

1 Supplement to Curating Research Assets in Behavioral
2 Sciences

3 Matti Vuorre¹ & James P. Curley^{1,2}

4 ¹ Department of Psychology, Columbia University, New York, USA

5 ² Department of Psychology, University of Texas at Austin, Texas, USA

6 Contents

7	Git setup	1
8	The command line	2
9	Setting Git's user information	3
10	Using Git from the Command Line	4
11	Organizing Files and Folders	4
12	Initializing a Git Repository	4
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	Collaborating	11
19	Connect a Local Repository to a GitHub Remote Repository	11
20	Cloning a Remote Repository	11
21	Obtaining other's changes from the central repo	11
22	Resolving conflicts in collaborative work	12
23	Deleting a Git Repository	14

24 Git setup

25 Once you have installed Git, you can use it from the computer's command line or
26 through a GUI. Before using a GUI, it is important to learn a few basic Git commands
27 from the OSs command line, because it is simultaneously the most basic and most flexible
28 interface to Git's functionality. Understanding the basic Git commands as entered through

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

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).

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 once, and the basic functionality requires using only four commands¹. However, users can

¹If 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

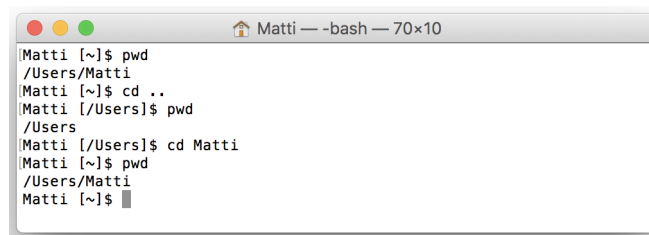


Figure 1. The Terminal command line shell. The functions are explained in more detail in the text. Each command (preceded with a \$ symbol) is followed by its output on the following line. To the left of the \$ symbol, this user's shell application displays the user's user name and the current directory 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.

57 also read through the first part of the tutorial without executing the commands and simply
58 focus on the workflow; we show how to use Git with a GUI in the second part of the tutorial.

59 **Setting Git's user information**

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

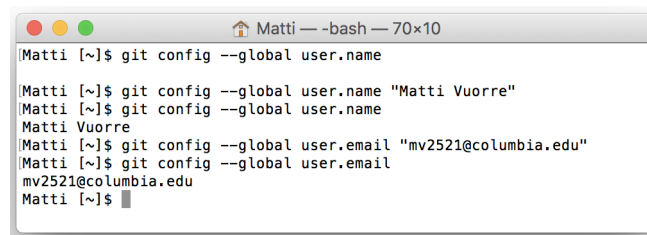
```
$ git config --global user.name
```

64 This should not return anything, unless a previous user of the computer has set the
65 global Git user name. Each Git command starts with the word `git`, then a command (such as
66 `config`), and then arguments to the command, such as `--global` (for global configuration),
67 followed by variables, such as `user.name`. To ensure that Git knows who you are, execute
68 the following command (replacing `User Name` with your desired user name):

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

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

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).

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

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

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

Using Git from the Command Line

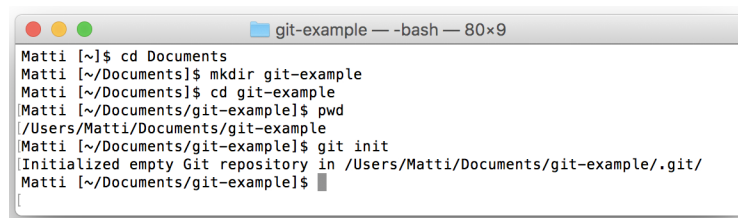
Organizing Files and Folders

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

Initializing a Git Repository

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

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



```

git-example — -bash — 80x9
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.

```
$ git init
```

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

Adding a File to Git

To see what files have changed since the last status change in the repository, 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 ??) 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
```

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

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

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

134 Keeping Track of Changes with Git

135 The `git-example` project (or rather, Git) now keeps track of all and any changes to
 136 README. To illustrate, you can change the text in the README file with a text editor,
 137 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")
```

138 Git can tell that the README file has been modified since the last commit. It is often
 139 useful to know exactly *how* a file has changed, before committing it. To view differences to
 140 a file not yet committed, use `git diff file`. It shows changes within `file`, line by line,
 141 highlighting removed lines of text with red and added lines with green. Once you are happy
 142 with the changes, you can repeat the add and commit steps from above to permanently
 143 record the current state of the project to Git's history (below we use the `.` shortcut for
 144 adding all files with changes²):

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

²For 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.

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
```

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

```

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

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

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

```

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

```

173 (Slightly More) Advanced Git

174 **Make Git ignore files.** To make Git ignore files, you simply add a plain text file
 175 called `.gitignore` to the home folder of the repository. You can use any text editor to
 176 create this file³. Notice that the file is *hidden* (by default, not visible in the OSs file viewer),
 177 but can be seen in the command line with the command `ls -la`. Each row of this file should
 178 specify a file or a folder (or a regular expression) that Git should ignore. In the current
 179 example you could make Git ignore the `admin/` folder entirely (first line in the code listing
 180 below), and any file with the `.pdf` extension inside `manuscript/` (second line). The example
 181 `.gitignore` file would look like this:

