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

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

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

Abstract

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

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

Word count: 7700

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

Introduction

Lack of reproducibility is increasingly recognized as a problem across scientific disciplines, and calls for changing the scientific workflow to enhance reproducibility have been published in a wide range of research areas, including Biology (Markowetz, 2015), Ecology (Ihle, Winney, Krystalli, & Croucher, 2017), Neuroscience (Eglen et al., 2017) and Psychology (Munafò et al., 2017). Some studies suggest that one specific challenge to reproducibility is the ways in which researchers organize, curate, share, and collaborate on the research assets—e.g. data and code used to analyze the data—which support the product's conclusions (Vanpaemel, Vermorgen, Deriemaecker, & Storms, 2015; Wicherts, Borsboom, Kats, & Molenaar, 2006).

Fortunately for the empirical sciences, challenges related to organizing and curating materials across time, space, and collaborators have been solved to a high standard in computer science with Version Control Systems (VCS). In this tutorial article, 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 by using text commands from the computer's command line, and with a Graphical User Interface (GUI) implemented in the popular R Studio Integrated Development Environment (IDE; RStudio Team (2016)). We also show how to use Git with the popular online service GitHub for collaborative workflows. Learning how to use Git(Hub) will streamline workflows and help researchers to stay better organized, and thereby facilitate reproducibility.

Version Control Systems

Before proceeding with the practical tutorial, it is important to understand what Version Control Systems (VCS) are and what problems they were designed to address. VCS are computer programs designed for tracking changes to computer code, and collaborating on the code with others. They were initially developed by software engineers for writing code collaboratively[1], but are increasingly being adopted to enhance workflows outside computer science. To help understand why, it is helpful to think of "code" more broadly as any text written on a computer: Manuscripts, books, statistical analysis scripts, source code of computerized experiments, and even data files are "code" or have source code, which is just plain text written on a computer, and which can benefit from version control. Going forward, when we use the word "code" in this tutorial, we mean it in this broad sense (e.g. this manuscript's source "code" was written on a computer, and was version controlled.)

For example, computerized behavioral experiments are usually written with computer programming languages such as MATLAB or Python. The 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. Keeping track of the versions of the program and changes to the code, and allowing many authors to contribute to it (without breaking the experimental program) are problems that VCSs were specifically designed for. It might be less obvious that writing a manuscript is quite similar, at least as far as the computer is concerned: Multiple authors write multiple versions of a text document, and sometimes previous versions need to be inspected, and the text needs to be "merged"

---

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

across the many authors. Even less obvious is that datasets are often plain text: In most computerized behavioral experiments, the output data are numbers and text written into a text file. These text files can be version controlled (and researchers would usually not want to see that their raw data files have changed after they were created—VCS allows verifying that they haven't changed because their history is logged.)

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

To understand what VCS is, it is useful to contrast an example scenario with VCS to what we believe is a fairly standard workflow. Consider collaborating with one person on a manuscript that reports results from a data set using some statistical model. In the standard workflow, one person might format the raw data in one way to fit a statistical model, and then write a draft of the manuscript. This would create three files: `data.csv`[2], `analysis.R`, and `manuscript.doc`. If the coauthor then wished to explore another statistical model, which requires the data in a different format, and then edit the manuscript, she would create three more files: `data_new.csv`, `analysis_new.R`, and `manuscript_new.doc`. This cycle would then repeat as many times as required, each time creating more files, making it more and more difficult for the authors to remember which data was paired with which analysis, and which analyses (and data format) was reported in which version of the paper.

VCS greatly streamlines this workflow: With VCS, there would be only three files (data, analysis, and manuscript), but they would all be under version control, which would keep track of changes to the files and their different versions. Because VCS monitors changes to the files and allows saving a history for each of them, creating new files for every new idea or edit is unnecessary. This lack of duplicity, in turn, possibly reduces room for error in remembering which data file was linked with which analysis, and which manuscript version had the correct numerical results, and so forth. Further, as we explain below, VCS allows many coauthors to work on the files simultaneously, eschewing the need for emailing different versions of the same file to the coauthors, then waiting for them to make changes before continuing with your work. Box 1 shows practical examples of scenarios where VCS (specifically, Git, see below) can be used to improve the scientific workflow. To get a birds-eye view of the Git workflow, before beginning with the details, we suggest readers to inspect Figures 3 and 7 at this point.

---

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

> **Box 1. How Git facilitates the scientific workflow.**
>
> How to try different ways of visualizing and modeling data while keeping track of the different versions—and the differences between them?
> - Git saves each version of the analysis, allowing testing new features without losing previous versions or proliferating files in the project's directories. All the past versions can be directly compared with each other, and retrieved from the Git's history.
>
> How to work on the same code or manuscript files simultaneously with collaborators?
> - Git was designed for distributed collaborative work: Collaborators work on their local copies and "push" and "pull" material to and from each other or a central "repository". Git keeps precise track of who has done what, when, and (possibly) why. Git never loses information or overwrites work with another collaborator's work, but allows for many authors to work on the same ideas simultaneously.
>
> How to share work easily, safely, and in an organized manner?
> - Git enforces a common organization scheme among collaborators, making it easier to keep everyone "on the same page" with what goes where, and how to contribute to specific parts of the project. Git projects are shared as a whole, so complex projects with multiple files linking to each other are easy to share. Sharing projects with others is built into Git, and can be facilitated with online services such as GitHub: Once a project uses Git, it can be very easily copied to GitHub, from where others can easily download the entire project onto their local computers.

