Git
by George Boorman and datacamp

What is a version?
contents of a file at a given point in time
metadata associated with the file
-author
-location of file
-file type
-when it was last saved

version control is a group of systems and processes to manage changes made to documents, programs, and directories
useful for dynamic processes that need to be shared and changed over time
allows us to track files in different states and lets multiple people work on the same file simultaneously
this is called continuous development
allows us to combine different versions and revert changes

its like cooking without a recipe > it will be hard to reproduce the exact same results again
Git is not the same as GitHub
Git is a version control system
GitHub is a cloud-based Git repository hosting platform

Git stores everything
Git automatically notifies us when our work conflicts with someone else's > this makes it harder to accidentally overwrite content

Using Git
Git commands are run on the shell, also known as the 'terminal'
the shell is a program for executing commands
here we can run commands used in our version control workflow
such as previewing, modifying, and inspecting files or directories
*'directory' is often referred to on a computer as a folder

Useful shell commands
pwd - print working directory > to see our location
ls - what is in our current directory > returns a list of all files and directories
cd - change directory > we execute this command followed by the name of the directory to which we want to move into

example
cd archive > this would move us into the directory named ‘archive’

can also use the shell to preview and edit files
nano - opens a text editor > we execute this command followed by the filename that we want to modify
with nano we can delete, add, or change contents of a file
Ctrl + 0 will let us save within the text editor
Ctrl + X will let us return to the shell

can also use ‘echo’ command to create or edit a file
example - create a new file todo.txt
echo ‘Review for duplicate records’ > todo.txt
*if the file already exists, then we can append content by using two arrows instead of one
echo ‘Review for duplicate records’ >> todo.txt

**Nano is used for editing, and echo is better for printing messages to the terminal

Saving files
two parts to git project
first is the files and directories that we create and edit
the second is extra information that git records about the project’s history (recorded as .git)
*important not to edit or to delete these files
the combination of these two parts is called a ‘repository’ often referred to as ‘repo’

How to make changes to a repo?
we save a draft by placing it in a staging area
we save files and update the repo through a process called ‘commit’
analagy
staging area is like putting a letter in an unsealed envelope
making a commit is like putting a letter in a mailbox

.git files do not show when using the command ‘ls’
it is a hidden directory
we can pull it like this:
ls -a

Git workflow
-modify a file
-save the draft to the staging area

-commit the updated file to our repo
-repeat as needed

To add updated file to the staging area we use git add
example
git add report.md
add all files:
git add .
. = all files and directories in current location

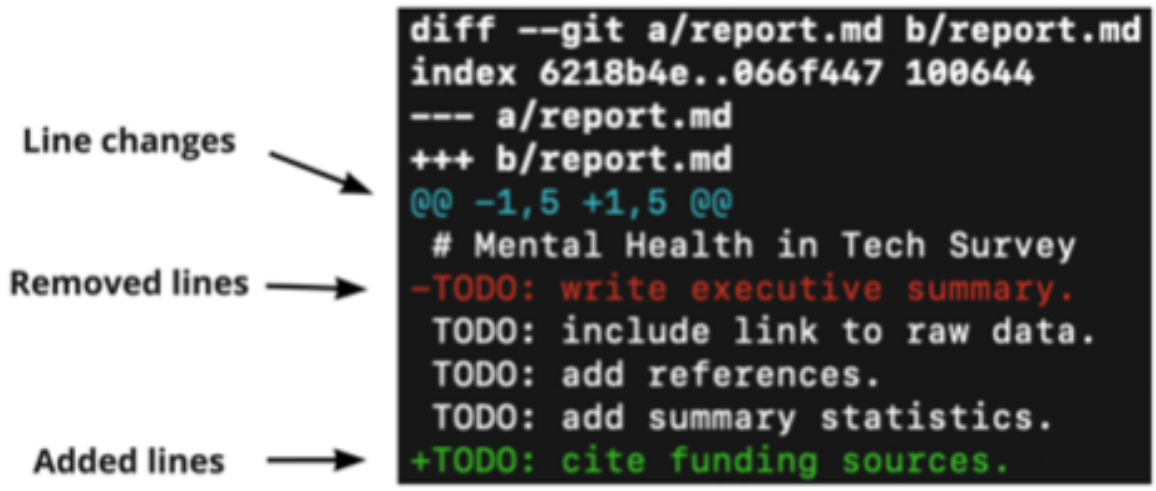
Making a commit
example
git commit -m ‘updating todo list in report.md’
#-m flag allows us to include a log message
best practice for log messages is short and sweet

