What is version control?

A **version control system** is a tool that manages changes made to the files and directories in a project. Many version control systems exist; this lesson focuses on one called Git, which is used by many of the data science tools covered in our other lessons. Its strengths are:

- Nothing that is saved to Git is ever lost, so you can always go back to see which results were generated by which versions of your programs.
- Git automatically notifies you when your work conflicts with someone else's, so it's harder (but not impossible) to accidentally overwrite work.
- Git can synchronize work done by different people on different machines, so it scales as your team does.

Version control isn't just for software: books, papers, parameter sets, and anything that changes over time or needs to be shared can and should be stored and shared using something like Git.

Which of the following does Git do?

Answer the question

- Keep track of changes to files.
- Notice conflicts between changes made by different people.
- Synchronize files between different computers.
- All of the above.

Where does Git store information?

Each of your Git projects has two parts: the files and directories that you create and edit directly, and the extra information that Git records about the project's history. The combination of these two things is called a **repository**.

Git stores all of its extra information in a directory called `.git` located in the root directory of the repository. Git expects this information to be laid out in a very precise way, so you should never edit or delete anything in `.git`.

Suppose your home directory `/home/repl` contains a repository called `dental`, which has a sub-directory called `data`. Where is information about the history of the files in `/home/repl/dental/data` stored?

Possible Answers

- `/home/repl/.git`
- `/home/repl/dental/.git`
- `/home/repl/dental/data/.git`
- None of the above.

How can I check the state of a repository?

When you are using Git, you will frequently want to check the **status** of your repository. To do this, run the command `git status`, which displays a list of the files that have been modified since the last time changes were saved.

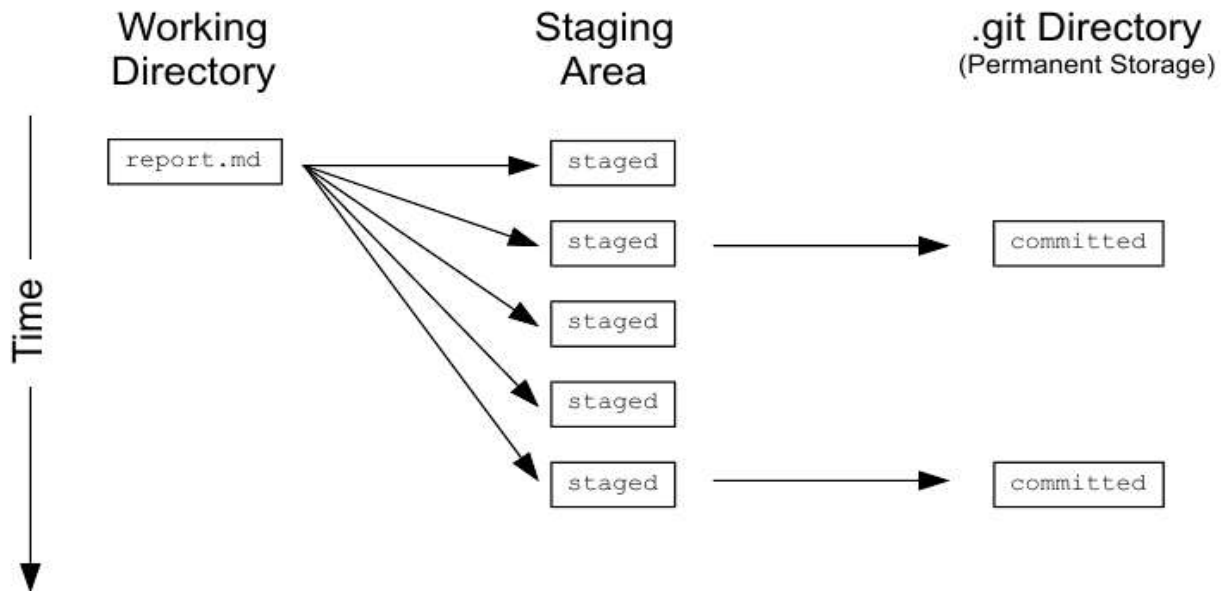
You have been put in the `dental` repository. Use `git status` to discover which file(s) have been changed since the last save. Which file(s) are listed?

Possible Answers

- `data/summer.csv`.
- `report.txt`.
- Neither of the above.
- Both of the above.

How can I tell what I have changed?

Git has a **staging area** in which it stores files with changes you want to save that haven't been saved yet. Putting files in the staging area is like putting things in a box, while **committing** those changes is like putting that box in the mail: you can add more things to the box or take things out as often as you want, but once you put it in the mail, you can't make further changes.



`git status` shows you which files are in this staging area, and which files have changes that haven't yet been put there. In order to compare the file as it currently is to what you last saved, you can use `git diff filename`. `git diff` without any filenames will show you all the changes in your repository, while `git diff directory` will show you the changes to the files in some directory.