84

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[3], Mercurial[4], and Git[5]. In this tutorial, we focus on Git because it is already increasing in popularity within the scientific community, and is especially good for scientific collaboration because of online tools which allow seamless collaboration even for very large research teams. One of these online tools is GitHub, which we discuss in more detail below. Further, Git is free and open source, works on Windows, Mac, and Linux operating systems (among others).

Git, unlike the operating system's (OS) default file viewer (Mac's Finder, Windows' File Explorer), is not a point-and-click program for navigating files and folders on a computer. Instead, Git adds functionality to the existing file system, by making available a specific set of commands—either executed from the command line, or through a graphical point-and-click user interface (GUI)—which allow keeping track of files and their history, and distributing the files across multiple computers and users. Because Git is a standalone program not usually included in standard OS installs, users must first download and install the Git software on their computers.

---

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

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

[5]https://git-scm.com/. The creator of Git, Linus Torvalds (who also is the principal developer of the Linux operating system), named Git after himself as "the stupid content tracker" (McMillan (2005); https://git-scm.com/docs/git.html)

Installing Git

Even if you already have Git installed (some computers do) it is good practice to install the latest version[6], which can be downloaded as a standalone program from https://git-scm.com/download. For Mac users, the easiest way to install or update Git to the latest version is to download the installer from http://git-scm.com/download/mac, and install it like any other program. Similarly, Windows users can download the Git software installer from http://git-scm.com/download/win and install it like any other application. Linux (and other) users can download Git and find further install instructions on the Git website (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git).

Git setup

Once you have installed Git, you can use it from the computer's command line or through a GUI—there are various GUIs for using Git, and below we will explain how to use the R Studio Git GUI. Before using 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.

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.

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

---

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

Table 1

Basic command line functions.

Command	Result
pwd	Show current directory.
cd ..	Move out of current directory (up one level).
cd folder	Move into folder (must be inside current directory).
ls	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 (prepended 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.

once, and the basic functionality requires using only four commands[7]. However, users can also read through the first part of the tutorial without executing the commands and simply focus on the workflow; we show how to use Git with a GUI in the second part of the tutorial.

Setting Git's user information. The first step in using Git is making it aware of who is using the computer. You need to set the user's name and email address by entering a few basic commands in the command line. First, to show the current user information, run the following command in the command line:

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

This should not return anything, unless a previous user of the computer has set the global Git user name. Each Git command starts with the word `git`, then a command (such as `config`), and then arguments to the command, such as `--global` (for global configuration),

---

[7]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).

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).

153 followed by variables, such as `user.name`. To ensure that Git knows who you are, execute
154 the following command (replacing `User Name` with your desired user name):

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

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

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

161 You can verify that the correct email address was saved with `git config --global`
162 `user.email`. These commands are shown as entered to the Terminal command line shell
163 application in Figure 1.
164 Once this information is entered, Git will know who you are, and is able to track who is
165 doing what and when within a project, which is especially helpful when you are collaborating
166 with other people, or when you are working on multiple computers. For detailed instructions
167 on how to use Git commands, you can type `git --help`. For help on how to use the `git`
168 `config` command, type `git config --help`. When printed in the command line, some
169 help pages run for several pages; you can press the space bar to move to the next page, or `q`
170 to quit looking at the help page.

Using Git

171

172 The first operating principle of Git is that your work is organized into independent
173 projects, which Git calls *repositories*[8]. A repository is a folder on your computer which is
174 version controlled by Git[9]. Everything that happens inside a repository is tracked by Git,
175 but you have full control of what is committed to Git's history and when. Because you have

---

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

[9]There are no visible changes to a folder once it is tracked by Git. Once Git is initialized in a folder, the only change is that a hidden folder, called `.git` is added, but users do not need to interact with it directly.
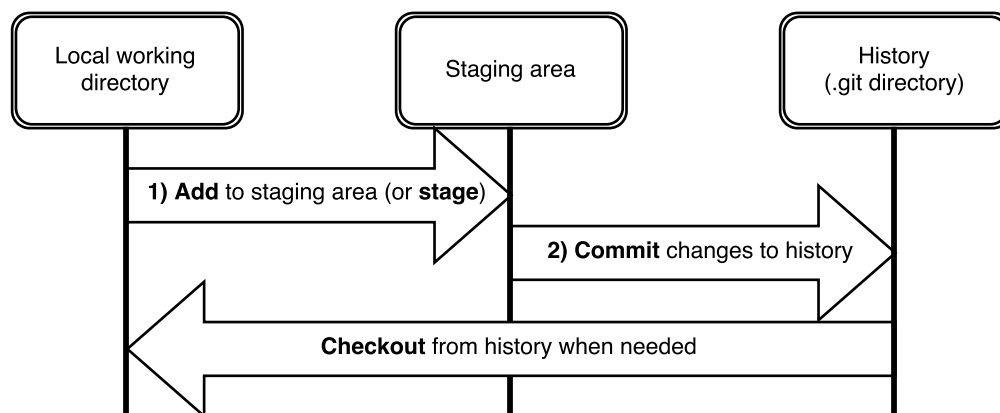
Figure 3. A diagram illustrating the typical Git workflow. Verbs indicated in bold text are Git operations and explained in detail in the main text. In brief, this workflow consists of making changes to files in the project (such as editing a manuscript.) Once the revised version of the file is saved, the user adds the changed file to the staging area (1). Many files can be added to the staging area, if desired. Once the changes in the staging area reflect a conceptual entity, such as edits to an image file and its accompanying caption in the manuscript, the user then commits the changes to Git's history (2). These commits are accompanied with short commit messages that describe the changes made in that commit. Finally, when needed, changes to files can be discarded by checking out an earlier version of the file from Git's history. Figure adapted from https://git-scm.com/book/en/v2/Getting-Started-Git-Basics.

full control of what and when is committed to history, there is a small set of operations you need to know[10].

