



GIT IT TOGETHER!
VERSION MANAGEMENT FOR RESEARCH
PROJECTS



Paul E. Johnson, CRMDA <pauljohn.edu>
Benjamin Kite, CRMDA <bakite@ku.edu>
Kenna Whitley, CRMDA <kennamarie@ku.edu>

Guide No: 31

Keywords: Git

Aug. 19, 2020

See crmda.ku.edu/guides for updates.

Cheat Sheet

Basic Usage

`git clone` - Copies a version-tracking repository, including all of its history.
`git init` - Initiates a new version-tracking repository in current working directory.
`git pull/push` - Keep up to date with remote repository (retrieve/send).
`git add` - Tell Git to begin monitoring a file.
`git commit` - Tell Git to take a “snapshot” of altered files.
`git status` - Ask for report on files in project. Suggest “`git status .`”.
`git log` - Ask for history report on project.

Intermediate Usage

`git branch` - Lists branches
`git branch branchname` - Creates a branch named “branchname”
`git checkout branchname` - Opens the branch “branchname”
`git fetch` - Downloads changes and stores them in the `.git` folder. Does not alter files in working directories
`git merge branchname` - Retrieve “branchname” and apply its changes to the currently checked out branch

Contents

1 Overview	4
2 Advantages of Version Management	4
3 Installing Git Software	6
4 Helpful Webpages	7



I	Basic Knowledge and Operations	7
5	Three Scenarios for Git Use	7
5.1	Use the Terminal to Interact with Git	8
5.2	Scenario 1: Track a remote repository	8
5.3	Scenario 2: Create a local “change tracking” repository	10
5.4	Scenario 3: This is the CRMDA Project Workflow: Participate in Projects	12
6	Essential Git Tools	18
6.1	git log (print project history)	18
6.1.1	Arguments to alter behavior of git log.	18
6.2	git status (check if files are committed or tracked)	19
6.3	git rebase (Combine commits to simplify history)	20
6.4	git branch (Review branches)	22
6.5	git checkout (open files in a branch)	22
6.6	Create a Branch	23
6.7	Delete a branch	23
6.7.1	Delete a branch	23
6.7.2	Delete the same branch from the remote repository	24
6.8	Conceptual Framework: Local and remote Branches	24
6.9	git fetch(retrieve abranches)	27
6.9.1	fetch --prune (remove branches that no longer exist on remote)	27
6.10	git stash	28
6.11	git diff (compare versions)	29
6.11.1	Is my current working directory up to date with the remote repository? . . .	30
6.11.2	Experiment with gitk	30
6.12	git mv (move, rename)	31
6.13	git rm (remove files)	31
6.14	git merge	32
6.15	git show (recover previous versions)	32
6.16	git tag (labels for important commits)	33

7	Troubleshooting Pull Fails, Push Fails	33
7.1	Push Fail: out-of-date local repository	34
7.2	Pull Fail: A Fortunate Outcome May Occur	34
7.3	Git pull causes merge conflict	37
7.4	Git pull refused: locally untracked files	37
7.5	Git pull refused: uncommitted local revisions	37
7.6	Manually fixing merge conflicts	38
8	User Conveniences (Graphical Interfaces)	40
8.1	Windows Explorer Tortoise	41
8.1.1	Tortoise interacts with branches	42
8.2	Emacs	42
8.2.1	Commit via Emacs	42
8.2.2	Retrieve an old file (interacting with history)	43
8.2.3	Integrate with Emacs ChangeLog	43
9	We use Git-LFS for binary files	43
II Advanced Knowledge and Operations		43
10	Getting Particular Things Done	43
10.1	Recover lost files (or re-set accidentally edited files)	43
10.1.1	Uncommitted revisions: restore individual files/folders	44
10.1.2	Remove all uncommitted changes (not just particular files/folders)	44
10.1.3	Committed revisions: retrieve one file that was removed	44
10.1.4	git reset --hard to restore files in a project	45
10.1.5	git reset --soft to undo a commit, but leave files unchanged	45
10.2	Copy one file from a branch	46
10.3	Reverse accidental edits on the wrong branch.	46
10.4	Undo Merge	47
10.5	Re-sequence edits “on top” of master (rebase)	48
10.6	Interactively merge revisions	49

11 Concluding Advice	50
11.1 Don't Get Carried Away by Russian Teenagers	50
11.2 Let us Know What You Find Out (We'll Let You Know What We Find Out)	50
11.3 Things to Not Do	51
11.4 Things to Do	51

1 Overview

Git is “version tracking” software. It can be used to:

- monitor changes in files over time
- “step back in time” to get old versions
- allow teams to coordinate contributions of many workers

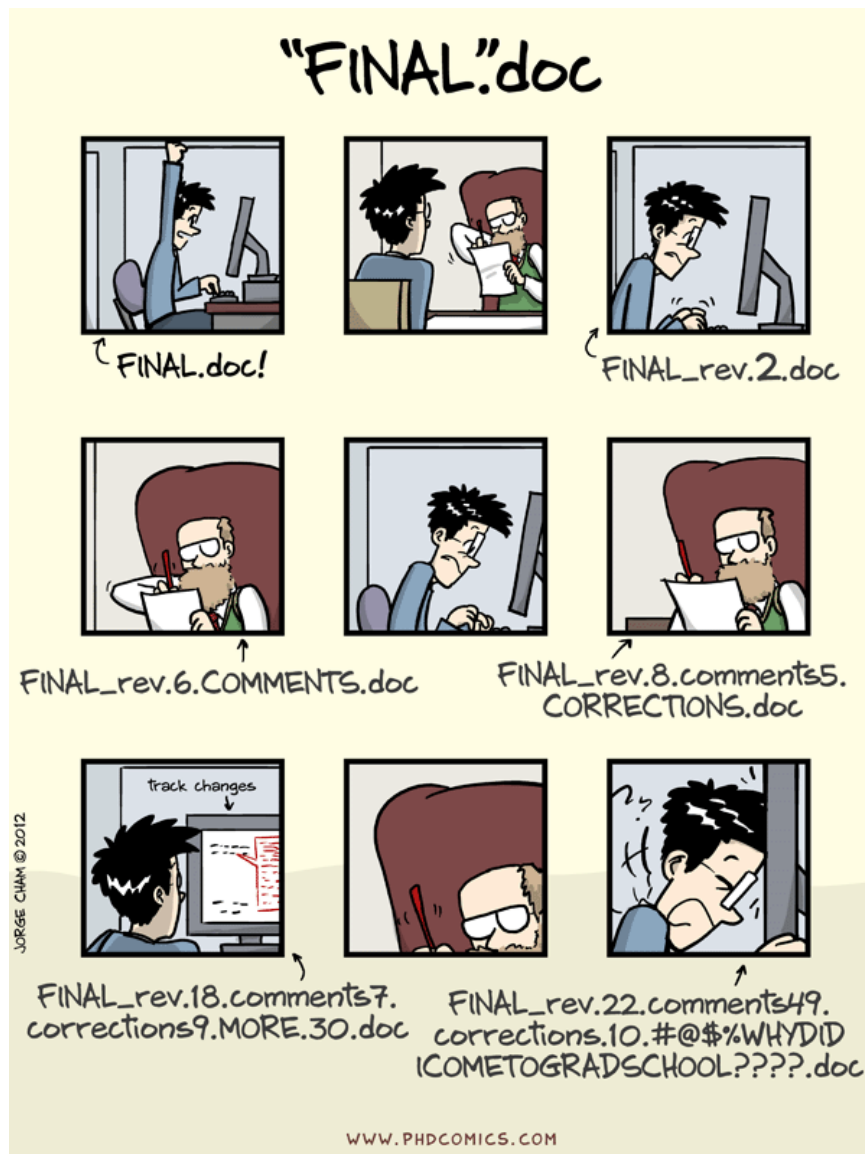
Git is currently the most widely used version management system. Git is a free, cross-platform, and open source version control system that was invented by Linus Torvalds, the creator of the Linux kernel. Git has many powerful features that are designed for the harmonization of entries by 10s or 100s of workers.

2 Advantages of Version Management

While we have written 1000's of words about it, the cartoon in Figure 1 conveys the message entirely.

Git keeps track of all of your changes and allows you to grab any past version of a file. Further, Git allows teamwork – several people can edit the same set of files at once and the system tries to reconcile the changes. The system tracks who makes changes, it asks them to explain the changes they make, and it allows rollbacks.

Figure 1: Why Git is Necessary



We DO NOT want to fill up folders with versions of a single file. We want one file that benefits from version management. Any previous version can be retrieved.

Glossary: Absolutely vital terms everybody has to tolerate

There are quite a few “cheat sheets” and “Git guides” (this document!). Please see Figure 2. Here are the essential terms:

working directory The directory inside the user’s computer where files are edited.

local repository A hidden directory called “.git” where changes are tracked. It is *INSIDE* the working directory. It is a database, not a human-readable set of files.

remote repository A server from which revisions can be retrieved. For testing, we can create a “remote” on our own personal computers, but for real-life effort, we always have a remote server. Currently, we have a GitLab server.

origin The nickname of primary Git remote repository. Many remotes may exist.

branch A collection of files with a directory structure and historical records.

master The default branch that all Git projects have.

commit To enter the current version of a file in the history log (i.e., tell Git to take notice of your changes!).

head The current position of the file set within the repository’s history.

tracked files Files which have been added into a repository.

staging area Tracked files that have been revised that are not yet committed. The staging area is essentially the same as the working directory in the ordinary use of that term.

3 Installing Git Software

Windows Get the *real Git* from <http://git-scm.com/downloads>. It is delivered with “Git BASH”, a Unix-style “terminal emulator”.

Macintosh Install either the Xcode tool set (the easiest approach), the individual components of a command line Git environment from <http://git-scm.com/downloads>, or Homebrew, a free-software delivery system for Macs.

Don’t install other convenient Graphical User Interfaces (GUIs)

We emphasize using Git in the command line. There are graphical interfaces for using Git, but we avoid using them. We discourage our research assistants from becoming dependent on them. These things work correctly about 85 percent of the time. For the other 15 percent, the command line is truly necessary. To keep command line skills sharp, use it as often as possible.

Do use a programmer’s file editor that can interact with Git

Use a programmer’s file editor, such as Emacs, that can interact with files in a Git repository. Many IDEs will have have similar features. The main purpose here is that authors can ask Git to take snapshots as they work, without distracting themselves too much from the substance of their project.

Many editors will work best if the operating system supplies other helper functions that make it possible to compare files and isolate the sections that have been changed. Emacs will work best if the GNU program `diff` is also installed (it compares text files). The executable `diff` is generally installed in all Linux systems and is easily available on Macintosh. It is also available separately for Windows as a part of the `diffutils` package (see <https://crmda.ku.edu/setup>).