You have been put in the `dental` repository. Use `git diff` to see what changes have been made to the files.

What is in a diff?

A **diff** is a formatted display of the differences between two sets of files. Git displays diffs like this:

```
diff --git a/report.txt b/report.txt
index e713b17..4c0742a 100644
--- a/report.txt
+++ b/report.txt
@@ -1,4 +1,5 @@
-# Seasonal Dental Surgeries 2017-18
+# Seasonal Dental Surgeries (2017) 2017-18
+# TODO: write new summary
```

This shows:

- The command used to produce the output (in this case, `diff --git`). In it, `a` and `b` are placeholders meaning "the first version" and "the second version".
- An index line showing keys into Git's internal database of changes. We will explore these in the next chapter.
- `--- a/report.txt` and `+++ b/report.txt`, wherein lines being *removed* are prefixed with `-` and lines being added are prefixed with `+`.
- A line starting with `@@` that tells *where* the changes are being made. The pairs of numbers are `start line` and `number of lines` (in that section of the file where changes occurred). This diff output indicates changes starting at line 1, with 5 lines where there were once 4.
- A line-by-line listing of the changes with `-` showing deletions and `+` showing additions (we have also configured Git to show deletions in red and additions in green). Lines that *haven't* changed are sometimes shown before and after the ones that have in order to give context; when they appear, they *don't* have either `+` or `-` in front of them.

Desktop programming tools like **RStudio** can turn diffs like this into a more readable side-by-side display of changes; you can also use standalone tools like **DiffMerge** or **WinMerge**.

You have been put in the `dental` repository. Use `git diff data/northern.csv` to look at the changes to that file. How many lines have been added or removed?

Possible Answers

- 

  None.

- 

  1.

- 

  2.

- 

  20.

What's the first step in saving changes?

You commit changes to a Git repository in two steps:

1. Add one or more files to the staging area.
2. Commit everything in the staging area.

To add a file to the staging area, use `git add filename`.

1.You have been put in the `dental` repository. Use `git add` to add the file `report.txt` to the staging area.

2.Use another Git command to check the repository's status.

How can I tell what's going to be committed?

To compare the state of your files with those in the staging area, you can use `git diff -r HEAD`. The `-r` flag means "compare to a particular revision", and `HEAD` is a shortcut meaning "the most recent commit".

You can restrict the results to a single file or directory using `git diff -r HEAD path/to/file`, where the path to the file is relative to where you are (for example, the path from the root directory of the repository).

We will explore other uses of `-r` and `HEAD` in the next chapter.

1.You have been put in the `dental` repository, where `data/northern.csv` has been added to the staging area. Use `git diff` with `-r` and an argument to see how files differ from the last saved revision.

2. Use a single Git command to view the changes in the file that has been staged (and *only* that file).

3. `data/eastern.csv` hasn't been added to the staging area yet. Use a Git command to do this now.

Interlude: how can I edit a file?

Unix has a bewildering variety of text editors. In this course, we will sometimes use a very simple one called Nano. If you type `nano filename`, it will open `filename` for editing (or create it if it doesn't already exist). You can then move around with the arrow keys, delete characters with the backspace key, and so on. You can also do a few other operations with

control-key combinations:

- Ctrl-K: delete a line.
- Ctrl-U: un-delete a line.
- Ctrl-O: save the file ('O' stands for 'output').
- Ctrl-X: exit the editor.

Run `nano names.txt` to edit a new file in your home directory and enter the following four lines:

```
Lovelace
Hopper
Johnson
Wilson
```

To save what you have written, type Ctrl-O to write the file out, then Enter to confirm the filename, then Ctrl-X and Enter to exit the editor.

How do I commit changes?

To save the changes in the staging area, you use the command `git commit`. It always saves everything that is in the staging area as one unit: as you will see later, when you want to undo changes to a project, you undo all of a commit or none of it.

When you commit changes, Git requires you to enter a **log message**. This serves the same purpose as a comment in a program: it tells the next person to examine the repository why you made a change.

By default, Git launches a text editor to let you write this message. To keep things simple, you can use `-m "some message in quotes"` on the command line to enter a single-line message like this:

```
git commit -m "Program appears to have become self-aware."
```

If you accidentally mistype a commit message, you can change it using the `--amend` flag.

```
git commit --amend - m "new message"
```

- 1

You have been put in the `dental` repository, and `report.txt` has been added to the staging area. Use a Git command to check the status of the repository.

- 2

Commit the changes in the staging area with the message "Adding a reference."

How can I view a repository's history?