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

To understand the Git workflow in practice, we now turn to a practical example using a hypothetical project. Git can be added to a project at any stage of the project's life cycle, but to most clearly show its use, we begin with an empty project with nothing in it.

---

[10]It helps to have a Git command cheat sheet (https://services.github.com/resources/cheatsheets/) printed and taped on your wall, but it contains many more commands than are needed for the basic use of Git in standard Psychology studies.
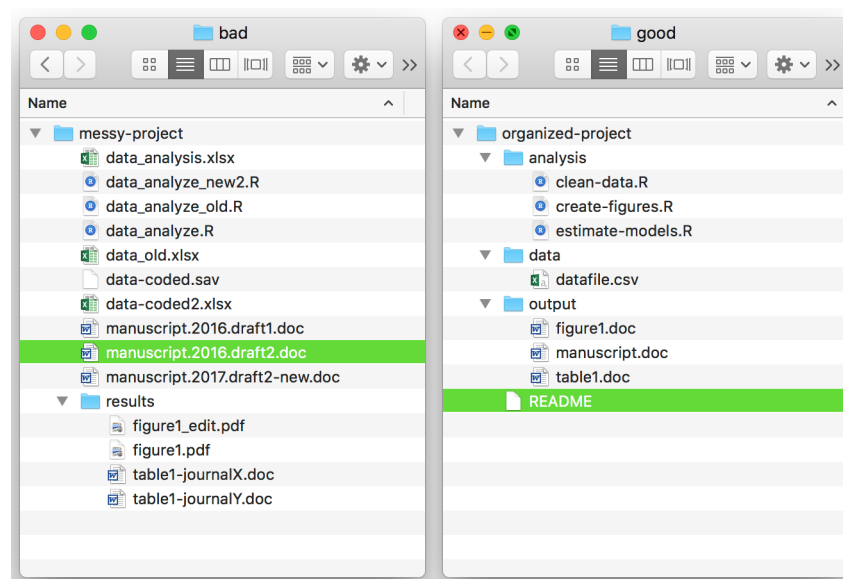
Figure 4. Organizing a project's files and folders helps potential collaborators (including yourself in the future) to quickly and reliably find the correct files. Some bad practices (left panel) include having multiple versions of each file, increasing the potential for e.g. accidentally using the wrong data for the analysis, or forgetting which version was used in the analysis. Better practices (right panel) organize the data to subfolders with meaningful names, and have one file per purpose. These files are then versioned using Git, eschewing the need for multiple files, thereby reducing potential for mistakes.

Organizing Files and Folders

Implementing reproducibility into the scientific workflow is less time-consuming and effortful if it is planned from the onset of the project, rather than added to the project after all the work has been completed. It is therefore important to organize the project keeping a few key goals in mind (here, we follow guidelines such as the Project TIER recommendations[11]): The files and folders should have easy to understand names (avoid idiosyncratic naming schemes), and the names should indicate the purposes of the files and folders. We illustrate some good and not-so-good practices in Figure 4.

The first step is to create a home folder for the project. This folder should have an immediately recognizable name, and should be placed somewhere on your computer where you can find it. We call the example project (and therefore its home folder) `git-example`. All the materials related to this project will be placed in sub-folders of the home directory, but for now `git-example` is just an empty folder waiting to be filled with materials. 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 sub-folders—is also known as the *root* directory. In what follows we use the terms home directory and root (directory) interchangeably.

To create a new folder for the Git repository, you first choose an appropriate folder on your computer (such as `User/Documents/`) where you'd like to create the project. You can either use the system's file navigator (Finder / File Explorer) to create this folder, or use

---

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

```
● ● ●                git-example — -bash — 80×9
Matti [~]$ cd Documents
Matti [~/Documents]$ mkdir git-example
Matti [~/Documents]$ cd git-example
Matti [~/Documents/git-example]$ pwd
/Users/Matti/Documents/git-example
Matti [~/Documents/git-example]$ git init
Initialized empty Git repository in /Users/Matti/Documents/git-example/.git/
Matti [~/Documents/git-example]$ ▊
```

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

²¹⁰ the command line: Navigate to the desired folder by using `cd Documents` to move into the
²¹¹ `Documents` folder (assuming it exists in the folder where you currently are). Use `cd ..` to
²¹² move out of a folder (to its containing folder), if needed.

Initializing a Git Repository

²¹⁴ Once you are in the folder where you want to create the project, type `mkdir`
²¹⁵ `git-example` to make the `git-example` directory, then `cd git-example` to move into
²¹⁶ it. You can, of course, also use the point-and-click interface (Finder or File Explorer) to
²¹⁷ create folders, instead of the `mkdir` command. Once you are in the project's home folder
²¹⁸ (you can verify where you are by typing `pwd`), you can turn the folder into a Git repository
²¹⁹ by initializing Git with the `git init` command. These commands are shown in 5
²²⁰ Instead of screenshots, for the rest of the tutorial we present the commands as follows:

```
$ git init
```

²²¹ In these code listings, each command is preceded by a `$` symbol to indicate that
²²² they are inputs to the command line. The command line typically displays this symbol
²²³ (some versions might use another symbol, such as `>`) on the current line where commands
²²⁴ can be entered. Do not type the `$` symbol as part of the command. To help remember
²²⁵ the commands, we recommend typing them out, instead of copy-pasting. The `git init`
²²⁶ command initializes the folder as a Git repository, and the only change so far has been the
²²⁷ addition of a hidden `.git` folder inside `git-example` (and possibly a `.gitattributes` file.
²²⁸ Users can ignore these hidden files and folders, however they can be shown with the `ls -la`
²²⁹ command.) Now that the folder is initialized as a Git repository, Git monitors any changes
²³⁰ within it, and allows you to add and commit these changes (Figure 3.)

