Git

Course as on datacamp compiled by Asad Noman

(Compiler's note: The document you are about to read is compiled from transcript given in course Git for Data Science on Datacamp. You might therefore see some randomness in topics. However, it is assumed that you do know the basics of git.)

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.

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**.

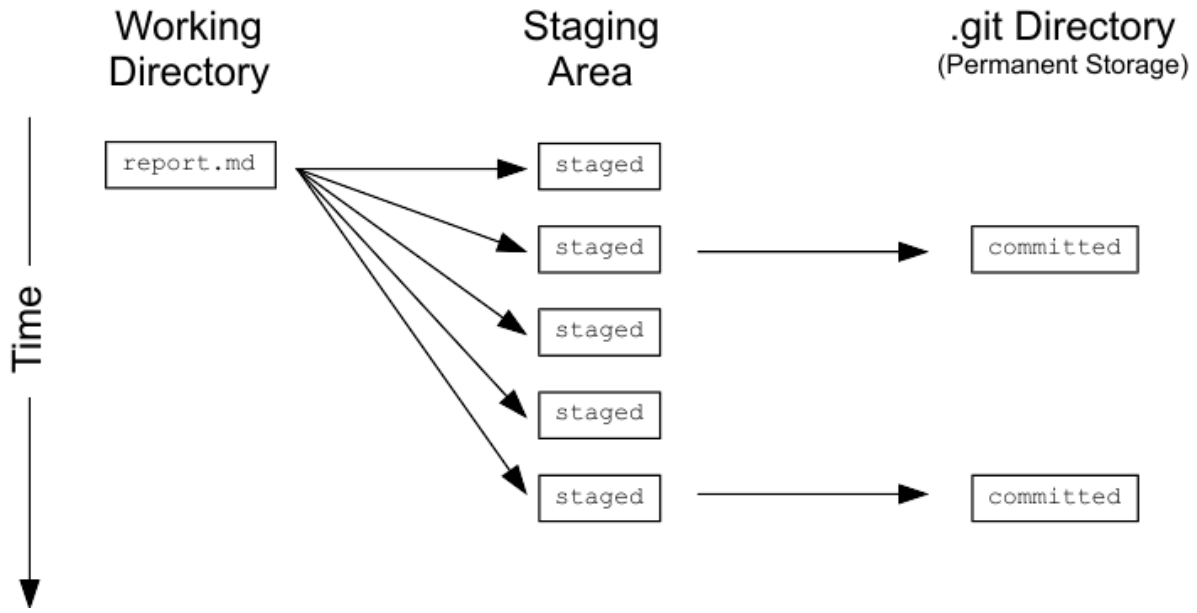
Git stores all of its extra information in a directory called `.git` located in the root directory of the repository. Git expects this information to be laid out in a very precise way, so you should never edit or delete anything in `.git`.

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.

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.

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 +1,4 @@
```

```
-# Seasonal Dental Surgeries 2017-18
+# Seasonal Dental Surgeries (2017) 2017-18

 TODO: write executive 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`, which indicate that lines being *removed* are prefixed with `-`, while lines being added are prefixed with `+`.
- A line starting with `@@` that tells where the changes are being made. Here, the line shows that lines 1-4 are being removed and replaced with new lines.
- 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**.

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`.

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

To compare a file's current state to the changes in the staging area, you can use `git diff -r HEAD path/to/file`. The `-r` flag means "compare to a particular revision", `HEAD` is a shortcut meaning "the most recent commit", and the path to the file is the 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.

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.