The command `git log` is used to view the **log** of the project's history. Log entries are shown most recent first, and look like this:

```
commit 043070548738119599 3bac9c21512ccfb511056d
Author: Rep Loop <repl@datacamp.com>
Date:   Wed Sep 20 13:42:26 2017 +0000

    Added year to report title.
```

The `commit` line displays a unique ID for the commit called a **hash**; we will explore these further in the next chapter. The other lines tell you who made the change, when, and what log message they wrote for the change.

When you run `git log`, Git automatically uses a pager to show one screen of output at a time. Press the space bar to go down a page or the 'q' key to quit.

You are in the directory `dental`, which is a Git repository. Use a single Git command to view the repository's history. What is the message on the very first entry in the log (which is displayed last)?

Keep in mind that not all entries may be visible on the first screen, and that you might need to check additional pages to see the very first entry.

Instructions

Possible Answers

- "Added summary report file."
- "Added seasonal CSV data files"
- "Fixed bug and regenerated results."
- "Added reminder to cite funding sources."

How can I view a specific file's history?

A project's entire log can be overwhelming, so it's often useful to inspect only the changes to particular files or directories. You can do this using `git log path`, where `path` is the path to a specific file or directory. The log for a file shows changes made to that file; the log for a directory shows when files were added or deleted in that directory, rather than when the contents of the directory's files were changed.

You have been put in the `dental` repository. Use `git log` to display only the changes made to `data/southern.csv`. How many have there been?

Possible Answers

- 0.
- 1.
- 2.

3.

How do I write a better log message

Writing a one-line log message with `git commit -m "message"`is good enough for very small changes, but your collaborators (including your future self) will appreciate more

information. If you run `git commit` *without* `-m "message"`, Git launches a text editor with a template like this:

```
# 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 up-to-date with 'origin/master'.
#
# Changes to be committed:
#       modified:   skynet.R
#
```

The lines starting with `#` are comments, and won't be saved. (They are there to remind you what you are supposed to do and what files you have changed.) Your message should go at the top, and may be as long and as detailed as you want.

→You have been put in the `dental` repository, and `report.txt` has been added to the staging area. The changes to `report.txt` have already been staged. Use `git commit` *without* `-m` to commit the changes. The Nano editor will open up. Write a meaningful message and use Ctrl+O and Enter to save, and then Ctrl+X to leave the editor.

---------- CHAPTER2 ----------→

How does Git store information?

You may wonder what information is stored by each commit that you make. Git uses a three-level structure for this.

1. A **commit** contains metadata such as the author, the commit message, and the time the commit happened. In the diagram below, the most recent commit is at the bottom (`feed0098`), underneath its parent commits.
2. Each commit also has a **tree**, which tracks the names and locations in the repository when that commit happened. In the oldest (top) commit, there were two files tracked by the repository.
3. For each of the files listed in the tree, there is a **blob**. This contains a compressed snapshot of the contents of the file when the commit happened (blob is short for *binary large object*, which is a SQL database term for "may contain data of any kind"). In the middle commit, `report.md` and `draft.md` were changed, so the blobs are shown next to that commit. `data/northern.csv` didn't change in that commit, so the tree links to the blob

> from the previous commit. Reusing blobs between commits help make common operations fast and minimizes storage space.

https://assets.datacamp.com/production/repositories/1545/datasets/1bb404075fe1164d8b3fd78f4065b0bf3d86bc16/gds_2_1_SVG.svg

---

Looking at the diagram, which files changed in the most recent commit to this repository?

Possible Answers

- `data/northern.csv`
- `report.md`
- `draft.md`

What is a hash?

Every commit to a repository has a unique identifier called a **hash** (since it is generated by running the changes through a pseudo-random number generator called a **hash function**). This hash is normally written as a 40-character hexadecimal string like `7c35a3ce607a14953f070f0f83b5d74c2296ef93`, but most of the time, you only have to give Git the first 6 or 8 characters in order to identify the commit you mean.

Hashes are what enable Git to share data efficiently between repositories. If two files are the same, their hashes are guaranteed to be the same. Similarly, if two commits contain the same files and have the same ancestors, their hashes will be the same as well. Git can therefore tell what information needs to be saved where by comparing hashes rather than comparing entire files.

Use `cd` to go into the `dental` directory and then run `git log`. What are the first four characters of the hash of the most recent commit?

Possible Answers

- bedc
- 2e1b
- 2e95
- None of the above.

How can I view a specific commit?

To view the details of a specific commit, you use the command `git show` with the first few characters of the commit's hash. For example, the command `git show 0da2f7` produces this:

```
commit 0da2f7ad11664ca9ed933c1ccd1f3cd24d481e42
Author: Rep Loop <repl@datacamp.com>
Date:   Wed Sep 5 15:39:18 2018 +0000

    Added year to report title.

diff --git a/report.txt b/report.txt
index e713b17..4c0742a 100644
--- a/report.txt
+++ b/report.txt
@@ -1,4 +1,4 @@
-# Seasonal Dental Surgeries 2017-18
+# Seasonal Dental Surgeries (2017) 2017-18

 TODO: write executive summary.
```

The first part is the same as the log entry shown by `git log`. The second part shows the changes; as with `git diff`, lines that the change removed are prefixed with `-`, while lines that it added are prefixed with `+`.