Adding a File to Git

²³² Every project (repository) should contain a brief note explaining what the project
²³³ is about and who to contact. This note is usually called a readme, and therefore our first
²³⁴ contribution to this project will be a README file (this file is so important that it has
²³⁵ become standard practice to write it in capital letters). The README file should be a
²³⁶ plain text `.txt` file (i.e. not created with Microsoft Word) that can be read with a simple
²³⁷ text editor¹². You can create this file with any text editor (e.g. the TextEdit and Notepad
²³⁸ apps which allow for creating plain text files are installed by default on Mac and Windows

---

¹²Plain text has many advantages over proprietary file formats such as Microsoft Word's .docx files. Briefly,
plain text is both human and computer readable (.docx are unreadable without a copy of Microsoft Word),

OS respectively). We created the file with some text in it, and can now see the file in the `git-example` folder, either by using the system's graphical file viewer or with the `ls` command line command.

Because we added this file to a (folder that is a) Git repository, Git is also aware of it. To see what files have changed since the last status change in the repository (there clearly has been only this one), you can ask for Git's **status** (in subsequent code listings, command line input is prepended with `$`, 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)
```

The relevant output returned from executing this command is the "Untracked files:" part. There, Git tells the user that there is an untracked file (`README`) in the repository. To start tracking changes in this file, we **add** it to Git's staging area (Figure 3) by using the command `git add` followed by the file name (i.e. `README`, in our example, the file doesn't have an extension), or `.` which is a shortcut for adding all files with changes to the staging area.:

```
$ git add README
```

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

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

The quoted text after the `-m` argument is the *commit message*. Entering this command to the command line returns a brief description of the commit, such as how many files changed, and how many characters inside those files were inserted and deleted. The distinction between adding and committing in Git is important: The adding stage allows the user detailed control of what to add to the staging area in preparation of a commit to Git's history. The commit, then, commits the files from the staging area to history. These two operations can be run simultaneously by including the `-a` option to `git commit` (e.g. `git commit -a -m`

---

is both forward and backward compatible (there will always, and has always been, software capable of reading it), and plain text takes very little space. The file extension (the `.docx` part of a Word file's name, for example) of plain text doesn't matter much, but we recommend using either `.txt` because it is widely recognized, `.md` for markdown syntax, or in the case of a readme file, simply not using any file extension.

²⁶⁵ "`Commit message`"), but it may be more difficult to control what gets committed with this
²⁶⁶ command, and we therefore recommend beginning users to do `git add` and `git commit` in
²⁶⁷ separate commands.

Keeping Track of Changes with Git

²⁶⁹ The `git-example` project (or rather, Git) now keeps track of all and any changes to
²⁷⁰ README. To illustrate, you can change the text in the README file with a text editor,
²⁷¹ 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")
```

²⁷² Git can tell that the README file has been modified since the last commit. It is often
²⁷³ useful to know exactly *how* a file has changed, before committing it. To view differences to
²⁷⁴ a file not yet committed, use `git diff file`. It shows changes within `file`, line by line,
²⁷⁵ highlighting removed lines of text with red and added lines with green. Once you are happy
²⁷⁶ with the changes, you can repeat the add and commit steps from above to permanently
²⁷⁷ record the current state of the project to Git's history (below we use the . shortcut for
²⁷⁸ adding all files with changes[13]):

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

What Does Git Know?

²⁸⁰ The real importance of these somewhat abstract steps becomes apparent when we
²⁸¹ consider the Git **log**. To reveal the commit log of your repository, call

```
$ git log
```

²⁸² The output of this command shows that each commit is identified with a unique
²⁸³ hash code (long alphanumeric string), which we can use to call for further information (see
²⁸⁴ below); an author; a date and time; and a short commit message. Executing `git log` on
²⁸⁵ our example project at this stage returns this:

---

[13]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.

```
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
```

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

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

The first and second line in this output show the file which received deletions (lines were removed, `---`) and the file which received additions (lines were added, `+++`). Alternatively, if a file was moved or renamed, these lines would indicate that. Following these lines, the output shows where the changes were made:
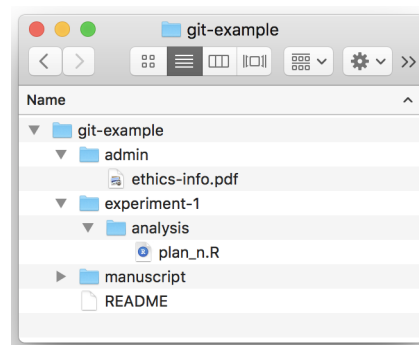
Figure 6. A project with more components.

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

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

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

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 doesn't allow full appreciation of the benefits of using Git for version control. To better illustrate Git's functioning, we now fast-forward in the hypothetical example project to a stage where more files and materials have been created.