4 Helpful Webpages

- The official Git guides: <https://git-scm.com/doc>
- Git Manual Page <https://www.kernel.org/pub/software/scm/git/docs/>
- Git Reference: <http://gitref.org>
- Git - the simple guide: <http://rogerdudler.github.io/git-guide>
- Become a git guru: <https://www.atlassian.com/git/tutorials>
- Git Cheat Sheets
 - Git-tower website: <http://www.git-tower.com/blog/git-cheat-sheet>
 - GitHub Training: <https://training.github.com/kit/downloads/github-git-cheat-sheet.pdf>
- Undo (Almost) Anything: <https://blog.github.com/2015-06-08-how-to-undo-almost-anything-with/>
- KU CRMDA GitLab Guide: <http://crmda.dept.ku.edu/guides/34.GitLab/34.gitlab.pdf>

Part I

Basic Knowledge and Operations

5 Three Scenarios for Git Use

We are trying to write down just enough so that Git beginners can participate effectively. There are three especially common scenarios.

1. “Track a Remote Repository”. This is not for a project contributor. It is for somebody who wants to monitor a project without contributing. The git **clone** will be a snapshot of the project’s entire history. The git **pull** functional will download updates. This can be used to track course notes or software development on, for example, GitHub, GitLab, or BitBucket. Generally, authors of those projects will not allow contributions. Sharing revisions back to the author requires a somewhat tedious process known as “forking”, which we do not discuss in these notes.
2. “Track Your Own Project”. This does not use a remote server. It is simply for personal record keeping. Authors add files and make notes on their revisions as they go. We expect all graduate research assistants who work our center will track their efforts on all projects, whether or not they are working alone and whether or not they are using a remote server.

3. “Server Integration”. Interacting with a remote server, usually in coordination with team mates. The efforts of many workers can be harmonized. Revisions are **pushed** to the remote. Updates are **pulled**. Because the efforts of team members may conflict with each other, it is often necessary to handle “merge conflicts” and develop team rules about who is editing what.

We believe most readers of this document will probably never need to create their own remote “bare” repositories. The rise of convenient server-based Web tools, like GitHub and GitLab, has made it unnecessary.

5.1 Use the Terminal to Interact with Git

How to start a Terminal? When installing Git for Windows, a terminal they call “Git BASH” is installed. It can be launched by navigating in the Windows Explorer to a desired directory and right clicking. Macintosh has a Terminal program in the Utilities folder, but there are plenty of other terminal programs. On Linux, of course, there is a seemingly endless supply of terminal emulators. It is not truly necessary to use BASH as the shell program, but it is fairly common at the current time.

Suggestion: Create a GIT directory

This is for personal organization. Create a folder where all Git repositories can be saved. This avoids the “where did I download that repository?” problem.

5.2 Scenario 1: Track a remote repository

Professors or program authors make Git repositories available, but they don’t expect us to revise them and contribute back to the project. We’ll show how to download and stay up to date.

We will track one repository on GitHub. The Web interface for this is <https://github.com/pauljohn32/RHS>. It is a collection of exercises and course notes.

Make sure you are in a terminal and the current working directory is “GIT”.


The RHS repository can be cloned by running this:

```
$ git clone https://github.com/pauljohn32/RHS.git
```

Here’s the response:

```
Cloning into 'RHS'...
remote: Counting objects: 871, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 871 (delta 21), reused 37 (delta 11), pack-reused 824
5 Receiving objects: 100% (871/871), 10.14 MiB | 7.81 MiB/s, done.
Resolving deltas: 100% (353/353), done.
```


Caution: https versus ssh-based clones

GitHub allows either https or secure shell based downloads. That is apparent in their graphical interface, by the bright green button: . We chose “Use HTTPS” and we found the required address in the panel:



You could use SSH security instead. If you have a GitHub account, and your SSH key is registered with them, one could click “Use SSH” and the appropriate clone address would change to.

```
$ git clone git@github.com:pauljohn32/RHS.git
```

After the clone, inspect.

The top level directory “RHS” should exist, and it will have a “hidden” folder named “.git” along with the author’s contents. In the terminal, change into the RHS directory and list the files with the “-lah” flags (“l”=include details, “a”=include dot files, and “h”=human readable file size):

```
$ cd RHS
$ ls -lah
```

```
total 32K
drwxrwxr-x 6 pauljohn32 pauljohn32 4.0K Feb 17 12:18 .
drwxr-xr-x 3 pauljohn32 pauljohn32 4.0K Feb 17 12:18 ..
drwxrwxr-x 43 pauljohn32 pauljohn32 4.0K Feb 17 12:18 exercises
drwxrwxr-x 8 pauljohn32 pauljohn32 4.0K Feb 17 12:18 .git
-rw-rw-r-- 1 pauljohn32 pauljohn32 242 Feb 17 12:18 .gitignore
drwxrwxr-x 6 pauljohn32 pauljohn32 4.0K Feb 17 12:18 guides
drwxrwxr-x 4 pauljohn32 pauljohn32 4.0K Feb 17 12:18 notes
-rw-rw-r-- 1 pauljohn32 pauljohn32 810 Feb 17 12:18 README.md
```

Note the directory `.git`, along with the other materials provided by the project.

Check the project history

To review the project history, run:

```
$ git log
```

This will show the project entries, one by one, in page-sized sections. The letter “q” can break out of a long listing display.

Staying up to date To stay up to date with the author’s effort, open the RHS folder and run:

```
$ git pull
```

This will succeed, almost always.

Caution: Don’t edit the author’s files

If you edit that author’s files, then git will refuse to pull the updated versions. Git does not want to throw away your edits by putting the author’s version on top. To fix that, one can restore the author’s file to their original condition, as explained in section 10.1.

Caution: Don’t edit or damage contents in the hidden folder “.git” The hidden directory named .git includes all of the project’s history. All commands, such as “git add”, “git log”, “git status”, and so forth, interact with material in the .git directory. Don’t damage that folder.

5.3 Scenario 2: Create a local “change tracking” repository

This local repo will be used to track some files. It does not link to a remote server. Use this to practice your Git skills. It keeps all of your records in the .git directory, which will be in the top level folder.

We suggest you proceed as follows:

1. Begin by opening a terminal. Navigate to the folder where you keep all of your GIT projects. Make a directory there. For example:

```
$ cd GIT
$ mkdir smith_jones
$ cd smith_jones
```

2. Create a repository:

Run the following:¹.

```
$ git init
```

Git should say

```
Initialized empty Git repository in <location and name of your
directory>
```

3. Add Files

We created a file named “basic.txt”, and here we add this file to be tracked by git:

¹If you have teammates who might edit this repository in a networked file system, add the command line argument “- -shared=group”.

```
$ git add basic.txt
```

Please note, adding the file does nothing except make Git aware that you plan to track that file. It is “staged” but not “committed” (the next step).

Any files that you have not added to the repository will NOT be included in the repository’s history. They appear as “untracked files.”

4. Commit that file.

Lets keep this simple. Commit the file and insert the “commit message” with it in the command line

```
$ git commit basic.txt -m 'basic.txt: initial file version'
```

If several files were edited, they can all be committed in one step by running the following

```
$ git commit -a
```

This triggers an editor to open and Git wants a message to describe the changes. See Caution 2 below about that process.

Caution 1: Only add files you truly need to track.

Add files one-by-one until you have some practice. Don’t add whole directories, don’t add password files, don’t add backup files or trash files.

Please do not commit confidential data to Git. If clients send us some data with which to work, we generally DO NOT check that into Git because

1. we don’t want copies of the data traveling along with the Git repository,
2. we don’t edit client data files, there is no need to track changes in them.

Usually, what we want researchers to do is “mount” or “symbolically link” the data folder into the project directory in order to access the data.

Caution 2: An editor will open if you do not include the “-m” argument

Git’s default editor is Vi, but Git may be configured to interact with other editors. Windows users may prefer Notepad++. Two options are:

Cancel the commit: When the editor opens, the commit can be canceled by closing the file without saving it.

Complete the commit: edit the commit message, save the file and close the editor.

About Vi: If the default message editor in Git is Vi. Vi may be unfamiliar, but *don’t panic*. You are not alone. “Editor Hell” is a nickname created by one of our young staffers who had never seen Vi before. Vi is an editor from the days before computers had mice. It is completely keyboard driven. If you end up in Vi, here’s what to do:

- Hit the letter “i” to enter **insert mode**. You will see “– INSERT –” on the bottom left.
- Cursor should be in line 1, column one. Type your message.
- Now save your message through this seemingly bizarre sequence of keys
 - Hit the **Escape** key (breaks out of insert mode)
 - Hit the **colon** key (should put the cursor at the bottom of the screen)
 - Type “**wq**” to *write* and *quit*
 - Hit the **Enter** key

You could customize your computer’s Git setup to use a more pleasant editor.

Caution 3: Combining “-a” and “-m” is dangerous

When users run “`git commit -a`”, the editor will open and display a list of files. If there were accidental edits, or if files were accidentally added to Git tracking, it will become apparent at that point. Canceling the edit session will cancel the commit.

Inserting a command line message in combination with “-a” eliminates the editor phase, taking away the author’s ability to review the files that are being committed. In short, we suggest users should not run with scissors and avoid typing `git commit -a -m “my big mistake”`.

5.4 Scenario 3: This is the CRMDA Project Workflow: Participate in Projects

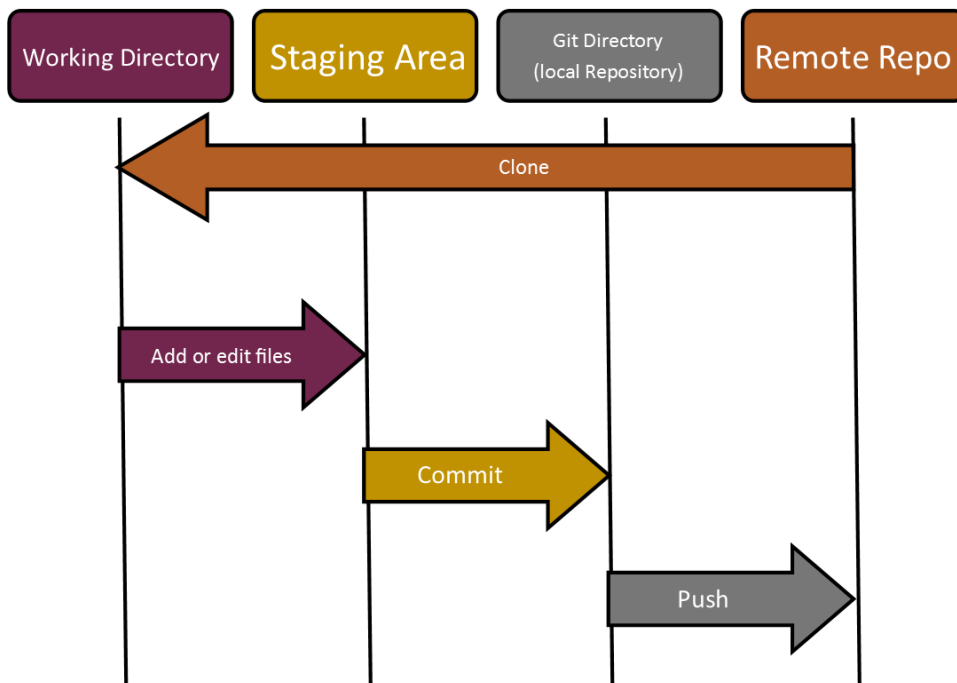
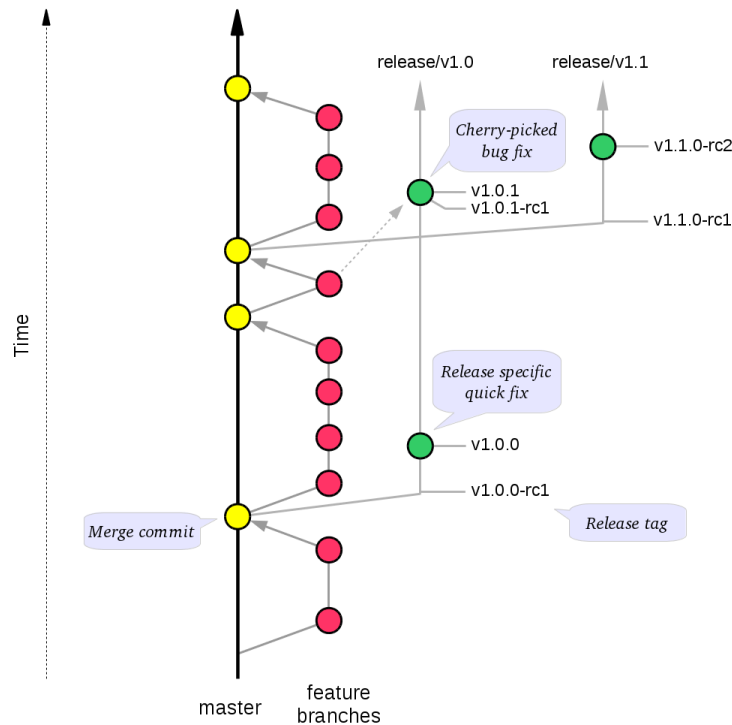