How to check the status of files
example
git status
#this will tell us which files are in the staging area

all of the information about a repo is sotred under its main directory

Example - appending a file, staging, then committing
$ echo "49,M,No,Yes,Never,Yes,Yes,No" >> mental_health_survey.csv
$ git add mental_health_survey.csv
$ git commit -m "Adding one new participant's data"

The power of git > letting us check the current state of our files versus at other times
compare the last committed version of a file with the unstaged version
example
git diff report.md
here the output would show us two versions
-a is the last one saved
-b is the one we have not added to the staging area



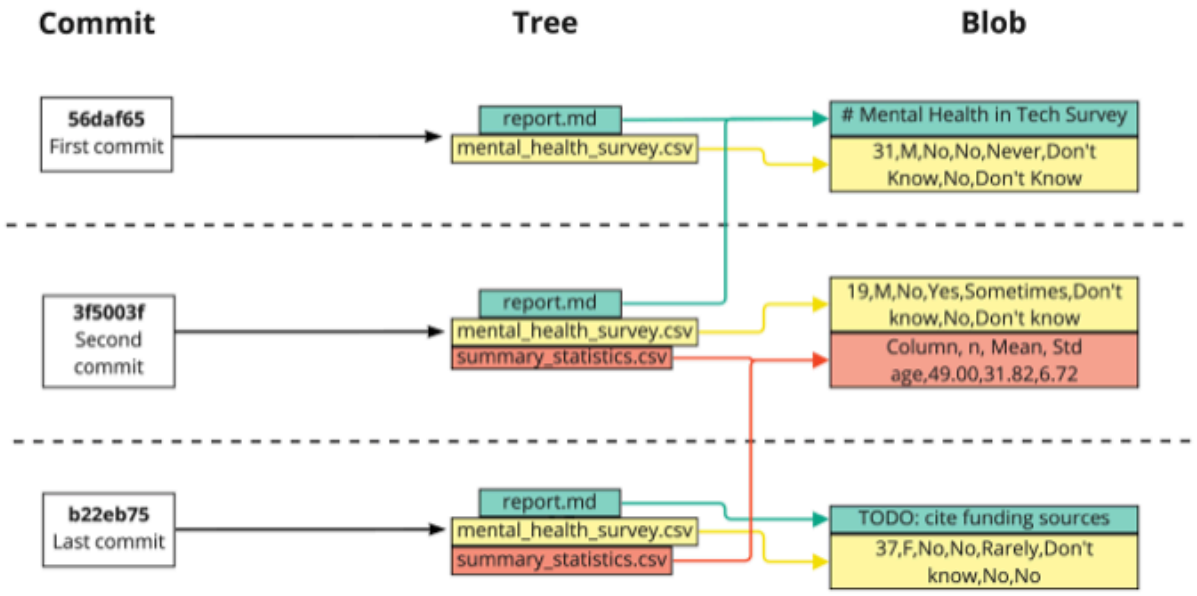
the two @ symbols tells us the location of the changes
the '-1,5' tells us one line was removed at line 5
the '+1,5' tells us one line was added back in at line 5

What if we had already added the file to the staging area?
still use the git diff
but now we add -r which indicates that we want to look at a particular version of the file
we add HEAD, which is a shortcut for the most recent commit
-r won't work unless it is followed by HEAD
git diff -r HEAD
the output will be similar to above

Recap
compare an unstaged file with the last committed version:
git diff filename
compare a staged file with the last committed version:
git diff -r HEAD filename
compare all staged files with the last committed versions:
git diff -r HEAD