You have been put in the `dental` directory. Use `git log` to see the hashes of recent commits, and then `git show` with the first few digits of a hash to look at the most recent commit. How many files did it change?

Reminder: press the space bar to page down through `git log`'s output and `q` to quit the paged display.

Possible Answers

- None.
- 1.
- 2.
- 4.

What is Git's equivalent of a relative path?

A hash is like an absolute path: it identifies a specific commit. Another way to identify a commit is to use the equivalent of a relative path. The special label `HEAD`, which we saw in the previous chapter, always refers to the most recent commit. The label `HEAD~1` then refers to the commit before it, while `HEAD~2` refers to the commit before that, and so on.

Note that the symbol between `HEAD` and the number is a tilde `~`, *not* a minus sign `-`, and that there cannot be spaces before or after the tilde.

You are in the `dental` repository. Using a single Git command, show the commit made just before the most recent one. Which of the following files did it change?

Possible Answers

- `report.txt`.
- `data/western.csv`.

- Both of the above.
- Neither of the above

How can I see who changed what in a file?

`git log` displays the overall history of a project or file, but Git can give even more information. The command `git annotate file` shows who made the last change to each line of a file and when. For example, the first three lines of output from `git annotate report.txt` look something like this:

```
04307054         (  Rep Loop     2017-09-20 13:42:26 +0000       1)#
Seasonal Dental Surgeries (2017) 2017-18
5e6f92b6         (  Rep Loop     2017-09-20 13:42:26 +0000       2)
5e6f92b6         (  Rep Loop     2017-09-20 13:42:26 +0000       3)TODO:
write executive summary.
```

Each line contains five elements, with elements two to four enclosed in parentheses. When inspecting the first line, we see:

1. The first eight digits of the hash, `04307054`.
2. The author, `Rep Loop`.
3. The time of the commit, `2017-09-20 13:42:26 +0000`.
4. The line number, `1`.
5. The contents of the line, `# Seasonal Dental Surgeries (2017) 2017-18`.

You are in the `dental` repository. Use a single command to see the changes to `report.txt`. How many different sets of changes have been made to this file (i.e., how many ***distinct*** hashes show up in the first column of the output)?

Possible Answers

- 1.
- 3.

- 4.
- 7.

How can I see what changed between two commits?

`git show` with a commit ID shows the changes made *in* a particular commit. To see the changes *between* two commits, you can use `git diff ID1..ID2`,
where `ID1` and `ID2` identify the two commits you're interested in, and the connector `..` is a pair of dots. For example, `git diff abc123..def456` shows the differences between the commits `abc123` and `def456`, while `git diff HEAD~1..HEAD~3` shows the differences between the state of the repository one commit in the past and its state three commits in the past.

You are in the `dental` repository. Use `git diff` to view the differences between its current state and its state two commits previously. Which of the following files have changed?

Possible Answers

- `data/western.csv`.
- `report.txt`.
- `data/southern.csv`.
- `report.txt` and `data/western.csv`.

-

`report.txt` and `data/southern.csv`.

How do I add new files?

Git does not track files by default. Instead, it waits until you have used `git add` at least once before it starts paying attention to a file.

In the diagram you saw at the start of the chapter, the untracked files won't have a blob, and won't be listed in a tree.

The untracked files won't benefit from version control, so to make sure you don't miss anything, `git status` will always tell you about files that are in your repository but aren't (yet) being tracked.

1.You are in the `dental` repository. Use `git status` to find the files that aren't yet being tracked.

2.Use `git add` to add the new file to the staging area.

3. Use `git commit` to save the staged files with the message "Starting to track data sources."

How do I tell Git to ignore certain files?

Data analysis often produces temporary or intermediate files that you don't want to save. You can tell it to stop paying attention to files you don't care about by creating a file in the root directory of your repository called `.gitignore` and storing a list of **wildcard** patterns that specify the files you don't want Git to pay attention to. For example, if `.gitignore` contains:

```
build
*.mpl
```

then Git will ignore any file or directory called `build` (and, if it's a directory, anything in it), as well as any file whose name ends in `.mpl`.

Which of the following files would *not* be ignored by a `.gitignore` that contained the lines:

```
pdf
*.pyc
```

```
backup
```

Possible Answers

- `report.pdf`
- `bin/analyze.pyc`
- `backup/northern.csv`
- None of the above.

How can I remove unwanted files?