Figure 2: Git Illustrated

Figure 3: Stable Mainline branching model (source: [Stable mainline](#))



Our workflow is similar to the “GitLab Flow” or the “Stable Mainline” branching model. The premise is that the master branch is the perfected, most up-to-date working version, and most (almost all) branches are short lived enhancements (“feature branches”) that will be finished, merged onto master, and deleted. Long-lived branches exist only in the rare (for us) case that a previous release version of a project must be corrected the revision does not fit within the long term master plan. In Figure 3, we have a copy of the drawing from the Stable Mainline webpage and this summarized the way we think about it rather well. In CRMDA, we’d almost never have retroactive repair branches like “release/v1.0”.

A project manager creates the Git repository on the server. Team members do not concern themselves with creating “bare repositories” or any of the security issues of regulating team participation. That’s the manager’s job. The manager will provide the address. In our case, we always use SSH keys for security.

The main issue is that it is difficult to manage teamwork. When several users are editing an interconnected set of files, it is easy for the users to fall out of step with one another. There are some technical fixes for these problems, but the most important fixes are sociological and organizational.

GitLab at CRMDA

We use a GitLab server that is located at <http://gitlab.crmda.ku.edu>. People who have KU online IDs can register for accounts there. The process is described in our [guide on using GitLab](#)

(CRMDA Guide #34). We require project participants to use SSH security. Contributors must register SSH keys with this server.

Our server will allow non-contributors to use HTTPS to pull a copy of a repository, but people who want to push changes back to the server must use SSH security. For example, one can retrieve a copy of our working examples for High Performance Computing like so:

```
git clone git@gitlab.crmda.ku.edu:crmda/hpccexample.git
```

In case you see these Warnings, they are harmless.

```
Warning: untrusted X11 forwarding setup failed: xauth key data not
generated
Warning: No xauth data; using fake authentication data for X11
forwarding.
```

Step 1. Clone the repository

Consider, for example, we clone a copy of a Web Scraping project:

```
$ git clone git@gitlab.crmda.ku.edu:crmdaprojects/Ticket-790-Ukraine_web_scrape.git
```

```
Cloning into 'Ticket-790-Ukraine_web_scrape'...
X11 forwarding request failed on channel 0
remote: Counting objects: 270, done.
remote: Compressing objects: 100% (77/77), done.
5 remote: Total 270 (delta 220), reused 202 (delta 191)
Receiving objects: 100% (270/270), 207.53 KiB | 7.69 MiB/s, done.
Resolving deltas: 100% (220/220), done.
```

That will create the directory named “Ticket-790-Ukraine_web_scrape” in the current working directory.

In case you already have a directory and you simply want to copy the project files into it, add a period at the end of the command. The period at the end represents the current working directory. This command is one line (even if it wraps in this document).

```
$ git clone git@gitlab.crmda.ku.edu:crmdaprojects/Ticket-790-Ukraine_web_scrape.git
.
```

Step 2. Inspect your cloned repository

Run `git log`, and `git status`, and `git branch`, as discussed below.

Step 3. Create your own branch

A branch can be created, and “checked out” in one shortcut step. This will create a branch named “pj-fix”.

```
$ git checkout -b pj-fix
```

See more about creating, naming, monitoring, and deleting branches in the following section.

In our projects, we want branches to have short names that use the initials of the author in the beginning. This makes it easier for us to understand what is what.

Step 4. Edit, add files, and commit the changes

Some teamwork will help avoid future problems. If two team members have branches that edit the same file, it is likely to be complicated to integrate their work later on. As a result, *unless it is necessary* to do otherwise, we suggest that only one team member should have responsibility for editing each file.

Step 5. Make sure your branch is up to date with the master branch

We want workers to run the Standard Four Step Update Sequence early (Table 1) and often.

Here’s why. Work on your branch may be irrelevant or out of date.

See Figure 4a. The branch `pj-fix` split from master, but master evolved. The other team members are busy making revisions and putting them on the master branch. The `pj-fix` branch has grown and made changes as well.

When the work on `pj-fix` is done, the author intends for it to merge with the master branch. The eventual merge will fail, however. The branch fixes don’t apply to master in its current state. Notice that in Figure 4b, the merge up from `pj-fix` to the master branch is trying to blend a branch from which master has wandered away. It is not likely that master—as `pj-fix` remembers it—is compatible anymore.

The author of the branch has the duty to make sure that the branch is still beneficial for the larger project. The author should run the “Standard Four Step Branch Update Sequence” (Table 1) to bring the branch up to date with the master branch.

Table 1: Standard Four Step Branch Update Sequence

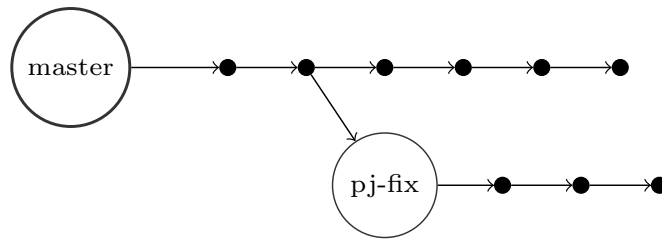
```
$ git checkout master
$ git pull
$ git checkout pj-fix
$ git merge master
```

Immediately after that merge, the author can fully understand the impact of the changes that the branch will have for the project. If there are merge conflicts, they must be attended to (see section 7.6).

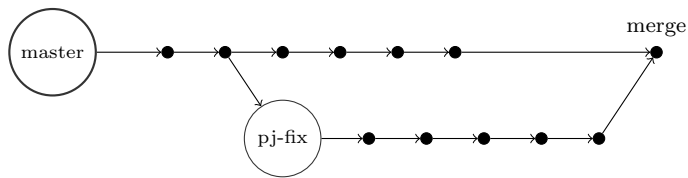
The diagram in Figure 4c represents this new logic.

Figure 4: A Team Member's Branch

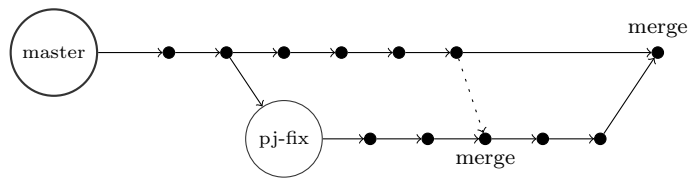
(a) A branch named pj-fix



(b) Merge error likely



(c) Four step updated branch succeeds



Step 6 Push changes back to main repository

When the author makes the first push to the server, it is necessary to tell the server (AKA “origin”) that the current branch should be pushed to the remote server as a branch of the same name.

```
$ git push -u origin pj-fix
```

The flag `-u` is short for `--set-upstream`. In pushes after that, git will remember where the branch is intended to go, and it is sufficient to run

```
$ git push
```

Caution: there are now local and remote versions of the branch

This is the point at which the branch flags `-avv` (“-a”=show all, “-vv”=very verbose) become especially helpful in understanding the separate branches. Once the branch `pj-fix` is pushed to the server `origin`, the user’s branch list (output from “`git branch -avv`”) will show 2 branches, one named `pj-fix` and one named `origin/pj-fix`. If the author has `pj-fix` checked out and keeps editing, then the local “tracking branch” `pj-fix` is now out of date with `origin/pj-fix`. The next time the author runs `git push`, the two will be re-synchronized.

Step 7 Make a merge request

In GitLab, there is a graphical interface that can be used to notify the project manager that a branch is ready to be merged onto the master branch.

Pressure the manager to pay attention and do the merge

In our experience, there is one main concern. The author of the branch pushes it to the server and requests a merge. The project manager is not aware of the request (email gets lost, etc), and the branch is not merged and destroyed promptly. The master wanders away from the branch again and then the branch has to be re-updated. Team members who fear this should speak with the repository manager to make sure she/he is aware of the need to merge the branch promptly.

Be Prepared: your branch will disappear from the server

The project manager wants to keep the project clean by deleting branches wherever possible. Generally, the “remove merged branches” policy is in effect.

When the user runs “`branch -avv`”, the branch `origin/pj-fix` will still seem to exist, even though the manager has in fact deleted it. Adding the “prune” option to a fetch command will delete `origin/pj-fix` from the list:

```
$ git fetch -p
```

The fetch command is discussed in more detail below.

6 Essential Git Tools

In this section, we will learn about common chores. Review your situation with “git log”, “git status”, and “git diff”. We have not discussed Git branches in detail yet, and though some of the output here does mention branches, we think users are going to benefit from these commands even if they don’t yet fully understand them.

6.1 git log (print project history)

Display the commit messages, one by one:

```
$ git log

commit d64553e21c06dff97502a12a5cdfdb3a828385ef
Author: Paul E. Johnson <pauljohn@ku.edu>
Date:   Fri Feb 13 15:11:05 2015 -0600

    updates for version 1.8.92, EIS eliminate

commit 670ab2b24dc686a325d3b1c352f5b710e875cb31
Author: Paul E. Johnson <pauljohn@ku.edu>
Date:   Fri Jan 23 10:54:29 2015 -0600

    Rework summary and print methds, insert code example to use
    tables/tabular.

commit 8fe813e5e040efd89870248da3812eac6f406657
Author: Paul E. Johnson <pauljohn@ku.edu>
Date:   Mon Jan 19 19:19:03 2015 -0600

    Updating ptable
```

Those long commit numbers are unique identifiers known as SHA1 values. We can retrieve, for example, the full version corresponding to entry “8fe813e5e040efd89870248da3812eac6f406657”. We don’t need to specify the whole number, the first four to six letters will suffice: “8fe813”.