(Slightly More) Advanced Git

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

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

326   **Make Git ignore files.**   To make Git ignore files, you simply add a plain text file
327 called `.gitignore` to the home folder of the repository. You can use any text editor to
328 create this file[14]. Notice that the file is *hidden* (by default, not visible in the OSs file viewer),
329 but can be seen in the command line with the command `ls -la`. Each row of this file should
330 specify a file or a folder (or a regular expression) that Git should ignore. In the current
331 example you could make Git ignore the `admin/` folder entirely (first line in the code listing
332 below), and any file with the .pdf extension inside `manuscript/` (second line). The example
333 `.gitignore` file would look like this:

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

334   Re-running `git status` now only shows the `plan_n.R` file and the newly created
335 `.gitignore` file, which is also under version control, naturally. Because there are now
336 two untracked files, which are not specified to be ignored in `.gitignore` (`.gitignore` and
337 `plan_n.R`), and you usually should aim to maintain a clean commit history for the project,
338 you can create two separate commits: One for the `.gitignore` file, and one for the power
339 analysis file.[15]

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

340   After this last commit you can, at any time in the future, come back to this commit
341 with `git log` or `git show` and see what was inside the newly created power analysis file
342 when it was first created. For instance, if new information suggests that you should change
343 the assumed effect size in the power analysis, you can simply edit and save the file, then
344 add and commit the changes to Git with a helpful message that logs this important event in
345 Git's history.
346   This possibility of "rewinding history" is especially useful for files that might go under
347 multiple revisions (manuscripts, analysis files), or if you are interested in when and in
348 what order the files were created—and who created or changed them. One might consider
349 committing a power analysis file to Git as a small personal pre-registration of a part of the
350 research plan. Below, we show some useful ways in which Git can be used to "rewind" the
351 commit history.
352   **Try a new feature.**   We often find that making some changes to a project didn't
353 have the desired effect: The manuscript ended weaker or the analysis didn't work anymore.
354 Git allows great flexibility in trying new features, then undoing the changes[16]. Starting with

---

[14]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.

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

[16]A particularly powerful approach for trying new features is **branching**: The project can be duplicated
to a new branch and modified, then merged back to the main branch after work on the new feature is

355 an empty staging area, you could start modifying a file (e.g. `plan_n.R`) and later realize
356 that the changes were not good. At this point it is common to press "Undo" in the text
357 editor, but if the file has been saved multiple times or multiple files have been changed, it is
358 difficult to get to the starting point by simply using the "Undo" button. Instead, with Git
359 you can **checkout** the file's previous version from history. To undo all changes to `plan_n.R`
360 (since the last commit), run

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

361   Notice that you have to write the full path of the file (relative to the project's root,
362 see Figure 6) so that Git knows precisely which file you want to checkout from history. With
363 these example operations, we have discussed the main Git operations as outlined in Figure
364 3: Make changes to files, **add** them to the staging area, **commit** to history; **checkout** from
365 history to undo changes. Although these operations are enough to maintain a Git repository,
366 we have only scratched the surface of Git's functionality.

367   **Undo committed changes.**   Another common scenario is one where a user makes
368 changes to a file, adds the changes to the staging area, commits them to Git's history, and
369 only then realizes that the changes weren't good. If you have committed changes to a file,
370 and would like to revert back to an older version of the file, you could **checkout** the file's
371 earlier version, and then commit the older version.[17] For example, suppose you have made
372 bad changes to a file called `file.txt`, and committed the changes to history, and would
373 then like to undo the bad changes by reverting to an older version of the file. View the
374 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
```

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

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

382   Now asking for `git status` reveals that `file.txt` has been modified in the working
383 directory (its current state is as it was in commit `039d6ff`). You can now add and

---

complete—or the new branch can be discarded if the work ended unsatisfactory. Branches are outside
the scope of this tutorial, for more information see the Git website (https://git-scm.com/book/en/v2/
Git-Branching-Branches-in-a-Nutshell).

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

commit these changes with `git add file.txt`, then `git commit -m "Undid bad changes to file.txt"` (type a commit message suitable for your situation).[18] `git log --oneline` would then show:

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

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

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

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

Collaborating

The true advantages of using Git become apparent when we consider projects with more than one contributor. For example, consider a project where data is collected at multiple sites, and the data files are then saved onto a central server, or shared through a service that automatically merges files from multiple sources (such as the popular Dropbox service). If two or more sites accidentally save a data file with the same name to the server (or Dropbox), and these changes are then automatically merged, the later file will simply overwrite the earlier file. Disaster! Alternatively, consider a data analysis where two or more people work simultaneously on some complicated analysis script, and share their work using a similar system as in the above example. If user A and B are making changes to the same file and user B saves the file, user A's version of the file will be overwritten. Disaster!

Git and other VCSs, on the other hand, were specifically designed to allow (and facilitate) multi-site collaboration on arbitrarily complex projects: Microsoft Windows is

---

[18]If, for some reason, you preferred the latest version after all, you can undo the revert process by `git checkout HEAD file.txt`, instead of adding and committing the older version. This function checks out the current state ("`HEAD`") which contained the bad changes.
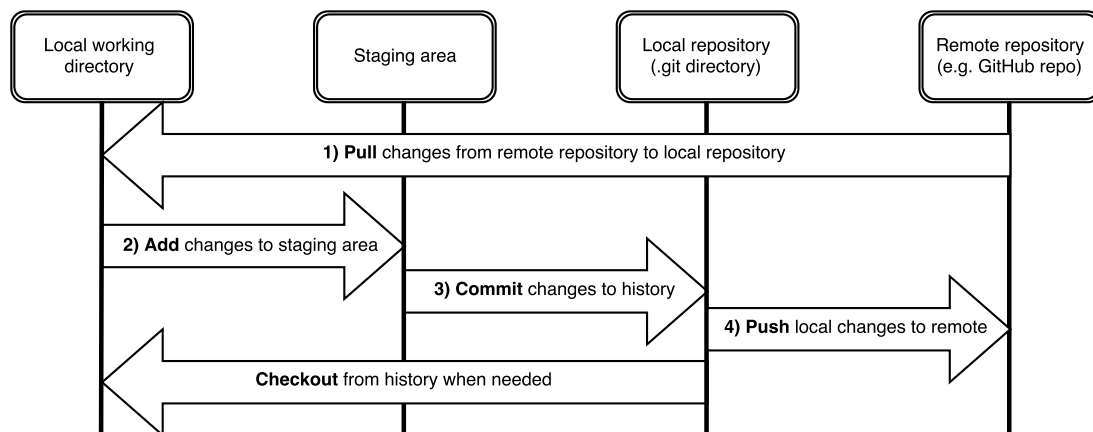
Figure 7. A diagram illustrating the typical collaborative Git workflow with a remote repository (e.g. GitHub). Verbs indicated in bold text are Git operations and explained in detail in the main text.

developed on a Git platform collaboratively by about 4,000 engineers (Harry, 2017). We believe that Git can be especially helpful in scientific collaboration, a topic to which we turn next.

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

Overview of the Centralized Git Workflow

In this workflow, a central project hosted on a research team's server, or an online service like GitHub, is used to send and receive information from and to local projects. Because the word "project" can now refer to virtual copies of projects, they are instead commonly referred to as **repositories**. First, one user creates this central repository, then other users **clone** local copies of it (see Box 2). Contributors, including the one who created the central repository, then work on their local projects as detailed above, and after committing, **push** their changes to the central repository. To get changes which other users have pushed to the central repository, contributors **pull** changes from it.

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

GitHub

GitHub is one of the 100 most popular website worldwide, and hosts over 60 million software projects with a total of over 20 million users.[19] We chose GitHub for collaboration because it is already the de facto standard among scientists who use VCS, it offers free private repositories (see below), and repositories from GitHub can easily be connected to projects hosted on the Open Science Framework (https://osf.io).

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

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

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

Create a GitHub repository from an existing project. Alternatively, we may already have a project with several files in it that is not yet a Git repository and that we wish to share with others. To do this, first log in to your GitHub account and create a new

---

[19]https://github.com/about.

⁴⁷⁴ repository. This can be done by clicking on the large green button called `New` under the
⁴⁷⁵ `Repositories` tab. On the next page enter the repository name in the text box and choose
⁴⁷⁶ whether to make the repository public or private. The name of the repository should match
⁴⁷⁷ the name of the top-level folder of the project you wish to share. We recommend not checking
⁴⁷⁸ the box for adding a README or adding a `.gitignore` file or license at this point. These
⁴⁷⁹ can be added after completing the setup. Next open the command line (e.g. the Terminal
⁴⁸⁰ for Mac users, Git Bash for Windows users) and change the current working directory to
⁴⁸¹ the home directory of the project you wish to share.

⁴⁸² First initialize the local directory as a Git repository:

```
$ git init
```

⁴⁸³ Now add all files to the staging area (if there are files or folders that you would not
⁴⁸⁴ like to include, see the above section on how to make Git ignore files with the `.gitignore`
⁴⁸⁵ file):

```
$ git add .
```

⁴⁸⁶ Commit these files including a comment such as "First commit":

```
$ git commit -m "First commit"
```

⁴⁸⁷ To push the local files to GitHub we follow the same steps as in the previous section
⁴⁸⁸ where `username` and `reponame` are the user's GitHub username, and the GitHub repository
⁴⁸⁹ name.

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

⁴⁹⁰ You can then verify the new repository:

```
$ git remote -v
origin  https://github.com/username/reponame.git (fetch)
origin  https://github.com/username/reponame.git (push)
```

⁴⁹¹ The two lines of output indicate the remote repository from where changes are to
⁴⁹² be received, and from they are to be sent, respectively. These should usually be the same
⁴⁹³ address. You can then push all files to GitHub with the first commit:

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

⁴⁹⁴ Like when sharing a git repository created from scratch, after these steps all users
⁴⁹⁵ can contribute modifications to the central GitHub repository by pushing changes with `git`
⁴⁹⁶ `push` after adding and committing locally.

Contributing to a central (GitHub) repository.   Once another team member
has set up Git on their own computer, and signed up for GitHub, they need to **clone** the
remote GitHub repository onto their local computer. To clone a repository, the new user
must first navigate to an appropriate location on the computer where they would like to
save the project on their computer, for example their `User/Documents` folder. They do not
need to create the project's folder locally before cloning the remote repository. Once the
appropriate location is found, cloning will create a new folder for the repository inside this
folder. Use `git clone` to clone a repository from a URL:

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

To find out the correct URL to enter to `git clone`, you can navigate your web browser
to the repository's GitHub address (e.g. https://github.com/mvuorre/reproguide-curate for
this tutorial's repository) and click the big green "Clone or download" button; the complete
address is in the text box. New contributors can then work on their local copies as detailed
in earlier parts of this tutorial; making changes, adding, committing, and pushing. After
committing their changes, they can update the status of the central Git repo by pushing
their changes to it:

```
$ git push
```

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

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

```
$ git pull
```

The way in which users and their local repositories interact with the central repository
by pushing and pulling is the cornerstone of collaboration on GitHub, and thoughtful use of
these commands allows for complex workflows without any important code (data, ideas in
manuscript, analysis code) ever being overwritten. For instance, prior to restarting work on
a project it is often worth checking if any files have changed in the central repository by
checking the status of your local repository with `git status` as described above. If someone
else has made changes to the central repo, pull them to your local repo before starting your
work. It is also worthwhile in committing and pushing changes made after each period of
work. However, there is no automatic way for a computer to tell what changes to prioritize:
If two or more users have worked on the same code and then attempt to push their changes,
it is possible that they have made changes which conflict with each other. If this happens,
there is no need to worry; you simply need to know how to resolve the conflict.

Resolving conflicts in collaborative work

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

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

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

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

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

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

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

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

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

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

565 The `--rebase` argument turns Git into rebasing mode, meaning that you are now
566 applying your local commits on top of the "base" obtained from the remote repository. You
567 can `git pull` without the `--rebase` argument, but that would create an unnecessary commit
568 message and lead to a slightly different workflow[20]; we recommend using the `--rebase`
569 argument. At this point, Git is in "rebasing" mode, allowing User A to resolve the conflict
570 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")
```