Git can help you clean up files that you have told it you don't want. The command `git clean -n` will show you a list of files that are in the repository, but whose history Git is not currently tracking. A similar command `git clean -f` will then delete those files.

Use this command carefully: `git clean` only works on untracked files, so by definition, their history has not been saved. If you delete them with `git clean -f`, they're gone for good.

1.You are in the `dental` repository. Use `git status` to see the status of your repo.

2. `backup.log` appears to be an untracked file and it's one we don't need. Let's get rid of it. Use `git clean` with the appropriate flag to remove unwanted files.

3. Use `ls` to list the files in your current working directory. `backup.log` should no longer be there!

How can I see how Git is configured?

Like most complex pieces of software, Git allows you to change its default settings. To see what the settings are, you can use the command `git config --list` with one of three additional options:

- `--system`: settings for every user on this computer.
- `--global`: settings for every one of your projects.

- `--local`: settings for one specific project.

Each level overrides the one above it, so **local settings** (per-project) take precedence over **global settings** (per-user), which in turn take precedence over **system settings** (for all users on the computer).

You are in the `dental` repository. How many local configuration values are set in for this repository?

Possible Answers

- None.
- 3.
- 4.
- 7.

How can I change my Git configuration?

Most of Git's settings should be left as they are. However, there are two you should set on every computer you use: your name and your email address. These are recorded in the log every time you commit a change, and are often used to identify the authors of a project's content in order to give credit (or assign blame, depending on the circumstances).

To change a configuration value for all of your projects on a particular computer, run the command:

```
git config --global setting value
```

Using this command, you specify the `setting` you want to change and the `value` you want to set. The settings that identify your name and email address are `user.name` and `user.email`, respectively.

Change the email address (`user.email`) configured for the current user for *all* projects to `rep.loop@datacamp.com`.

---------- CHAPTER3 ----------→

How can I commit changes selectively?

You don't have to put all of the changes you have made recently into the staging area at once. For example, suppose you are adding a feature to `analysis.R` and spot a bug in `cleanup.R`. After you have fixed it, you want to save your work. Since the changes to `cleanup.R` aren't directly related to the work you're doing in `analysis.R`, you should save your work in two separate commits.

The syntax for staging a single file is `git add path/to/file`.

If you make a mistake and accidentally stage a file you shouldn't have, you can unstage the additions with `git reset HEAD` and try again.

1.From the output of `git status` on the right, you'll see that two files were changed; `data/northern.csv` and `data/eastern.csv`. Stage only the changes made to `data/northern.csv`.

2. Commit those changes with the message "Adding data from northern region."

How do I re-stage files?

People often save their work every few minutes when they're using a desktop text editor. Similarly, it's common to use `git add` periodically to save the most recent changes to a file

to the staging area. This is particularly useful when the changes are experimental and you might want to undo them without cluttering up the repository's history.

1.You are in the `dental` repository. Use `git status` to check the status of the repository.

2. It appears that some changes to `data/northern.csv` have already been staged, but there are new changes that haven't been staged yet. Use `git add` to stage these latest changes to `data/northern.csv` again.

How can I undo changes to unstaged files?

Suppose you have made changes to a file, then decide you want to **undo** them. Your text editor may be able to do this, but a more reliable way is to let Git do the work. The command:

```
git checkout -- filename
```

will discard the changes that have not yet been staged. (The double dash `--` must be there to separate the `git checkout` command from the names of the file or files you want to recover.)

Use this command carefully: once you discard changes in this way, they are gone forever.

Instructions

You are in the `dental` repository, where all changes to `.csv` files in `data` were staged. `git status` shows that `data/northern.csv` was changed again after it was staged. Use a Git command to undo the changes to the file `data/northern.csv`.

How can I undo changes to staged files?

At the start of this chapter you saw that `git reset` will unstage files that you previously staged using `git add`. By combining `git reset` with `git checkout`, you can undo changes to a file that you staged changes to. The syntax is as follows.

```
git reset HEAD path/to/file
git checkout -- path/to/file
```

(You may be wondering why there are two commands for re-setting changes. The answer is that unstaging a file and undoing changes are both special cases of more powerful Git operations that you have not yet seen.)

Instructions 1/2

1.Use `git reset` to unstage the file `data/northern.csv` (and *only* that file).

2. Use `git checkout --` to undo the changes since the last commit for `data/northern.csv`.

How do I restore an old version of a file?

You previously saw how to use `git checkout` to undo the changes that you made since the last commit. This command can also be used to go back even further into a file's history and restore versions of that file from a commit. In this way, you can think of committing as saving your work, and **checking out** as loading that saved version.

The syntax for restoring an old version takes two arguments: the hash that identifies the version you want to restore, and the name of the file.