6.1.1 Arguments to alter behavior of git log.

For more concise output, run

```
$ git log --oneline
d64553e updates for version 1.8.92, EIS eliminate
670ab2b Rework summary and print methds, insert code example to use
        tables/tabular.
8fe813e Updating ptable
```

By default, the Git log includes commits in the history leading up to the current file set (a branch). To view all commits in the repo regardless of branch, run

```
$ git log --all
```

In order to view both commits, their associated tags, and which commit HEAD, the remote branch(es), and the local branch(es) point to, run

```
$ git log --decorate
```

All these, or a number of the above, can be combined to produce very concise and efficient log outputs:

```
$ git log --all --decorate --oneline
```

6.2 git status (check if files are committed or tracked)

Sometimes a programmer wonders, “are these files being tracked?” and “Do these files need to be committed?”

If none of the tracked files were edited, this is the result from running

```
$ git status
On branch master
nothing to commit, working directory clean
```

The output will be more substantial if files have been edited. Here we have edited “newfile.txt”, which was added to tracking and committed previously.

```
$ 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:   newfile.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

The comment “Changes not staged for commit” means that tracked files were edited but not committed. Git notices the file is different from the version stored in the repository.

Repositories that are linked to remotes have more elaborate output. Here is example output from a working directory that holds the CRMDA workshop materials:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
```

```
5 Untracked files:
  (use "git add <file>..." to include in what will be committed)

    data/els/R/.Rhistory
    data/els/R/analysis-1.R
10   data/els/R/analysis-1.html
```

Your branch is ahead of origin/master means that we have already committed 2 changes, but they were not yet pushed to the server. We will try to explain the meaning of `origin/master` in section 6.8 below.

Untracked files means Git noticed that we have files that are not being tracked.

Because the remote server is involved, the output from “git status” will always be a little different. Here is what we see in an up-to-date working directory:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
```

The complexity in the output of the status command follows from the fact that a file can be in states as far as Git is concerned.

1. File is not added to Git tracking.
2. File is added, but not committed.
3. File is added, and committed.
4. File was previously added and committed, but it has been edited since then.
5. The file was edited, committed, but not yet pushed to the remote server.

6.3 git rebase (Combine commits to simplify history)

We encourage code-writers to make frequent commits, so that they can compare the impact of edits. When work is done, there may be 30 commits that can be combined into one single “patch”. This is especially true when a particular feature is being developed. There is no reason that the teammates—or history—should have to wade through 30 commit messages like “trying method 1”, “fixing error in method 1” and “method 1 nearly works”.

The commits can be squashed into one commit. There is a nice writeup called “[Squashing commits with rebase](#)” by Nick Quaranto. Our notes about this are similar, <http://crmda.dept.ku.edu/guides/31.git/31.git-squash.md>, but we have just a few additions from the school of hard knocks.

The basic idea is this. Look at the log:

```
$ git log --oneline
6255521 (HEAD -> my-new-branch) method 1 works
7549bd2 method 1 nearly works
611ff1e fixing error in method 1
5 46d6518 try method 1
```

This shows we want to squash together the last 4 commits. Note we put the number 4 after the tilde:

```
$ git rebase -i HEAD~4
```

That will open an editor with a display like the following

```
pick 46d6518 try method 1
pick 611ff1e fixing error in method 1
pick 7549bd2 method 1 nearly works
pick 6255521 method 1 works

# Rebase ea9df6f..6255521 onto ea9df6f (4 commands)
```

We change “pick” to “squash” (or “s” for short) in lines 2-4

```
pick 46d6518 try method 1
s 611ff1e fixing error in method 1
s 7549bd2 method 1 nearly works
s 6255521 method 1 works
```

When the rebase editor closes, another git message editor will open. It will show all 4 commit messages in one section and we can edit this. We suggest a commit message like “method 1 worked as anticipated and on the very first try!” None of the teammates will ever know the difference. After that, history will be compressed. First, the “git log” will show just one element.

Why do we want this? To clean up history before pushing!

Caution: Never do this after you have pushed to the remote.

Do this if you have a lot of small commits that you did not push yet. Do not do it after pushing to a server. If teammates have downloaded the revisions, then rebasing the history may leave them in a state of disorientation.

Caution: Do not delete lines from the first rebase editor window

Do not delete items from the “pick/squash” list of commit messages. If you delete an item, *it will delete that source code change*. All items must be left either “pick” or “squash”.

6.4 git branch (Review branches)

The following indicates that there are two branches, `master` and `cx-41`.

```
$ git branch
  cx-41
* master
```

The “git branch” command will be discussed in greater detail in section 6.8.

6.5 git checkout (open files in a branch)

The effect of the “git checkout” command is a little bit difficult to fathom. In essence, it copies the tracked files into the current working directory. Suppose we checkout `cx-41`:

```
$ git checkout cx-41
```

All of the changes that exist in that other branch will magically appear in the files of our current working directory. Untracked files will remain in the directory. If `cx-41` does not have files that are in my current branch, those files will *seem to disappear*.

Checking out a branch should not cause anything to be permanently lost. If I change back to the other branch,

```
$ git checkout master
```

the files will reappear.

Caution: checkout can fail

A request for a Git checkout will often be refused if files that are being tracked have not been committed. The following is an example of that kind of error:

```
$ git checkout cx-41
error: Your local changes to the following files would be
  overwritten by checkout:
    31.git/31.git.lyx
Please commit your changes or stash them before you switch branches.
5 Aborting
```

In that case, the instructions from Git are helpful. Save and commit the edits, or move them out of the way by stashing them (see section 6.10). If your branches are tracking different files, there may be problems when you alter a file that is tracked in one branch but is a locally untracked variable in the other.

Caution: Untracked files in the working directory

If there are files that are not added to Git tracking, they will be present in all branches. This can cause problems when we forget that programs in one branch might alter that untracked file. There will also be a conflict if the file is added on one branch, but not the others. The checkout will fail because changing to that branch would alter an untracked file. This is an easily solved problem (either rename the untracked file or add it to all of the branches), but it can be confusing.

6.6 Create a Branch

There are two ways to do this. First, we demonstrate a two step process that creates the branch and then checks it out.

```
$ git branch jj-fix2
$ git checkout jj-fix2
Switched to branch 'jj-fix2'
```

Note: there is no return value (“confirmation”) from the first command.

The two-step method of creating and switching onto a branch can be replaced by a single step.

```
$ git checkout -b jj-fix2
Switched to a new branch 'jj-fix2'
```

6.7 Delete a branch

6.7.1 Delete a branch

Deleting a branch does not remove it from the project history. However, it removes it from the current view of active branches displayed by commands like “`git status -a`”. Here we will delete the branch “`jj-fix2`” that was created in previous subsection.

```
$ git branch -d jj-fix2
Deleted branch jj-fix2 (was b6af81e).
```

Caution: Branch Deletion can fail

If the branch is checked out, Git will refuse to delete it:

```
$ git checkout jj-fix2
Switched to branch 'jj-fix2'
pauljohn@delllap-16:RHS$ git branch -d jj-fix2
error: Cannot delete branch 'jj-fix2' checked out at '/tmp/RHS'
```

The fix is to check out another branch, say master.

Also, if the branch has changes that are not committed, Git will refuse to delete the branch. One should either commit the changes on the branch and push results to the remote (to finalize records on the project effort) or force the deletion by running “`git branch -D jj-fix2`”.

6.7.2 Delete the same branch from the remote repository

If your branch exists on the remote server, the easiest way to delete it is to run this command

```
$ git push origin --delete jj-fix2
To gitlab.crmda.ku.edu:crmda/guides.git
- [deleted]          jj-fix2
```

If your remote repo is running under GitHub or GitLab, then it is also possible to use their Web graphical interfaces to delete the branch. But, honestly, this is easier.

6.8 Conceptual Framework: Local and remote Branches

List the branches:

```
$ git branch
  cx-41
* master
```

These are the branches that have been checked out at some point on this computer.

If there is a remote server involved, additional branches exist, both inside the local repository and, of course, on the remote. When we add the flag “-a”, we see additional branches that were found on the remotes.

```
$ git branch -a
  cx-41
* master
remotes/origin/HEAD -> origin/master
remotes/origin/KH_Guide_34
remotes/origin/cx-41
remotes/origin/master
```

The branches prefixed with “remotes/” were found the last time we fetched. They were downloaded and the records were placed in `.git`. After adding `-a`, we see clearly that the local branch `cx-41` has a doppelganger on the server.

Also we see branch that we were not aware of before, named `origin/KH_Guide_34`. It is, clearly, on the remote and now exists in our local file set. It is important to remember, then that the ones prefixed with “remotes” are only displayed if we add the “-a” flag. They are there, however, in the local repository, all along.

First, consider the “local tracking branch”

The name “local tracking branch” refers to a checked-out version of a remote branch. A local tracking branch is `cx-41`, while the remote snapshot of the same-named branch, `origin/cx-41`, also exists. Here are some of the frequently asked questions.

- Question: Where did `cx-41` come from?

- Answer: There are two possible sources.
- 1. It was created by another user (or same user on another computer) and it was pushed to the remote. After the repository was fetched, and the user ran “`git checkout`” to open a local copy.
- 2. On the local system, the author created the branch, say “`git checkout -b cx-41`”, and then made some edits/commits and then ran “`git push -u origin cx-41`”. The last command makes a copy of it in the “remotes” section that displays in “`git branch -a`” and puts it on the server.
- Question: Are the two branches (`cx-41` and `origin/cx-41`) the same?
 - Answer: No, not necessarily.
- Question: Should we try to make them the same?
 - Answer: Possibly.

Here’s why the last two answers are vague. `cx-41` and `origin/cx-41` live in peace with each other, but the local and remote branches wander out of sync. We usually want to allow that, at least temporarily. As we experiment with new features in `cx-41`, there is some security in the knowledge that `origin/cx-41` is safe and undisturbed. We can restore from that point, easily (see section 10.3).

The version that exists on the remote `origin` can also wander. If we go to another computer, checkout `cx-41`, and push the changes back, then obviously the remote server’s version will differ from what we have in this computer.

The fact that the remote `origin/cx-41` and the local `origin/cx-41` can diverge from each other, and that each can be different from `cx-41`, is a source of confusion. To the best of our knowledge, the following are correct:

1. If we run “`git fetch`”, then all remote branch updates will be retrieved. The content from the remote for this branch will be stored in `origin/cx-41`. That does not alter the local tracking branch, `cx-41`. Getting the changes from `origin/cx-41` into `cx-41` requires a separate merge step (see section 6.14).
2. The two step process is avoided if we run “`git pull`” on this computer. Git checks the server for updates in `origin/cx-41` and copies them into our `origin/cx-41` file collection, and then it merges those updates into the local branch `cx-41`. This is the sense in which “`git pull`” is equivalent to “`git fetch`” and “`git merge`”.

Right after the “`git pull`”, our copy of the remote, `origin/cx-41`, the local tracking branch `cx-41`, and the server’s copy of the same are all perfectly aligned. Also, when we push from our local branch to the server, the two will be perfectly aligned. But that only lasts until `cx-41` is edited.

Second, consider the remote branch

So far, the branch `origin/KH_Guide_34` seems to exist only (over there) on the server. Actually, it exists in our `.git` directory. We see it in “`git branch -a`”. The command “`git branch`” never actually talks to a server at all, it just inquires into the condition of our `.git` folder. The report from “`git branch`” is a snapshot of whatever was retrieved the last time that the repository was updated (either by a “`git pull`” or “`git fetch`”).

But the files in our `.git`’s records on `origin/KH_Guide_34` might as well be out of reach (over there) on the server because we cannot directly interact with `origin/KH_Guide_34`. To interact with those files, it is necessary to run `checkout`.

When we run a `checkout`, that remote branch “`origin/KH_Guide_34`” will be copied into a local tracking branch named `KH_Guide_34`:

```
$ git checkout KH_Guide_34
$ git branch -a
* KH_Guide_34
  cx-41
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/KH_Guide_34
  remotes/origin/cx-41
  remotes/origin/master
```

At that instant, the contents of `KH_Guide_34` and `origin/KH_Guide_34` are perfectly synchronized. The branch that existed before only “on” the server (via our `.git` directory) now has a separate life of its own working directory. As we edit the files, the contents of `KH_Guide_34` will gradually wander away from `origin/KH_Guide_34`. And the contents stored on the server, of course, will wander away of the snapshot we have saved as `origin/KH_Guide_34`.

How do the two become re-synchronized? We use `push` to migrate changes from `KH_Guide_34` to `origin/KH_Guide_34`. If we have “`KH_Guide_34`” checked out and we run

```
$ git push -u origin KH_Guide_34
```

then two effects occur. The changes are imposed on our local `.git`’s copy of `origin/KH_Guide_34` and the changes are also uploaded to the server’s copy of that branch.

If we go to another computer and clone the repository, the repository will show only one local branch, `master`, and it will show all of those branches that exist only (over there) on the remote server (and in our local `.git` folder’s memory of it).

```
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/KH_Guide_34
```

5

```
remotes/origin/cx-41
remotes/origin/master
```

6.9 git fetch(retrieve abranches)

Running “git fetch” updates your repository’s copies of the branches on the default remote `origin`. It does not alter files in your current working directory.

To update all local copies of remote branches, the ones that appear as `origin/branch` in the output of “git branch -avv” by running

```
$ git fetch
```

Git fetch does not alter files in any of your local tracking branches. It only updates the record keeping branches, our branches with names like `origin/branch`.

Git fetch can be selective. Retrieve the master branch as it exists on the remote origin:

```
$ git fetch origin master
```

Note the syntax difference. Our local copy of that is referred to as `origin/master`, like one long character string, while the branch that exists on the origin server is named `master`.

After fetch, then inspect!

```
$ git branch -avv
```

This shows all of the branches and their most recent commit. It is a quick way to see if you are up to date.

Output of `git status` draws our attention to the differences between our locally checked out copy of a branch and the one that was just fetched.

```
$ git status
On branch cx-41
Your branch and 'origin/cx-41' have diverged,
and have 1 and 1 different commit each, respectively.

Unmerged paths:
  (use "git add/rm <file>..." as appropriate to mark resolution)

    both modified:   00-README.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

6.9.1 fetch --prune (remove branches that no longer exist on remote)

A appears to exist in 3 places:

1. in the current working directory, it shows in “git branch” as `cx-41` .
2. in the `.git` folder’s records of that branch, it shows as `remotes/origin/cx-41` .
3. and on the server itself, which we never see directly, except by asking git to push or pull files to/from the server.

If a branch has been merged onto the `master` , and we do not need it any more. To keep the current list of active branches as small as possible, the branch should be deleted. The tricky part, however, is that there are 3 variants that must be deleted.

When your merge request is handled in the GitLab server, the project manager may delete your branch on the remote (GitLab offers this as option). The branch `cx-41` is thus deleted on the server. However, when you run “git branch -a”, it will still show that there are branches named `cx-41` and `origin/cx-41` . So there’s still some deletion to do.

Delete your local branch by running this:

```
$ git branch -d cx-41
```

Then the local tracking branch will be deleted, but the output from “git branch -a” will still show `origin/cx-41` . We can’t tell, actually, whether that branch still exists on the server.

If the project manager did not remove the branch on the server, it is possible for that to be handled in the command line. The manager can run a sequence like this to merge the `cx-41` branch with `master` and then it will delete the branch locally and remotely.

```
$ git checkout master
$ git merge cx-41
## Next: delete local copy of branch
$ git branch -d cx-41
5 ## Tell remote server to remove its copy
$ git push origin --delete cx-41
To gitlab.crmda.ku.edu:crmda/guides.git
- [deleted]          cx-41
```

If `origin/cx-41` was deleted on the remote, we usually don’t want to keep our local copy of it. We usually use the sledge hammer called “purge” to eliminate our local copies of all branches that no longer exist in the server.

```
$ git fetch --prune
```

or, the briefer version

```
$ git fetch -p
```

6.10 git stash

Git will often notice that you have unsaved edits and it will refuse to merge files until you either commit or stash your changes. We’ve discussed commit, but stashing the changes is an option.

This “puts them out of the way”. Do this if you are not sure you want to commit, but think your edits might be valuable. Start with

```
$ git stash
```

After that, the action you were trying to do, such as a pull or checkout, will be allowed. If you want to retrieve the stashed edits and merge them on top of the current directory, the command is “`git stash pop`”.

Here’s an example:

```
$ git stash
Saved working directory and index state WIP on master: fdb2e35
  Commit that was pushed first
HEAD is now at fdb2e35 Commit that was pushed first

5 $ git pull
Updating fdb2e35..927c88f
Fast-forward
31.git/31.git.lyx | 2 +-
1 file changes, 1 insertion(+), 1 deletion(-)

$ git stash pop
Auto-merging 31.git/31.git.lyx
On branch master
Your branch is up-to-date with 'origin/master'.
```

If the revisions apply cleanly, then all is well and the problem is solved. If the revisions do not apply, then you have some work to do. It is necessary to figure out what the changes were in your stashed files and then reapply them. We leave the details as an exercise for a future version of this guide.

6.11 `git diff` (compare versions)

We’d like to compare the current situation with the recently downloaded “`origin/cx-41`” archive. A comprehensive report of all changes between our branch `master` and the version we just fetched with this command:

```
$ git diff origin/master master
```

To see all changes between a working directory in branch “`cx-41`” and the remote “`origin/cx-41`”, replace “`master`” with the name of the branch:

```
$ git diff origin/branch_name local_branch_name
```

The difficult part to understand is that your system never directly compares against the remote server.

6.11.1 Is my current working directory up to date with the remote repository?

Because there is no way to directly compare your file against the remote, this question needs to be understood differently. We need to ask git to update its records by downloading a fresh copy of the branches from the server by fetching

```
$ git fetch
```

And after that, tools like `diff` and `branch` can compare the working directory (the checked out branch) to the records that exist in the `.git` folder that reflect the most recent snapshot from the server. We might run, for example,

```
$ git status
$ git branch -avv
$ git diff origin/branch_name branch_name
```

One way to view commits that are on the local master branch that are not yet on the remote master branch (`origin/master`), run:

```
$ git log origin/master..master
commit ab5a11fd6c3e450cfbb95dd64456ae92be4bd5a7
Author: Paul E. Johnson <pauljohn@ku.edu>
Date: Thu Jul 9 16:02:08 2015 -0500

    test
```

An empty return indicates that the remote and local branches are up-to-date (the most recent remote commit is present on the local branch).

6.11.2 Experiment with gitk

Git uses the Unix `diff` command to compare files. The output from `diff` may be difficult to understand.

The Git distribution on all major platforms now includes a graphical tool named “`gitk`”. This can be used to view the branches and compare files. Run:

```
$ gitk <<file name>>
```

A new window will appear displaying information on changes between present files and previous commits, as well as additional information, such as a diagram of the commits made in the repository.

Running

```
$ gitk
```

launches a more general overview of the project history and its branches.

6.12 git mv (move, rename)

This works for files and directories. To change a file's name from “x” to “y”, run “`git mv x y`”. A file can be moved into a directory named “z” by “`git mv x z/y`”. It is allowed to move directories by renaming them. For example:

```
$ git mv basic.txt anewfilename.txt
```

After the `git mv` command, it is also necessary to run a commit. One is tempted to run either `git commit basic.txt` or `git commit anewfilename.txt`, but as it turns out, neither seems to capture the renaming effect of this command. Rather, we find this case is especially suitable for the “-a” flag.

```
$ git commit -a
```

after which the editor opens saying:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#   renamed:    basic.txt -> anewfilename.txt
```

6.13 git rm (remove files)

It is easy, perhaps too easy, to delete files from a Git project. This is one of the differences between Git and other version managers. If the author deletes a file—by whatever means—then the next run of “`git commit -a`” will notice the deleted files and it will remove those files from the tracking system. This does not delete the file from history, it can be retrieved if needed.

Those deletions may be accidental, however, and there is a way to more clearly keep records and remove files from a project.

Run “`git rm filename`”. A commit is also necessary.

```
$ git rm basic.txt
$ git commit -a
```

Caution: Directory deletion requires a flag

Git `rm` will refuse to delete a directory. Here we try to remove a directory named “notes”:

```
$ git rm notes
fatal: not removing 'notes' recursively without -r
```

Instead, one should insert the “-r” (“r”=recursive) flag, and all of the deleted files will be listed by name.

```
$ git rm -r notes
rm 'notes/README.md'
rm 'notes/RHS-notes-Ch-01.lyx'
rm 'notes/RHS-notes-Ch-01.pdf'
5 rm 'notes/RHS-notes-Ch-02.lyx'
rm 'notes/RHS-notes-Ch-02.pdf'
$ git commit
```

6.14 git merge

Go onto the target branch, the one into which we want the material to be inserted. Then apply the revisions from the source branch onto the target branch.