```

admin/
manuscript/*.pdf

```

182 Re-running `git status` now only shows the `plan_n.R` file and the newly created
 183 `.gitignore` file, which is also under version control, naturally. Because there are now
 184 two untracked files, which are not specified to be ignored in `.gitignore` (`.gitignore` and
 185 `plan_n.R`), and you usually should aim to maintain a clean commit history for the project,
 186 you can create two separate commits: One for the `.gitignore` file, and one for the power
 187 analysis file.⁴

³You 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.

⁴It 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.


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

188 After this last commit you can, at any time in the future, come back to this commit
 189 with `git log` or `git show` and see what was inside the newly created power analysis file
 190 when it was first created. Below, we show some useful ways in which Git can be used to
 191 “rewind” the commit history.

192 “Rewinding History” with Command Line Git Functions

193 **Try a new feature.** We often find that making some changes to a project didn’t
 194 have the desired effect: The manuscript ended weaker or the analysis didn’t work anymore.
 195 Git allows great flexibility in trying new features, then undoing the changes⁵. Starting with
 196 an empty staging area, you could start modifying a file (e.g. `plan_n.R`) and later realize
 197 that the changes were not good. At this point it is common to press “Undo” in the text
 198 editor, but if the file has been saved multiple times or multiple files have been changed, it is
 199 difficult to get to the starting point by simply using the “Undo” button. Instead, with Git
 200 you can **checkout** the file’s previous version from history. To undo all changes to `plan_n.R`
 201 (since the last commit), run

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

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

208 **Undo committed changes.** Another common scenario is one where a user makes
 209 changes to a file, adds the changes to the staging area, commits them to Git’s history, and
 210 only then realizes that the changes weren’t good. If you have committed changes to a file,
 211 and would like to revert back to an older version of the file, you could **checkout** the file’s
 212 earlier version, and then commit the older version.⁶ For example, suppose you have made
 213 bad changes to a file called `file.txt`, and committed the changes to history, and would
 214 then like to undo the bad changes by reverting to an older version of the file. View the
 215 history with `git log` (you can use the `--oneline` argument for more concise output):

⁵A particularly powerful approach for trying new features is **branching**: The project can be duplicated to a new branch and modified, then merged back to the main branch after work on the new feature is complete—or the new branch can be discarded if the work ended unsatisfactory. Branches are outside the scope of this tutorial, for more information see the Git website (<https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell>).

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

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

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

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

223 Now asking for `git status` reveals that `file.txt` has been modified in the working
 224 directory (its current state is as it was in commit `039d6ff`). You can now add and
 225 commit these changes with `git add file.txt`, then `git commit -m "Undid bad changes
 226 to file.txt"` (type a commit message suitable for your situation).⁷ `git log --oneline`
 227 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
```

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

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

240 Both of these operations—checking out an earlier version of a file or of the entire
 241 project—are “safe” in the sense that your project’s history won’t be affected. However,
 242 checking out an earlier version of a specific file changes the current state of the project (the

⁷If, 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.

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

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
```

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.

Cloning a Remote Repository

To clone a repository, first navigate to an appropriate location on the computer where you would like to create the local 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.

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 ??, because it is important

274 that you start working on the most up to date version of the project (e.g. you don't want to
 275 reinvent the wheel or make unnecessary conflicting changes). Before starting to work on
 276 your proposed changes, pull the remote changes with:

```
$ git pull
```

277 Resolving conflicts in collaborative work

278 Let's assume that a collaborator (User B) has made changes to the `README` file in the
 279 `git-example` project and pushed the changes to the central repository. For brevity, we only
 280 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.
```

281 At the same time, User A might have changed her local version of the `README` file to
 282 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.
```

283 If User A now commits the changes locally, and attempts to push the changes to the
 284 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.
```

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

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

290 The `--rebase` argument turns Git into rebasing mode, meaning that you are now
 291 applying your local commits on top of the "base" obtained from the remote repository.
 292 You can `git pull` without the `--rebase` argument, but that would create an unnecessary

293 commit message and lead to a slightly different workflow⁸; we recommend using the `--rebase`
 294 argument. At this point, Git is in “rebasing” mode, allowing User A to resolve the conflict
 295 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")
```

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

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

304 The first line of the file was identical across the two Users’ versions of the file, and
 305 therefore remains the same. However, after the first line, there is first a line (`<<<<<<< HEAD`)
 306 indicating that what follows are the to-be-integrated lines of text. Anything after this line
 307 up to the `=====` line is the to-be-integrated text from the central repository. We can see
 308 that this is simply the line of text that User B created (“Hello world!”). After the separating
 309 line (`=====`) are User A’s local changes, followed by those changes’ commit message (“Some
 310 meaningful changes”) prepended with a `>>>>>>>`. User A can then edit this file however she
 311 chooses, using the tags to help her see which are her lines of code (text), and which are User
 312 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.
```

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

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 its 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 tutorial⁹. 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.

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.

⁹Covering 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.