `plan_n.R`) and later 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

```
$ 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) so that Git knows precisely which file you want to checkout from history. With these example operations, we have discussed the main Git operations: Make changes to files, **add** them to the staging area, **commit** to history; **checkout** from history to undo changes.

**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

<sup>4</sup>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.

<sup>5</sup>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-Brambling-Branches-in-a-Nutshell>).

earlier version, and then commit the older version.<sup>6</sup> 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. View the history with `git log` (you can use the `--oneline` argument for more concise output):

```
$ 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 its good state in commit `039d6ff`, you can 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).<sup>7</sup> `git log --oneline` would then show:

```
$ 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

<sup>6</sup>For more information on undoing changes, see <https://www.atlassian.com/git/tutorials/undoing-changes>.

<sup>7</sup>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.

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.

## Tagging important commits

Git also allows adding tags to commits. Tags can be used to signify important stages in the research cycle, or to mark otherwise particularly important commits. The simplest way to add a tag to the current commit is to use the `git tag` command followed by a name for the tag. The name should usually be a version number (e.g. v1.0), but text labels can be used as well. For example:

```
$ git tag v1.0
```

The above command will tag the current commit with the version number v1.0. All tags in the repository can be listed with `git tag`, and tagged commits can be accessed with their tag labels (e.g. `git show v1.0`). To create more informative tags (recommended), you can also use annotated tags by using the `-a` and `-m` options:

```
$ git tag -a v1.0 -m "Manuscript submitted"
```

Tags can become especially useful with larger projects. One common use for them is in software development to signify new versions or releases of the software.

## Collaborating

### Connect a Local Repository to a GitHub Remote Repository

After creating a GitHub repository, you need to link the existing local repository to the GitHub remote. To do this, use the following commands (the commands are also visible on GitHub, on a page that opens after you have created the repository):

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

Above, `username` and `reponame` are the user’s GitHub user name, and the GitHub repository name. The correct address is visible in the GitHub page that opens after creating the repository. Once you have added the GitHub remote to the local repository, you can verify that the correct address was given with the command `git remote -v`. Once the connection is set up, we can **push** local changes to the GitHub remote:



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

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

## 275 Cloning a Remote Repository

276 To clone a repository, first navigate to an appropriate location on the computer where  
 277 you would like to create the local repository. Once the appropriate location is found, cloning  
 278 will create a new folder for the repository inside this folder. Use `git clone` to clone a  
 279 repository from a URL:

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

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

## 284 Obtaining other's changes from the central repo

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

```
$ git pull
```

## 291 Resolving conflicts in collaborative work

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

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

295 At the same time, User A might have changed her local version of the `README` file to  
 296 look like this (User A also added to the second line):

*# Example Git Project*

Here are some changes.

This example project illustrates the use of Git.

297 If User A now commits the changes locally, and attempts to push the changes to the  
298 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.
```

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

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

304 The `--rebase` argument turns Git into rebasing mode, meaning that you are now  
305 applying your local commits on top of the “base” obtained from the remote repository.  
306 You can `git pull` without the `--rebase` argument, but that would create an unnecessary  
307 commit message and lead to a slightly different workflow<sup>8</sup>; we recommend using the `--rebase`  
308 argument. At this point, Git is in “rebasing” mode, allowing User A to resolve the conflict  
309 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")
```

310 Git again shows a helpful (if rather jargony) message; the relevant point is that the  
311 repository is in rebasing mode (technically, User A is re-basing her current local version

<sup>8</sup>For more information, see <https://www.atlassian.com/git/tutorials/comparing-workflows>.

312 (master) onto User B's latest contribution, identified with bada506). The next step is to  
 313 manually edit the conflicted file to reflect both User A's and User B's changes. Of course,  
 314 the user who is doing the rebasing (i.e. merging or resolving the conflict) may simply decide  
 315 to edit the file to reflect only her changes, completely rejecting User B's changes. When  
 316 viewed with a text editor, the conflicting README file, on User A's computer, now looks like  
 317 this:

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

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

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

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

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

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

335 How these potential conflicts appear depends on how users collaborate with one  
 336 another, and a detailed explanation of all potential scenarios is outside the scope of this  
 337 tutorial<sup>9</sup>. Most importantly, even in the event of conflicts, all committed changes are saved  
 338 in Git's history and can be retrieved, so experimenting with different approaches to resolving  
 339 conflicts is safe.

<sup>9</sup>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

## 340 **Deleting a Git Repository**

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

```
rm -rf .git
```

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

```
ls -la
```

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

---

more information: <https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/> and  
<https://www.atlassian.com/git/tutorials/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.