```
$ git checkout target
$ git merge source
```

That will apply edits from target to source. Frequently, there will be merge conflicts that must be manually resolved. See section 7.6.

If you use “git pull” to both retrieve remote branches and then bring the current working directory up to date, it will not be necessary to run a merge command. However, if you use “git fetch” to retrieve the remotes, then it is necessary to update the local tracking branch explicitly.

After git fetch, use merge to update a local tracking branch

Consider, for example, we want to work in our local branch `KH_Guide_34` but only after updating against the remote version, `origin/KH_Guide_34`

```
$ git fetch
$ git checkout KH_Guide_34
$ git merge origin/KH_Guide_34
```

This is equivalent to running “git pull” after checking out `KH_Guide_34`.

6.15 git show (recover previous versions)

This is easy to do in an editor like Emacs, where the menu `Tools -> Version Control -> Show Other Version` will allow the user to see an old version in a separate windows. That is discussed in section 8.2.

In the Git command line interface, it can be done with “git show”. Suppose the current directory has:

```
$ ls
test.xls
```


The commit history shows 5 commits:

```
$ git log --oneline
6bc9d2c Fix final paragraph in test.xls
c93cfe1 Found typo in second paragraph test.xls
4276f2e Inserted third paragraph test.xls
5647285 Created first 2 paragraphs in test.xls
6e043c7 Added test.xls
```

Create a file that is a copy of `test.xls` as it existed during the second commit:

```
$ git show 5647:test.xls > test-5647285.xls
```

The SHA1 of the desired commit is referenced (5647), the name of the file (`test.xls`), and the name of the new file to be created (`test-5647285.xls`).

The file list now shows

```
$ ls
test-5647285.xls  test.xls
```

6.16 git tag (labels for important commits)

A tag can be used as a handle for the current state of everything in a branch. Create a tag:

```
$ git tag -a YourTagNameHere -m 'YourTagMessageHere'
```

The “`-a`” switch is not required, but we recommend it. It creates an annotated tag. Annotated tags include creation date, the tagger name and email, and a tagging message.

View all tag names and the first 2 lines of the tag messages that went with them

```
$ git tag -n2
```

To see more information about a particular tag, use

```
$ git show YourTagNameHere
```

The most important purpose of a tag is to recover a project in a previous state. Suppose, for example, we know that there was a tag named “`version1`”, after which we have a lot of commits. We can check out the files as they were at that time by running:

```
$ git checkout version1
```

In Git, tags are not so vital as in other version managers. A tag is not actually needed to recover an old project. One can do the same by checking out the specific commit.

7 Troubleshooting Pull Fails, Push Fails

The problem usually starts when you have edited some files, committed them, and you want to push them back to a remote server. Git will refuse if your copy of the remote is not up-to-date (you were revising out of date files). The push fail leads off into a sequence of problems that we now explore.

7.1 Push Fail: out-of-date local repository

A user has copies of the same repository on 2 computers. She edits files on one and pushes the branch `sj-fix` to the server. Later, she edits the repository on a different computer. She has forgotten that she made edits on the other machine. When she tries to push the branch, the following error happens:

```
$ git push
To git@gitlab.crmda.ku.edu:software/kutils.git
! [rejected]          sj-fix -> sj-fix (fetch first)
error: failed to push some refs to
  'git@gitlab.crmda.ku.edu:software/kutils.git'
5 hint: Updates were rejected because the remote contains work that
hint: you do not have locally. This is usually caused by another
hint: repository pushing to the same ref. You may want to first
hint: integrate the remote changes (e.g., 'git pull ...')
hint: before pushing again.
```

The way to avoid this is to always stay up to date with the server. The Standard 4 Step sequence (Table 1) should be followed.

7.2 Pull Fail: A Fortunate Outcome May Occur

This happens if 2 people are working on the same branch. Or if one user has 2 computers with copies of the repository. Suppose an update for “`00-README.txt`” has been pushed to the remote from one repository. In another repository, it seems like `git pull` should “just work.”

Git is willing to pull, but it stops in the middle and asks for a commit message. The editor opens:

```
Merge branch 'sj-fix' of
  gitlab.crmda.ku.edu:crmdaprojects/Ticket-666-Client

# Please enter a commit message to explain why this merge is
  necessary,
# especially if it merges an updated upstream into a topic branch.
5 #
# Lines starting with '#' will be ignored, and an empty message
  aborts
# the commit.
```

The bothersome part is that `git pull` does not tell us which files are to be merged. In exasperation, we enter a commit message like “I have no idea what these changes are”².

When the pull merge succeeds, there will be a message explaining which files were altered. Generally, the resulting message is harmless, indicating that the file was updated.

²This is why some people urge us to never use `git pull`, but rather a two step `fetch/merge` process. That brings with it other complications that we do not want to impose on most of our users.

```

$ git pull
X11 forwarding request failed on channel 0
Merge made by the 'recursive' strategy.
 R/00-README.txt | 84 ++

```

A pull merge might be dangerous

Sometimes we find out the pull brings in a lot of files that we do not want. If another user made a lot of changes on the branch, look what we get.

```

$ git pull
X11 forwarding request failed on channel 0
Removing R/correctOutcomes.R
Merge made by the 'recursive' strategy.
5  R/00-README.txt | 84 ++
   R/accdbImport.R | 27 +
   R/{ => archive}/airDataMerge.R | 0
   R/{ => archive}/codeChecking.R | 0
   R/archive/correctOutcomes.R | 112 +++
10  R/archive/datadictionary.csv | 1 +
   R/archive/fullDataDictionary.csv | 488 ++++++++
   R/archive/fullDataDictionary_handEdit.csv | 1 +
   R/{ => archive}/import-2.R | 0
   R/{ => archive}/import.r | 0
15  R/{ => archive}/keyMaker.R | 0
   R/{ => archive}/mergeCheck.R | 0
   R/archive/saveRdsAsDta.R | 106 +++
   R/{ => archive}/zipCheck.R | 0
   R/correctOutcomes.R | 170 ----
20  R/import_2009-2015.R | 233 ++++++
   R/mergeAllHealth.R | 174 +++++
   R/metData.R | 113 +++
   R/newAir.R | 233 ++++++
   data/key.xlsx | Bin 6755 -> 10660
     bytes
25  writeup/FreqTable_20160426.lyx | 408 ++++++++
   writeup/finalReport.lyx | 1201
     ++++++++
   writeup/freqTable.lyx | 392 ++++++++
   writeup/table2.tex | 32 +
   24 files changed, 3605 insertions(+), 170 deletions(-)
30  create mode 100644 R/00-README.txt
   create mode 100644 R/accdbImport.R
   rename R/{ => archive}/airDataMerge.R (100%)
   rename R/{ => archive}/codeChecking.R (100%)

```

```

35 create mode 100644 R/archive/correctOutcomes.R
create mode 100644 R/archive/datadictionary.csv
create mode 100644 R/archive/fullDataDictionary.csv
create mode 100644 R/archive/fullDataDictionary_handEdit.csv
rename R/{ => archive}/import-2.R (100%)
rename R/{ => archive}/import.r (100%)
40 rename R/{ => archive}/keyMaker.R (100%)
rename R/{ => archive}/mergeCheck.R (100%)
create mode 100644 R/archive/saveRdsAsDta.R
rename R/{ => archive}/zipCheck.R (100%)
delete mode 100644 R/correctOutcomes.R
45 create mode 100644 R/import_2009-2015.R
create mode 100644 R/mergeAllHealth.R
create mode 100644 R/metData.R
create mode 100644 R/newAir.R
create mode 100644 writeup/FreqTable_20160426.lyx
50 create mode 100644 writeup/finalReport.lyx
create mode 100644 writeup/freqTable.lyx
create mode 100644 writeup/table2.tex

```

We accepted a lot of changes that we did not want or expect.

The most cautious Git users, the inhabitants of stackoverflow.com, will often recommend that we protect ourselves. Git users should avoid `pull`, and use a conservative two-step strategy, “`git fetch`” and “`git merge`”. See, for example, blog posts like “[GIT: Fetch and Merge, Don’t Pull](#)” If one has the patience, the changes that will result can be inspected before they are applied.

On the other hand, we may want the remote changes to come into our local folder. However, we want the remote changes to stay chronologically separate from our local revisions. By adding the `--rebase` argument, the project history may stay clean. Run

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

That does the following:

1. Separate your revisions of the file set and put them aside.
2. Fetch and apply remote changes to bring the local branch into line with the remote.
3. Your changes are *layered* back onto the up-to-date file set.

The files that result may be similar to a git merge, but the history will be different. The history of the commits will make it look as though your branch was up to date before you started editing. Git does not usually ask for a commit message when you do this, unless it detects some trouble re-merging your edits.

7.3 Git pull causes merge conflict

Git tracks file changes. It uses a format known as “diff” which create “patches” to represent changes from one version to another. When you pull from the remote, the diff may not apply cleanly. The merge does not fail, but it leaves some litter. Here is an example.

```
[pauljohn@login2 myGitPractice]$ git pull
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
5 From /crmda/users/pauljohn/myGitPractice
   ee9ee25..9e6cf8a  master      -> origin/master
Auto-merging 00-README.txt
CONFLICT (content): Merge conflict in 00-README.txt
Automatic merge failed; fix conflicts and then commit the result.
```

In this case, the file `00-README.txt` is conflicted. Below, in section 7.6, we have commentary about what to do when this happens.

7.4 Git pull refused: locally untracked files

The local directory may have files that are not tracked. If a file with the same name has been added to the repository from another computer, then “git pull” will fail. Git refuses to obliterate a locally untracked file by placing a tracked file on top of it. Look for the file name “accdbImport.R”.

```
$ git pull
X11 forwarding request failed on channel 0
remote: Counting objects: 280, done.
remote: Compressing objects: 100% (118/118), done.
5 remote: Total 280 (delta 190), reused 233 (delta 161)
Receiving objects: 100% (280/280), 69.25 KiB | 0 bytes/s, done.
Resolving deltas: 100% (190/190), done.
From gitlab.crmda.ku.edu:crmdaprojects/Ticket-673-Ahmed
   af6cff3..3852a11  master      -> origin/master
10 error: The following untracked working tree files would be
   overwritten by merge:
       R/accdbImport.R
Please move or remove them before you can merge.
Aborting
```

The obvious fix is to move `accdbImport.R` to another file name, then re-run `git pull`. Then figure out if the version of `accdbImport.R` that was pulled is better or worse than the one that you copied to another file name.

7.5 Git pull refused: uncommitted local revisions

In this case, `00-README.txt` is a file under Git tracking. We have edited it. A “git pull” failure results because our change was not yet committed.

```
$ git pull
X11 forwarding request failed on channel 0
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
5 remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From gitlab.crmda.ku.edu:crmdaprojects/Ticket-666-Client
  21392f1..38b0bb0  master      -> origin/master
Updating 21392f1..38b0bb0
10 error: Your local changes to the following files would be
    overwritten by merge:
        00-README.txt
Please, commit your changes or stash them before you can merge.
Aborting
```

That’s not as bad as it looks. *Git did nothing!* Git’s error message suggests either committing your edits or stashing them. Both of those will work.

When does this work? If local changes are not too radical and don’t contradict changes from the remote, then stashing the changes and re-applying them will probably succeed.

When can this cause trouble? If local changes do not apply to the pulled file set, then Git will enter a “rebase correction cycle.” Conflicting changes to the files must be corrected, as discussed in section 7.6.

7.6 Manually fixing merge conflicts

When the version manager tries to apply a set of changes to a file, and the changes do not fit together with the old file, then we have “rejected hunks” of edits. If there are rejected hunks, then the version tracker inserts markers. The conflicting sections are enclosed in brackets like <<< and >>>. This part of the file will look like this:

```
<<<<<<< This was entered during the first push
===== This was entered as your commit, and could not be pushed
    due to the merge conflict >>>>>>
```

A good deal of the Git framework is dedicated to avoiding those conflicted sections. But conflicts do happen.

When there are files with those conflict markers, they must be corrected. You will not be allowed to push changes to the remote server as long as git thinks files are still in conflict. It is necessary to

- Edit the files in which there are conflicts (make them correct). Editing these files can be frustrating. Between the <<< and >>>, the two conflicting committed edits are separated by =====.
- Run “git add filename” to let git know that you have corrected those problems.

- Then “git commit” and “git push”.

In a perfect world, that will go easily and the work is done. We recently had an example where it seemed not so easy, so here is a record of the incident. As you can see, when we ran `git pull`, the merge found a conflict and we were instructed to fix the file `variableKey.R`.