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."
```

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 0430705487381195993bac9c21512ccfb511056d
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.

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.

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:

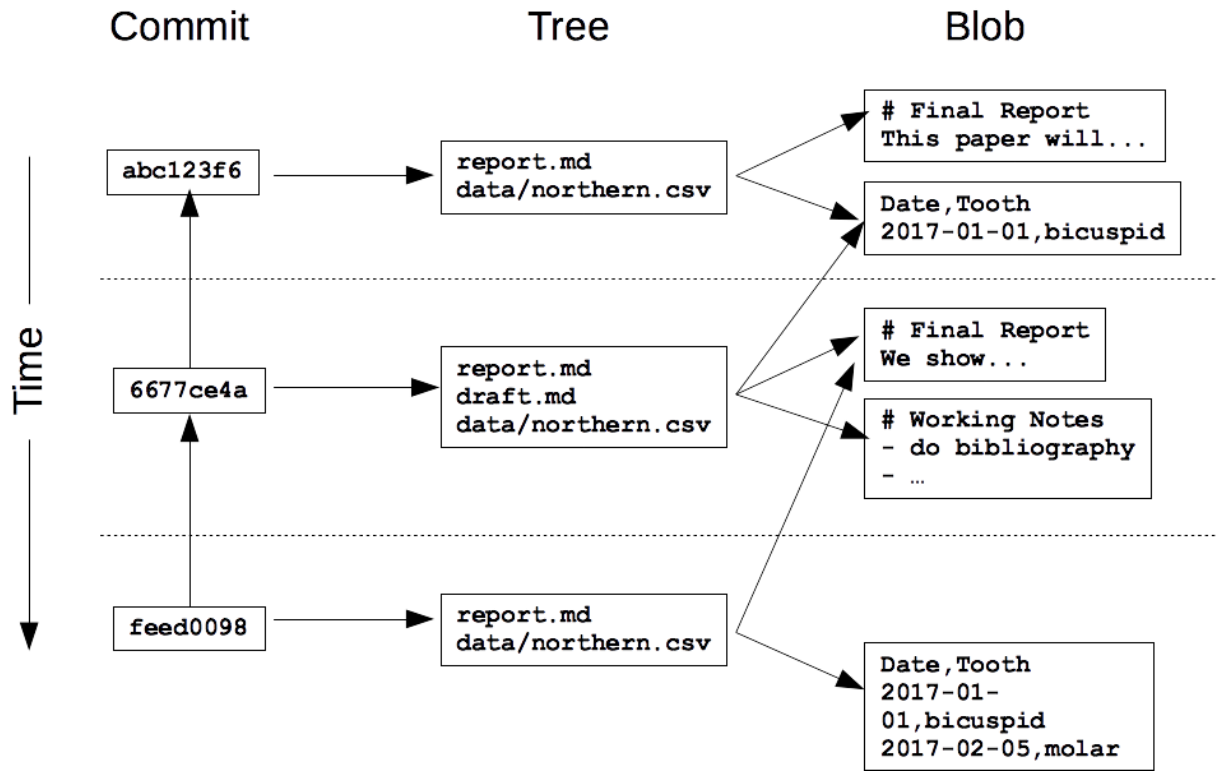
```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#       modified:   skynet.R
#
```

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.

How does Git store information?

In order to make common operations fast and minimize storage space, Git uses a multi-level structure to store data. In simplified form, this has three key parts:

1. Every unique version of every file. (Git calls these **blobs** because they can contain data of any kind.)
2. **tree** that tracks the names and locations of a set of files.
3. A **commit** that records the author, log message, and other properties of a particular commit.



As the diagram shows, each blob is stored only once, and blobs are (frequently) shared between trees. While it may seem redundant to have both trees and commits, a later part of this lesson will show why the two have to be distinct.

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.

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 043070` produces this:

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

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.

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.
```

The first column is the hash of the most recent commit to change that line. The other columns show who made the change, the date and time it was made, the line number, and the line itself.

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.

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. To remind you to do this, `git status` will always tell you about files that are in your repository but aren't (yet) being tracked.

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`.

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.

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).

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.name setting.value
```

with the setting's name and value in the appropriate places. The keys that identify your name and email address are `user.name` and `user.email` respectively.

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, 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.

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.

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.

How can I unstage a file that I have staged?

`git checkout -- filename` will undo changes that have not yet been staged. If you want to undo changes that *have* been staged, you can use `git reset HEAD filename`. This does *not* restore the file to the state it was in before you started making changes. Instead, it resets the file to the state you last staged. If you want to go all the way back to where you were before you started making changes, you must `git checkout -- filename` as well.

(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.)

How do I restore an old version of a file?

Since Git stores old versions of your files, you can use it to restore those files when you want to undo changes. The command for doing this is `git checkout`, which 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 `report.txt` with whatever was committed on October 16.

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.

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. You will sometimes want to undo changes to many files. One way to do this is to give `git reset` and `git checkout` a directory as an argument rather than the names of one or more files. For example, `git reset HEAD data` will unstage any files from the `data` directory that you have staged, and `git checkout -- data` will then restore those files to their previous state.

What is a branch?

One of the reasons Git is popular is its support for creating **branches**. A branch is like a parallel universe: changes you make in one branch do not affect other branches until you **merge** them back together. It's like creating sub-directories called `final`, `final-updated`, `final-updated-revised`, and so on, but with support for tracking work systematically.

Note: the first chapter 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.

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.

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.

How can I switch from one branch to another?

When you run `git branch`, it puts a `*` beside the name of the branch you are currently in. To switch to another branch, you use `git checkout branch-name`.

Note: 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.

How can I create a branch?

The easiest way to create a new branch is to run `git checkout -b branch-name`, which creates the branch and switches you to it. 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.

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 describes 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.

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.

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.

How can I create a brand new repository?

So far, you have been working with repositories that we created. If you want to create a repository for a new project, 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.

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.

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 **filesystem URLs** of the form `file:///existing/project`. The number of slashes at the start is important: the first part of the URL is `file://`, and then there is a third slash to start the absolute path `/existing/project`.

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 file:///existing/project` will create a new directory called `project`. If you want to call the clone something else, add the directory name you want to the command.

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 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.

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.

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. 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.

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.

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.

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.

(end)