571 Git again shows a helpful (if rather jargony) message; the relevant point is that the
572 repository is in rebasing mode (technically, User A is re-basing her current local version
573 (`master`) onto User B's latest contribution, identified with `bada506`). The next step is to
574 manually edit the conflicted file to reflect both User A's and User B's changes. Of course,
575 the user who is doing the rebasing (i.e. merging or resolving the conflict) may simply decide
576 to edit the file to reflect only her changes, completely rejecting User B's changes. When
577 viewed with a text editor, the conflicting `README` file, on User A's computer, now looks like
578 this:

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

579 The first line of the file was identical across the two Users' versions of the file, and
580 therefore remains the same. However, after the first line, there is first a line (`<<<<<<< HEAD`)
581 indicating that what follows are the to-be-integrated lines of text. Anything after this line

---

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

up to the ====== line is the to-be-integrated text from the central repository. We can see that this is simply the line of text that User B created ("Hello world!"). After the separating line (======) are User A's local changes, followed by those changes' commit message ("Some meaningful changes") prepended with a >>>>>>>. User A can then edit this file however she chooses, using the tags to help her see which are her lines of code (text), and which are User B's. For example, 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.
```

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

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

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

How these potential conflicts appear depends on how users collaborate with one another, and a detailed explanation of all potential scenarios is outside the scope of this tutorial[21]. Most importantly, even in the event of conflicts, all committed changes are saved in Git's history and can be retrieved, so experimenting with different approaches to resolving conflicts is safe.

Private or public collaboration?

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

There are two alternatives to working in a public GitHub repository: One, which we won't cover here, is to not use GitHub but instead place the central repository on the research team's private server. The second option is to make the repository private on

---

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

GitHub (this can be done when the repository is first created or afterwards by clicking Settings on the repository's website). Private repositories, and their contents, are only accessible to invited team members, and are therefore ideal for small teams who would like to work without revealing their master plans to the public just yet. For example, you might initially choose to work in a private repository, and only make it public once you feel the material is mature enough for public consumption.

To make a GitHub repository private, navigate to the repository's website with a web browser, and click "Settings", then "Make this repository private". Once one user has set the central GitHub repository to private mode, anyone wishing to clone, push, pull, or view the repo must provide their GitHub username and password. Only if they match an invited team members username and password can the user access the repository. At the time of this writing, GitHub offers five private repositories for free[22].

Using Git with a Graphical Interface

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

R Studio

In this section, we introduce how to use the popular R Studio[23] IDE (RStudio Team, 2016) for the R programming language (R Core Team, 2017) for interacting with Git. An IDE is an interface that bundles together many tools for software development—some users may be familiar with R Studio for conducting statistical analyses. Version control is an essential feature of software development, and R Studio provides a good GUI within its IDE for controlling Git[24].

For this example, we will begin a completely new project using only R Studio, and assume that readers are familiar with the concepts from the above tutorial. Once downloaded and installed, it can be used for accessing the R programming language, and for many operations involved in curating research assets, including a GUI for Git.

Managing projects (and Git repositories) with R Studio is centered on the idea of R Projects. To start a new project with R Studio, open the R Studio application, and click File -> New Project. This brings up a dialog (left panel in Figure 8) asking whether to create a project in a new directory, existing directory, or checkout an existing project from a version control repository. Here, we create the project a new directory. We then choose "New Project" in the following screen (middle panel in Figure 8). Then we'll give a name to the

---

[22]To obtain the free repositories, fill out the request form at https://education.github.com/discount_requests/new. Students with .edu email addresses, can obtain unlimited free repositories at https://education.github.com/pack.

[23]R Studio is a free and open source IDE, works on Windows, Mac, and Linux operating systems, and can be downloaded from the project's website at https://www.rstudio.com/products/rstudio/download/.

[24]Although in this example we refer to the R programming language, Git can be used through the R Studio GUI even if you do not use R (or any other programming language.)
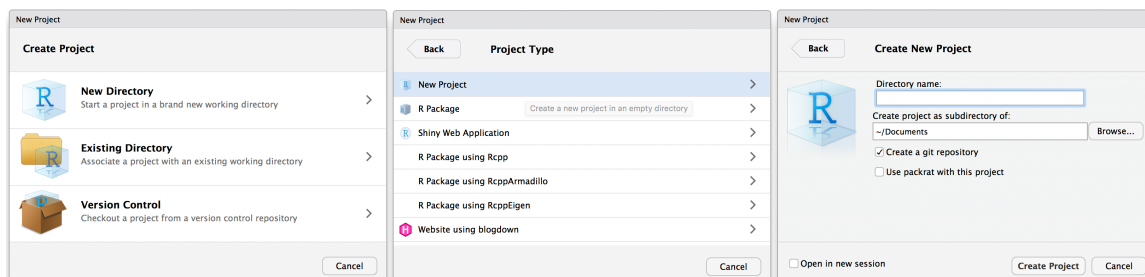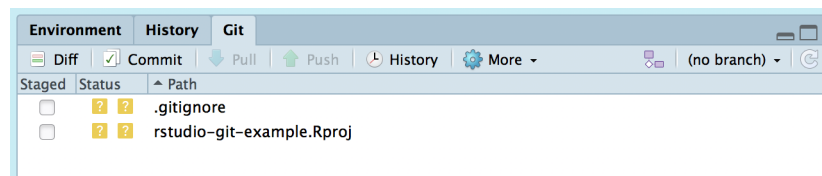
Figure 8. Creating a project in R Studio.

Figure 9. R Studio's Git tab for a newly created project. The Git tab in R Studio has buttons for Git commands covered in this tutorial, such as viewing Git's history, diffs, and commits.

project's home folder and choose where to save it on the computer. Importantly, we'll also check the "Create a git repository" box (right panel in Figure 8), which will automatically set up a new repository for the project. Clicking "Create project" creates the folder in the specified location, and two files inside the project's main folder.

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

Using Git with R Studio

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

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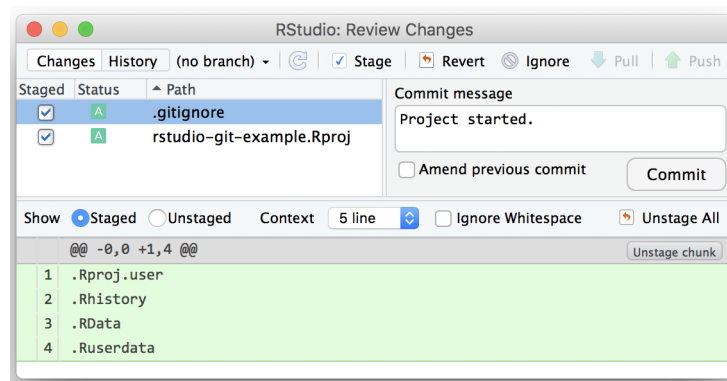
Figure 10. Creating a Git commit with R Studio.

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

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

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

Git Summary

Box 2 describes the main Git commands and their purported use in roughly the order in which they would be used in a typical workflow. Help on how to use each command is available by appending `--help` to each command (e.g. `git commit --help`). In Box 3 we give links to online materials that we have found particularly helpful in learning Git and teaching it to others.

> **Box 2. Main Git commands**
>
> `git init`: Initialize a local repository.
> - Create a local Git repository inside a folder on your computer. In effect, this command turns the local folder into a Git repository.
>
> `git clone URL`: Obtain a copy of a remote (GitHub) repository to local computer.
> - Create a local Git repository from a pre-existing remote Git repository, such as one hosted on GitHub.
>
> `git add file`: Add changes in `file` to Git's staging area.
> - Once a file has been edited or created in the local project, it needs to be added to Git's staging area. Multiple files can be added simultaneously using the `.` shortcut (`git add .`). Once the files have been added, they can be saved to Git's history.
>
> `git commit`: Save changes from staging area to Git's history.
> - A commit creates a snapshot of the project's current state by adding the most recent changes (which have been added to the staging area) to Git's history. Commits should be accompanied with short, descriptive messages to describe the changes introduced in that commit (`git commit -m "Commit message"`)
>
> `git log`: Show Git's history.
> - Git's log is a list of all the commits made in the current repository, and as such is a description of the project's history. The history can be very long and span multiple pages; to move forward one page, press the space bar, or "q" to quit viewing the log. To view the log in a more concise form, add the `-oneline` argument (`git log -oneline`).
>
> `git push`: Send locally committed changes to remote repository.
> - In the centralized workflow for collaboration as described in this tutorial, collaborators work on their own local repositories, and periodically send the changes that they have made to the central (remote) repository.
>
> `git pull`: Receive changes from the remote repository.
> - To keep the local repository up-to-date with other collaborators' changes, users must periodically pull changes from the remote (central) repository. It is good practice to pull the most up-to-date version of the project every time before making changes, to avoid unnecessary conflicts.

Deleting a Git Repository

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