```
$ git pull
X11 forwarding request failed on channel 0
remote: Counting objects: 12, done.
remote: Compressing objects: 100% (12/12), done.
5 remote: Total 12 (delta 10), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
From gitlab.crmda.ku.edu:software/kutils
   4940bc0..2af20f6  master    -> origin/master
Auto-merging package/kutils/R/variableKey.R
10 CONFLICT (content): Merge conflict in package/kutils/R/variableKey.R
Automatic merge failed; fix conflicts and then commit the result.
[1]+  Done                  emacs variableKey.R
```

After editing the file (we removed `<<< >>>` and cleared up parts between), the effort to commit that file was rejected:

```
$ git add variableKey.R
$ git commit
```

Caution: Run “git commit”, not “git commit -a” or similar

In the above, one is supposed to run “git commit” with no options or arguments. If one makes the mistake of giving a file name, then some errors arise.

```
# Here is the mistake
$ git commit variableKey.R
fatal: cannot do a partial commit during a merge.
```

The status output looked like this:

```
$ git status .
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 2 different commits each, respectively.
5 (use "git pull" to merge the remote branch into yours)
All conflicts fixed but you are still merging.
 (use "git commit" to conclude merge)
Changes to be committed:
   modified:   variableKey.R
```

After checking about, we guessed the right solution was

```
$ git commit
```

In the editor window, this was the message:

```
Merge branch 'master' of gitlab.crmda.ku.edu:software/kutils
# Conflicts:
#     package/kutils/R/variableKey.R
#
5 # It looks like you may be committing a merge.
# If this is not correct, please remove the file
#     .git/MERGE_HEAD
# and try again.
# Please enter the commit message for your changes. Lines starting
10 # with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch and 'origin/master' have diverged,
# and have 1 and 2 different commits each, respectively.
# (use "git pull" to merge the remote branch into yours)
15 #
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#     modified:   variableKey.R
```

We added a commit message and exited. After that, all was well.

Complicated merge problems

Suppose the Git merge creates a file with a huge number of sections. There may be hundreds of conflicts and resolving them manually will take too much time.

In our experience, it is often necessary to take the low road and avoid the Git merge problem altogether. Terminate the merge (see section 10.4) and look at the substance of the problem.

1. Get a copy of the old version. This is the last “good one”.
2. Get a copy of new revised version.
3. Edit the 2 file side by side. The programming file editor Emacs has a `Tools -> Compare buffers` option that can make it easier to see where the changes are.

8 User Conveniences (Graphical Interfaces)

We have avoided graphical user interfaces because 1) they don't always get everything correct every time, and 2) users who rely on them experience “skill decay” so that they are incapable of fixing

things that go wrong. There are some commercial Git GUIs that we have not used, but we have tested several others.

We have success using addons for programming file editors like Emacs. We also have experimented with Tortoise for Windows, which is a file manager addon.

8.1 Windows Explorer Tortoise

Tortoise (<https://code.google.com/p/tortoisegit/>) offers a Windows File Explorer plugin for Git.

Caution: Danger of accidentally adding files.


The biggest danger with TortoiseGit is that users chronically add way too many files with it. If a folder has 20 files, but the user wants to track only 1, it is a very common error to add all 20 files. This is a user error, of course, but the error is encouraged by the GUI.

Again, Cautiously, we proceed

1. File icons in Windows Explorer

 A file that has been added to be tracked

 A tracked file has changed

 An tracked file that has not changed

No icon indicates the file is not being tracked by git.

2. Create a Repository

- Right-click the directory that has files you want to track with Git (do not open it).
- Select “Create Git repository here”.
- Do not choose “Make it Bare”. This is strictly for local tracking. If a remote server repo becomes available, we can reconfigure to push/pull from it.

3. Adding files to a repository. Open the directory in the file manager.

- Right click on a file, select “TortoiseGit”, then select “Add...”, OK, Commit, etc.
- Editor will open, asking for a commit message.
- We suggest that you also check the “Set commit date” and “Set author” boxes.
- If all goes well, the check mark  appears.

4. Reviewing the Git log.

- Right click in the directory
- Select “TortoiseGit”, then select “Show log”. This brings up the Log Messages window.

5. Retrieve an old version of one file.

- In the log, click on any entry that interests you.
- The file version for the entry will be apparent.
- To retrieve a previous version, right-click on the file and select “Save revision to...”

8.1.1 Tortoise interacts with branches

Tortoise is aware of branches. The File Manager only shows the files of the current branch, just as we expect. In the terminal, for example, suppose we check out a branch:

```
$ git checkout kk-baseball
Switched to branch 'kk-baseball'
```

Now the files from that branch are evident in Windows Explorer

Name	Date modified	Type
.git	7/10/2015 3:53 PM	File folder
Dugout.txt	7/10/2015 3:53 PM	TXT File
Home.txt	7/10/2015 3:53 PM	TXT File

8.2 Emacs

The text editor Emacs includes a Version Management interface that can interact with Git and Subversion. Look under the **Tools** menu, find **Version Control**. This is a general purpose editor tool that can interact with various version management systems. It is not Git specific, but it does understand Git.

Emacs has very convenient tools to commit edits (it can interact with the Emacs ChangeLog) and retrieve old versions of files.

In Emacs documentation, we refer to the Control key as “C” and the Meta key (usually the Alt key) as “M”. For example, “M-w” (Alt-w) means to copy the highlighted region and “C-y” means paste (Control-y).

8.2.1 Commit via Emacs

Tools -> **Version Control** -> **Check In/Out** .

This is the equivalent of “commit”. It will add a file to Git if it is not currently being tracked.

When you choose Check In/Out, Emacs throws open a buffer where you type your explanation of what you changed. Many people would paste in the last bit of the ChangeLog file for the project. Some people just make a note to remember what this checkin represents.

To get out of that buffer, use the key strokes “**C-c C-c**” (C for control). DO NOT close the buffer with menus, that will erase the checkin.

8.2.2 Retrieve an old file (interacting with history)

1. `Tools -> Version Control -> Show History`

Highlight and copy (M-w) the first 5 or 6 characters of that SHA1 for the commit that we want to retrieve.

2. `Tools -> Version Control -> Show Other Version`

The minibuffer (bottom of frame) asks for the SHA1. Paste that in (C-y). Hit enter. The previous version of the file will appear in the current directory.

8.2.3 Integrate with Emacs ChangeLog

Emacs has a built-in change log framework. To open a ChangeLog file, run “`M-x add-change-log-entry`” followed by the Return key. Emacs will ask what file will be used for the change log (Default: “ChangeLog” in current working directory). After inserting the message, then run “`C-c C-c`”. This records the entry and then puts the editor back in the file that was being edited. The next time Emacs is asked to commit changes (Menu `Tools -> Version Control -> Check In/Out`), Emacs will know that there was a ChangeLog entry. It will retrieve that message and propose it as the Git commit message.

9 We use Git-LFS for binary files

We have a GitLab server that is LFS enabled. This section is a place holder that will be created. The current notes on this are available in [Git LFS notes](#).

Part II

Advanced Knowledge and Operations

10 Getting Particular Things Done

This section outlines solutions to particular problems that we have encountered. When we find one good answer for something, we are recording it here. These suggestions generally assume an understanding of Git basics we have outlined in previous sections.

10.1 Recover lost files (or re-set accidentally edited files)

There is a nice discussion of this in [Restore a deleted folder in a Git repo](#)

10.1.1 Uncommitted revisions: restore individual files/folders

If you did not yet commit changes, do the following to undo your edits (to recover last committed version of the file), do this:

```
$ git checkout -- <file>
```

Note that this will erase the current version of the file. If there are edits in the file you want to save, copy it somewhere else.

If you have committed changes to your file, and want to look at the previous commits, then the checkout will not work. Instead, we suggest the method detailed in 6.15.

It is often asserted that one can `git checkout` to restore a directory, the deletion of which has occurred (say, in a file manager) but this was not yet committed:

```
$ git checkout -- path/to/folder
```

However in testing, we have not been able to confirm that!

10.1.2 Remove all uncommitted changes (not just particular files/folders)

Would somebody tell us the difference in these two approaches?

Our notes indicate that one can restore all files by using a period as the argument for `git checkout`.

```
$ git checkout .
```

The expert advice is to use this command to restore an entire project:

```
$ git reset --hard HEAD
```

Note that destroys all edits in files and it applies throughout the repository, not just in a directory. It is a sledge hammer, in other words. See below section 10.1.4 for comparison of reset with “hard” “mixed” and “soft” variants.

10.1.3 Committed revisions: retrieve one file that was removed

In the past, you deleted a file and committed that deletion. Now you want that file back. Because the file was deleted, it is not possible to open it in Emacs and ask for a previous version.

You don’t want the entire project restored to the old state. You just want one file.

There is good guidance in a Stack Overflow post [Find and Rstore a deleted file](#). We have tested that suggestion and it worked. The use case concerns our guide 43, the KU Thesis guide, in which we inadvertently deleted `KU-thesis-20170413.zip`. We did not realize that until several commits later.

Step 1. Find the commit that deleted the file, then get the name of the commit that was one before the deletion occurred:

```
$ git rev-list -n 1 HEAD -- KU-thesis-20170413.zip
d7659613779433e35d8b5b1a5e342f8593271eb3
```

Step 2. Retrieve the required file from that commit:

```
$ git checkout d7659^ -- KU-thesis-20170413.zip
```

After that, there is no return value (silence means success!) and the file is found in the working directory.

In this example, we used only the first 5 symbols in the commit name, but it is allowed (does work) to include the full long name:

```
$ git checkout d7659613779433e35d8b5b1a5e342f8593271eb3^ --
  KU-thesis-20170413.zip
```

10.1.4 git reset --hard to restore files in a project

The questions are “where do you want to restore from?” and “what have you done so far?”. If you did not make any commits yet, this will put files back the way they were.