For example, if `git log` shows this:

```
commit ab8883e8a6bfa873d44616a0f356125dbaccd9ea
Author: Author: Rep Loop <repl@datacamp.com>
Date:   Thu Oct 19 09:37:48 2017 -0400

    Adding graph to show latest quarterly results.

commit 2242bd761bbeafb9fc82e33aa5dad966adfe5409
Author: Author: Rep Loop <repl@datacamp.com>
Date:   Thu Oct 16 09:17:37 2017 -0400

    Modifying the bibliography format.
```

then `git checkout 2242bd report.txt` would replace the current version of `report.txt` with the version that was committed on October 16. Notice that this is the same syntax that you used to undo the unstaged changes, except `--` has been replaced by a hash.

Restoring a file doesn't erase any of the repository's history. Instead, the act of restoring the file is saved as another commit, because you might later want to undo your undoing.

One more thing: there's another feature of `git log` that will come in handy here. Passing `-` followed by a number restricts the output to that many commits. For example, `git log -3 report.txt` shows you the last three commits involving `report.txt`.

Instructions 1/4

1.The current contents of `data/western.csv` are shown in the terminal. Use `git log -2` to list the last two changes to that file.

2. Use `git checkout` with the first few characters of a hash to restore the version of `data/western.csv` that has the commit message `"Adding fresh data for southern and western regions."`.

3. Use `cat data/western.csv` to display the updated contents.

4. Commit the restored version of `data/western.csv`, and be sure to include a message.

How can I undo all of the changes I have made?

So far, you have seen how to undo changes to a single file at a time using `git reset HEAD path/to/file`. You will sometimes want to undo changes to many files.

One way to do this is to give `git reset` a directory. For example, `git reset HEAD data` will unstage any files from the `data` directory. Even better, if you don't provide any files or directories, it will unstage everything. Even even better, `HEAD` is the default commit to unstage, so you can simply write `git reset` to unstage everything.

Similarly `git checkout -- data` will then restore the files in the `data` directory to their previous state. You can't leave the file argument completely blank, but recall from **Introduction to Shell for Data Science** that you can refer to the current directory as `.`. So `git checkout -- .` will revert all files in the current directory.

1.Use `git reset` to remove all files from the staging area.

2. Use `git checkout` to put those files back in their previous state. Use the directory name `.` to mean "all of the files in or below this directory", and separate it from the command with `--`.

---------- CHAPTER4 ----------→

What is a branch?

If you don't use version control, a common workflow is to create different subdirectories to hold different versions of your project in different states, for example `development` and `final`. Of course, then you always end up with `final-updated` and `final-updated-revised` as well. The problem with this is that it becomes difficult to work out if you have the right version of each file in the right subdirectory, and you risk losing work.

One of the reasons Git is popular is its support for creating **branches**, which allows you to have multiple versions of your work, and lets you track each version systematically.

Each branch is like a parallel universe: changes you make in one branch do not affect other branches (until you **merge** them back together).

Note: Chapter 2 described the three-part data structure Git uses to record a repository's history: *blobs* for files, *trees* for the saved states of the repositories, and *commits* to record the changes. Branches are the reason Git needs both trees and commits: a commit will have two parents when branches are being merged.

In the diagram below, each box is a commit and the arrows point to the next ("child") commit. How many merges have taken place?

https://assets.datacamp.com/production/repositories/1545/datasets/836c41b57bbbd4d589a3d0a08e9befebd9807790/gds_4_1_diagram.svg

Possible Answers

- None
- 1
- 2
- 3

How can I see what branches my repository has?

By default, every Git repository has a branch called `master` (which is why you have been seeing that word in Git's output in previous lessons). To list all of the branches in a repository, you can run the command `git branch`. The branch you are currently in will be shown with a `*` beside its name.

You are in the `dental` repository. How many branches are in this repository (including `master`)?

Instructions

Possible Answers

- None.
- 1.

- 2.
- 3.

How can I view the differences between branches?

Branches and revisions are closely connected, and commands that work on the latter usually work on the former. For example, just as `git diff revision-1..revision-2` shows the difference between two versions of a repository, `git diff branch-1..branch-2` shows the difference between two branches.

You are in the `dental` repository. How many files in the `summary-statistics` branch are different from their equivalents in the `master` branch?

Possible Answers

- None.
- 1.
- 3.
- 8.

How can I switch from one branch to another?

You previously used `git checkout` with a commit hash to switch the repository state to that hash. You can also use `git checkout` with the name of a branch to switch to that branch.

Two notes:

1. When you run `git branch`, it puts a `*` beside the name of the branch you are currently in.
2. Git will only let you do this if all of your changes have been committed. You can get around this, but it is outside the scope of this course.

In this exercise, you'll also make use of `git rm`. This removes the file (just like the shell command `rm`) then stages the removal of that file with `git add`, all in one step.

1.You are in the `master` branch of the `dental` repository. Switch to the `summary-statistics` branch.