```
rm -rf .git
```

We recommend caution with this operation, as it will permanently delete the `.git` folder, which may contain important information. To verify that the folder was deleted, you

can list all the files and folders in the current working directory, including hidden ones, with the following command:

```
ls -la
```

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

Discussion

We have presented an introductory tutorial on using the Git version Control System for curating and collaborating on research assets in behavioral sciences. The essential Git workflow includes adding and committing incremental changes to a version controlled repository, which can then be collaboratively worked on by many researchers through the online GitHub platform. Although this tutorial includes enough material to get started, Git (and GitHub) is a vast ecosystem with great opportunities, some of which are further discussed by Perez-Riverol and colleagues (2016). For example, the concept of "Born open data", where research data is automatically posted online upon collection, is made very easy with the Git + GitHub workflow (Rouder, 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.

> **Box 3. Further resources for learning Git**
>
> - Basic VCS workflow, an infographic explaining how VCS works (https://www.git-tower.com/blog/workflow-of-version-control).
> - GitHub's Git cheat sheet is available in multiple languages and contains the most used Git commands (https://services.github.com/resources/cheatsheets/).
> - GitHub's Git glossary describes Git's most common commands (https://help.github.com/articles/github-glossary/).
> - TryGit, an interactive website for learning the basics of Git (https://try.github.io).
> - Git + GitHub in an R programming context (http://r-pkgs.had.co.nz/git.html).
> - Pro Git book, a complete manual of Git (https://git-scm.com/book/en/v2).

Author contributions

MV and JPC designed the format of the tutorial, MV drafted the manuscript, MV and JPC wrote and approved the final version of the manuscript for submission.

Acknowledgments

References

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

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

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

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

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

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

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

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

Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., Leprevost, F. da V., . . . Vizcaíno, J. A. (2016). Ten Simple Rules for Taking Advantage of Git and GitHub. *PLOS Computational Biology*, *12*(7), e1004947. doi:10.1371/journal.pcbi.1004947

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

Rouder, J. N. (2016). The what, why, and how of born-open data. *Behavior Research Methods*, *48*(3), 1062–1069. doi:10.3758/s13428-015-0630-z

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

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

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

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