```
$ git reset --hard HEAD
```

We need to fill in details later. .

A `--hard` reset will destroy (remove) edits.

A `--soft` reset will restore files, but attempt to retain edits.

The symbol `HEAD` indicates the state to which you intend to return (possibly with edits). It means the most recent commit. Note the difference between `HEAD` and `HEAD~1` here. The former simply undoes edits but it does not alter anything that was committed. On the other hand, the latter will “peel off” one commit from history. See next subsection.

10.1.5 git reset --soft to undo a commit, but leave files unchanged

If you have committed the changes, but decide you want to undo the commit, but leave your edits in the files, then do

```
$ git reset --soft HEAD~
```

Note the “`HEAD~`” means “the commit before the most recent one”. It means the same thing as “`HEAD~1`” and, so far as we can tell, “`HEAD^`” (depending on the OS or git version, we have seen all of these notations). This soft reset does not remove your edits, it just uncommits them, so you can continue to revise before committing.

Scenarios for this: your previous commit accidentally deleted some files that you want to restore. But it also has valuable edits that you do not want to re-type. This commit leaves your revisions, but restores deleted files and the logs of the project will not show any records on the accidental deletion and restoration.

10.2 Copy one file from a branch

If you need a file that exists on a different branch, but you don't want to merge the whole branch onto the current branch, the individual file can be retrieved. The syntax is:

```
$ git checkout SourceBranch -- filename1
```

This will replace `filename1` in the current directory, so make a safe copy of the file before doing this.

First, check out the target branch, which in this case is `master` :

```
$ git checkout master
Switched to branch 'master'
```

Bring the `Dugout.txt` file from the `kk-baseball` branch to the `master` branch.

```
$ git checkout kk-baseball -- Dugout.txt
```

10.3 Reverse accidental edits on the wrong branch.

A usual scenario is that you wanted to work on a branch named `pj-fix` but you accidentally edited on the `master` branch. You want to

1. Save your edits
2. Restore accidentally revised local master

The easy approach, which we describe here, assumes that there is a remote. *Assume you did not push yet.* But you pulled recently. This means `origin/master` is a *reasonable fallback position*. This approach is one of the many suggested answers in Stack Overflow, “[How to fix committing to the wrong Git branch?](#)”.

Recover from `origin/master`

The local copy of `origin/master` will be used to set the `master` branch where it ought to be. When we are done, a temporary branch `pj-temp` will have the edits from the current working directory.

```
$ git stash # skip if all changes are committed
$ git branch pj-temp
$ git reset --hard origin/master
$ git checkout pj-temp
```

We put our non-committed edits out of the way with “`git stash`” (see section 6.10). We put the `master` branch where it needed to be, but we shift into the file set of the branch we intended to edit from the beginning. The `reset` put the files back where they were to start with. We have a new branch `pj-temp`. We check that out, and we can begin editing again, but we might save some work by re-applying the stashed edits.

```
$ git stash pop # skip if all changes were committed
```

The history of the commits will make it look as though your branch was up to date before you started editing.

Master will be reset to the state it had at the last time it was pulled or pushed from the remote.

This creates a new branch `pj-temp`, of course. If you want to go back and put the work on `pj-fix`, the approach is obvious. Commit changes on `pj-temp`, then checkout `pj-fix`, and run “`git merge pj-temp`”.

If the aim is to restore a branch that is not being tracked remotely, then some “roll up your sleeves” effort is needed. Rather than trying to write out all of the possible answers, we’ll wait until these problems actually happen and keep good notes on the fixes, to add here later.

10.4 Undo Merge

Suppose you run “`git merge lemon`” and you instead wanted to run “`git merge orange`”. If the merge causes a lot of peculiar changes in your files, you want to reverse the merge and pretend it never happened.

The right answer depends on whether you notice this right away, or only after making some additional edits and commits, or if you have already pushed this to a remote. If you have already pushed an erroneous merge to the remote, the situation is very serious because other team members might already have updated against your error.

If the simple strategies we suggest here fail, it may be necessary to wade into some very long threads on Stack Overflow, such as “[Undo a Git merge that hasn’t been pushed yet](#)”.

If there are merge conflicts:

If Git tries to combine the files and fails, there will be a message that the conflicts need to be resolved before proceeding. This is the best case scenario, because the merge was not finished and so it is relatively easy to undo it. If your Git version is recent, try

```
$ git merge --abort
```

The syntax in previous versions of Git for same was “`git reset - -merge`”.

If there are no merge conflicts:

If there are no merge conflicts, then Git thinks the merge went well, but you still don’t want it.

```
$ git reset --merge HEAD^
```

This peels off one commit, which was your merge commit (`HEAD^` is same as identical to “`HEAD~1`”). It is possible to step back in time to any previous commit SHA1 value, “`git reset --merge SHA`”.

The “`--merge`” argument is a special adaption for this situation, it is often suggested to use the general purpose “`--hard`” to step back in time by throwing away commits.

```
$ git reset --hard HEAD^
```

If other files were edited after the merge:

Running “`git reset --hard commit_SHA`” will move the current project back to the `commit_SHA` that you designate. It will obliterate other edits that you have made since then in other files that are not linked to the merged changes.

The following is likely to preserve the edits that occurred since the merge (*hopefully*).

```
$ git reset --merge ORIG_HEAD
```

Here, “`ORIG_HEAD`” is a literal string, not a name of a SHA to find.

If you committed and pushed the merge:

This is a somewhat dire situation. If teammates have not pulled yet, then this seems to be the suggested answer:

```
$ git revert -m 1 commit_SHA
```

Here, “`commit_SHA`” is the SHA of the revision.

10.5 Re-sequence edits “on top” of master (rebase)

This has happened more than once.

1. User forgets to run “`git pull`”.
2. User makes branch from (old/outdated) master.
3. User later runs the standard four-step sequence (Table 1) to update a branch against master.

The commit histories will be blended chronologically. A review of the log may make the sequence seem incoherent. The author on the master branch has a message saying “deleted feature 123” and then right after that a branch message says “fixed feature 123”. The chaotic history from the merge is usually not a serious source of concern for our projects.

Fixing the content of the edits is our main concern and we need to rescue whatever fixes we can get from the branch and put them to use in the future. If the Git merge fails, then there maybe be 100s of conflict sections in the merge files. It may be next to impossible to figure out what should be done.

In all honesty, we might not use Git to fix this. In many of our real life research situations, we do not bother to go down in the Git weeds. We make a brand new branch from an up-to-date master, and then hand-edit files to integrate the changes that were worked out in the branch that we are

trying to recover. In terms of programmer time, the author of the code in the new branch is most likely to be able to tell which edits are useful and need to be integrated.

However, it may be easier to do this with Git if you understand some benefits of re-organizing the information. Here is a good plan when a git merge has conflicts and the edits are too complicated. First, abort the merge, as described in 10.4. Second, rebase the branch. The effect is as follows. Peel off all of the edits on the branch, put a fully up-to-date master branch in place, and then try to apply the revisions of the current branch. The premise behind this is that the master branch is always correct and a user's branch is a proposed improvement in a feature of the master branch. The author of the branch should update against master, then start re-applying the changes.

```
$ git checkout my_local_branch
$ git rebase master my_local_branch
```

The conflicted files must be *edited*, one by one, *manually*. Each one must be added. In this context, add means to notify git that the file is now corrected. This must be done for each and every conflicted file.

```
$ git add name_of_conflicted_file
```

Then finish integrating the project.

```
$ git rebase --continue
```

The benefit of doing it the Git way is that the project history—the output of “git log”—might be more meaningful. If one takes our simpler method of having a programmer put the changes into a new branch, we'll come out of it with one commit that summarizes the changes, but not much history.

10.6 Interactively merge revisions

The Git merge function described in section 6.14 is a bit blunt. It is an all-or-nothing decision. All of the changes are accepted. Or none. Git uses text merging algorithms to combine all of the edits from one branch. It may not guess correctly what should be merged.

There are more subtle methods. One is the interactive merge.

Suppose X.R exists on the soccer branch. Adding “-p” with git checkout offers an interactive patch-importer that will pull over the file X.R and merge it with the one in our current branch:

```
$ git checkout -p soccer -- X.R
```

It launches an interactive chooser. Git shows differences and asks

```
Apply this hunk to index and worktree [y,n,q,a,d,/,e,?]?
```

Entering “?” will return an explanation of the given options:

```
y - apply this hunk to index and worktree
n - do not apply this hunk to index and worktree
e - manually edit the current hunk
```

Entering “e” will open the editor for you to enter an message. Lines beginning with “-” will be deleted, and lines beginning with “+” will be added. In order to prevent lines from being added, delete the lines in question. In order to prevent lines from being deleted, delete the “-” sign. Example:

```
-Oh no! This line will be deleted! Let's save it!
```

becomes

```
Oh no! This line will be deleted! Let's save it!
```

which will now remain in the document after editing. If the editor is Vi, exit with the usual “Escape : wq”.

11 Concluding Advice

11.1 Don't Get Carried Away by Russian Teenagers

There are people who spend all of their time using Git and dealing with its nuances. If you dip your toes into Stack Overflow, you will find unbelievably diverse opinions about how to to even the most basic things.

For example, suppose you want to merge changes selectively from one branch into another. Even this simple objective can set off a firestorm of differing opinions in Stack Overflow:

- [How to merge specific files from Git branches](#)
- [How do you merge selective files with git-merge?](#)
- [How do I merge changes to a single file, rather than merging commits?](#)

The main dilemma is we don't know how long it will take to have the fancy solution or be sure it works correctly.

We are a research unit and we do not want researchers to spend more time with version management than necessary. This guide is not a comprehensive of everything possible in Git. It is a listing of the things that we are pretty sure will work.

11.2 Let us Know What You Find Out (We'll Let You Know What We Find Out)

From time-to-time, we have problems and, in those cases, somebody has to

1. dig deep in the details and figure out the simplest, least error-prone-method.
2. write out the answer for insertion in the previous section.

We only write out answers for problems we have actually encountered.

If you think you learn something, send us a note. We'll try to verify and insert it.

11.3 Things to Not Do

We don't have a comprehensive list. But here's a start.

- Avoid running commands you don't fully understand.
- Avoid running that have the word "force" in them, even if you think you do understand them.
- Avoid doing things that will confuse teammates, such as rebasing a branch after pushing it to a server.

11.4 Things to Do

1. Make small test cases.

When in doubt, follow the instructions for Scenario #2 (in section 5.3). Make a small example of the problem you think you have. Then copy that folder entirely, even zip it so you can be sure you can recover it. Then try the changes on the minimal example project. If your fix fails, delete the copy and unzip the original.

If you don't find a solution, ask in the Git section of [Stack Overflow](#). If you can give commands to create your troublesome small example, people will be more than happy to help. They won't help if you just ask a vague question about something that does not work.

It is possible to even make a fake remote server that is a bare repository you create in a folder on your computer. But we aren't going to write out how to do that.

2. Keep verbatim records of what you are doing in the command line. If you hit trouble, copy/paste as much of the history as you can into a flat text file. Don't bother taking a picture of the screen, that's what novices do.

While you are still a beginner, find a way to save transcripts of sessions when you are typing Git commands. We don't know for sure if there is a perfect, universal way to accomplish this. We are eager to hear from users who have advice. We have recommended Emacs shell (or eshell) for this at some points, but users on Macintosh systems report some peculiarities.

3. Try to keep your Git repository clean. Delete branches after work on them is finished and merged into the master branch.