2. Use `git rm` to delete `report.txt`.

3. Commit your change with `-m "Removing report" as a message.

4. Use `ls` to check that it's gone.

5. Switch back to the `master` branch.

6. Use `ls` to ensure that `report.txt` is still there.

How can I create a branch?

You might expect that you would use `git branch` to create a branch, and indeed this is possible. However, the most common thing you want to do is to create a branch then switch to that branch.

In the previous exercise, you used `git checkout branch-name` to switch to a branch. To create a branch then switch to it in one step, you add a `-b` flag, calling `git checkout -b branch-name`,

The contents of the new branch are initially identical to the contents of the original. Once you start making changes, they only affect the new branch.

1. You are in the `master` branch of the `dental` repository. Create a new branch called `deleting-report`.

2. Use `git rm report.txt` to delete the report.

3. Commit your changes with a log message.

4. Use `git diff` with appropriate arguments to compare the `master` branch with the new state of the `deleting-report` branch.

How can I merge two branches?

Branching lets you create parallel universes; **merging** is how you bring them back together. When you merge one branch (call it the source) into another (call it the destination), Git incorporates the changes made to the source branch into the destination branch. If those changes don't overlap, the result is a new commit in the destination branch that includes everything from the source branch (the next exercises describe what happens if there *are* conflicts).

To merge two branches, you run `git merge source destination` (without `..` between the two branch names). Git automatically opens an editor so that you can write a log message for the merge; you can either keep its default message or fill in something more informative.

1.You are in the `master` branch of the `dental` repository. Merge the changes *from* the `summary-statistics` branch (the source) into the `master` branch (the destination) with the message "Merging summary statistics."

What are conflicts?

Sometimes the changes in two branches will conflict with each other: for example, bug fixes might touch the same lines of code, or analyses in two different branches may both append new (and different) records to a summary data file. In this case, Git relies on you to reconcile the conflicting changes.

The file `todo.txt` initially contains these two lines:

```
A) Write report.
B) Submit report.
```

You create a branch called `update` and modify the file to be:

```
A) Write report.
B) Submit final version.
C) Submit expenses.
```

You then switch back to the `master` branch and delete the first line, so that the file contains:

```
B) Submit report.
```

When you try to merge `update` and `master`, what conflicts does Git report? You can use `git diff master..update` to view the difference between the two branches.

-

  Just line B, since it is the only one to change in both branches.

-

  Lines A and B, since one was deleted and the other changed.

-

  Lines B and C, since one was changed and the other deleted.

-

  All three lines, since all were either added, deleted, or changed.

How can I merge two branches with conflicts?

When there is a conflict during a merge, Git tells you that there's a problem, and running `git status` after the merge reminds you which files have conflicts that you need to resolve by printing `both modified:` beside the files' names.

Inside the file, Git leaves markers that look like this to tell you where the conflicts occurred:

```
<<<<<<< destination-branch-name
...changes from the destination branch...
=======
...changes from the source branch...
>>>>>>> source-branch-name
```

In many cases, the destination branch name will be `HEAD` because you will be merging into the current branch. To resolve the conflict, edit the file to remove the markers and make whatever other changes are needed to reconcile the changes, then commit those changes.

1. You are in the `master` branch of the `dental` repository. Merge the changes *from* the `alter-report-title` branch (the source) into the `master` branch (the destination).

2. Use `git status` to see which file has conflicts.

3. It turns out that `report.txt` has some conflicts. Use `nano report.txt` to open it and remove some lines so that only the second title is kept. Save your work with Ctrl+O and Enter, and then leave the editor with Ctrl+X. You can easily remove entire lines with Ctrl+K.

4. Add the merged file to the staging area.

5. Commit your changes with a log message.

---------- CHAPTER5 ----------→

How can I create a brand new repository?

So far, you have been working with pre-existing repositories. If you want to create a repository for a new project in the current working directory, you can simply say `git init project-name`, where "project-name" is the name you want the new repository's root directory to have.

One thing you should *not* do is create one Git repository inside another. While Git does allow this, updating **nested repositories** becomes very complicated very quickly, since you need to tell Git which of the two `.git` directories the update is to be stored in. Very large projects occasionally need to do this, but most programmers and data analysts try to avoid getting into this situation.

1. Use a single command to create a new Git repository called `optical` in your current directory.

How can I turn an existing project into a Git repository?

Experienced Git users instinctively start new projects by creating repositories. If you are new to Git, though, or working with people who are, you will often want to convert existing projects into repositories. Doing so is simple, just run:

```
git init
```

in the project's root directory, or:

```
git init /path/to/project
```

from anywhere else on your computer.

1. You are in the directory `dental`, which is not yet a Git repository. Use a single command to convert it to a Git repository.

2. Check the status of your new repository.

How can I create a copy of an existing repository?

Sometimes you will join a project that is already running, inherit a project from someone else, or continue working on one of your own projects on a new machine. In each case, you will **clone** an existing repository instead of creating a new one. Cloning a repository does exactly what the name suggests: it creates a copy of an existing repository (including all of its history) in a new directory.

To clone a repository, use the command `git clone URL`, where `URL` identifies the repository you want to clone. This will normally be something like

```
https://github.com/datacamp/project.git
```

but for this lesson, we will use a repository on the local file system, so you can just use a path to that directory. When you clone a repository, Git uses the name of the existing repository as the name of the clone's root directory, for example:

```
git clone /existing/project
```

will create a new directory called `project` inside your home directory. If you want to call the clone something else, add the directory name you want to the command:

1. You have just inherited the dental data analysis project from a colleague, who tells you that all of their work is in a repository in `/home/thunk/repo`. Use a single command to clone this repository to create a new repository called `dental` inside your home directory.

How can I find out where a cloned repository originated?

When you a clone a repository, Git remembers where the original repository was. It does this by storing a **remote** in the new repository's configuration. A remote is like a browser bookmark with a name and a URL.

If you use an online git repository hosting service like GitHub or Bitbucket, a common task would be that you clone a repository from that site to work locally on your computer. Then the copy on the website is the remote.

If you are in a repository, you can list the names of its remotes using `git remote`.

If you want more information, you can use `git remote -v` (for "verbose"), which shows the remote's URLs. Note that "URLs" is plural: it's possible for a remote to have several URLs associated with it for different purposes, though in practice each remote is almost always paired with just one URL.

You are in the `dental` repository. How many remotes does it have?

Possible Answers

-

None.

-

1.

-

2.

How can I define remotes?

When you clone a repository, Git automatically creates a remote called `origin` that points to the original repository. You can add more remotes using:

```
git remote add remote-name URL
```

and remove existing ones using:

```
git remote rm remote-name
```

You can connect any two Git repositories this way, but in practice, you will almost always connect repositories that share some common ancestry.

You are in the `dental` repository. Add `/home/thunk/repo` as a remote called `thunk` to it.

How can I pull in changes from a remote repository?

Git keeps track of remote repositories so that you can **pull** changes from those repositories and **push** changes to them.

Recall that the remote repository is often a repository in an online hosting service like GitHub. A typical workflow is that you pull in your collaborators' work from the remote repository so you have the latest version of everything, do some work yourself, then push your work back to the remote so that your collaborators have access to it.

Pulling changes is straightforward: the command `git pull remote branch` gets everything in `branch` in the remote repository identified by `remote` and merges it into the current branch

of your local repository. For example, if you are in the `quarterly-report` branch of your local repository, the command:

```
git pull thunk latest-analysis
```

would get changes from `latest-analysis` branch in the repository associated with the remote called `thunk` and merge them into your `quarterly-report` branch.

1.the `master` branch of the repository `dental`. Pull the changes from the `master` branch of the remote repository called `origin`.

What happens if I try to pull when I have unsaved changes?

Just as Git stops you from switching branches when you have unsaved work, it also stops you from pulling in changes from a remote repository when doing so might overwrite things you have done locally. The fix is simple: either commit your local changes or revert them, and then try to pull again.

1. You are in the dental repository, which was cloned from a remote called `origin`. Use `git pull` to bring in changes from that repository.

2. Discard the changes in your repository.

3. Re-try the `git pull`.

How can I push my changes to a remote repository?

The complement of `git pull` is `git push`, which pushes the changes you have made locally into a remote repository. The most common way to use it is:

```
git push remote-name branch-name
```

which pushes the contents of your branch `branch-name` into a branch with the same name in the remote repository associated with `remote-name`. It's possible to use different branch names at your end and the remote's end, but doing this quickly becomes confusing: it's almost always better to use the same names for branches across repositories.

1.You are in the `master` branch of the `dental` repository, which has a remote called `origin`. You have changed `data/northern.csv`; add it to the staging area.

2. Commit your changes with the message "Added more northern data."

3. Push your changes to the remote repository `origin`, specifying the `master` branch.

What happens if my push conflicts with someone else's work?

Overwriting your own work by accident is bad; overwriting someone else's is worse.

To prevent this happening, Git does not allow you to push changes to a remote repository unless you have merged the contents of the remote repository into your own work.

In this exercise, you have made and committed changes to the `dental` repository locally and want to push your changes to a remote repository.

1. Use `git push` to push those changes to the remote repository `origin`, specifying the `master` branch.

2. In order to prevent you overwriting remote work, Git has refused to execute your push. Use `git pull` to bring your repository up to date with `origin`. It will open up an editor that you can exit with Ctrl+X.

3. Now that you have merged the remote repository's state into your local repository, try the push again.