The commit structure
three parts:
-commit > contains the metadata (author, commit message, and time of commit) - identified by a unique identifier (called a 'hash')
-tree > tracks the names and location sin the repo when that commit happened
-blob (binary large object) > contains data of any kind, compressed snapshot of the contents of the file when the commit happened

visualization:



can see the flow to the blob is connected to changes

git log
displays all commits made to the repo in chonological order
starts with the oldest
shows the commit hash, author, date, and commit message
*a colon (:) at the end indicates there are more commits
can move through the history by pressin the space bar
can exit the log and return to the terminal by pressing q

a Git hash
a 40 character string of numbers and letters
git produces it by using a pseudo-random number generator called a hash function
hashes allow data sharing between repos

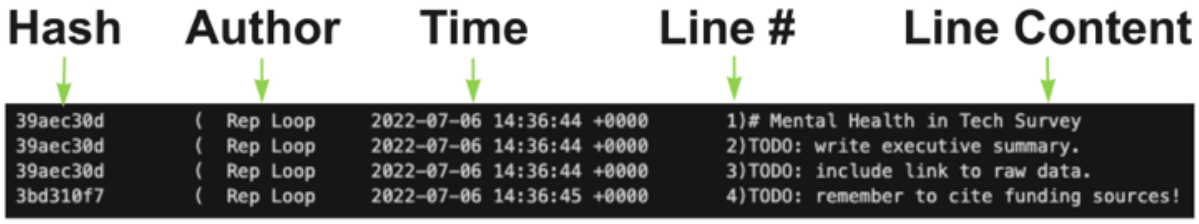
When reviewing commits
need to find the day in question
then with git log can scour through the commits using the first 6-8 characters of the hash to find what we are looking for
then use git show to see the log, the diff, and potential error (ie data entry error)

Using the HEAD shortcut
git diff -r HEAD
compares staged files to the version in the last commit
adding a ~ gives us more versatility with this command

~1 is the path to the second most recent commit
~2 to the third
**be aware must not use spaces before or after the tilde or the command will not work

git show is useful for viewing changes made in a particular commit
git diff allows us to compare changes between two commits (use the two hashes you wish to compare)
example
git diff 35fb4d 186398f
or
git diff HEAD~3 HEAD~2

git annotate
allows us to see who and when made changes
example
git annotate report.md
output>



Unstaging a single file in Git
git reset HEAD filename

Unstaging all files
git reset HEAD

**Undoing changes to a file in the repository
git checkout — filename
'checkout' means switching to a different version
defaults to reverting to the last commit
*means losing all changes made to the unstaged file FOREVER

Undoing changes to all unstaged files
git checkout .
here . refers to the current directory when used in a shell command
*command must be run in the main directory

Unstaging and undoing process
git reset HEAD > unstage all files
git checkout . > revert to last commit and replacing all version of files in the repo
git add . > save the repo in this state we then need to add all files to the staging area
git commit -m ‘Restore repo to previous commit’ > need to solidify with a commit

Nice example
$ git reset HEAD mental_health_survey.csvUnstaged changes after reset:
M	data/mental_health_survey.csv
$ echo '41,M,Yes,No,No,No,Often,Yes' >> mental_health_survey.csvgit
$ git add mental_health_survey.csv
$ git commit -m "Extra participant"
[main 057dd16] Extra participant
 2 files changed, 6 insertions(+)
 create mode 100644 summary_statistics.csv

Dealing with project scale
git log can become tedious as a project becomes abundant with commits
using the dash (-) we can relay how many of the most recent commits to send back
we can also add the filename if we are only looking for the commit history on one specific file
example
git log -3 report.md

can also get log output via date
git log —since=‘Month Day Year’
example
git log —since=‘Feb 2 1985’
for a range
git log —since=‘Sep 12 2023’ —until=’Sep 18 2023’

Restoring an old version of a file
reverting to a version from a specific commit
we do this by replacing the two dashes with the desired git hash
git checkout dc9d8fac mental_health_survey.csv
if recent known version can also use HEAD~

Cleaning a repository
if you want to delete files that are not currently being tracked
these would be unsaved files
git clean -n

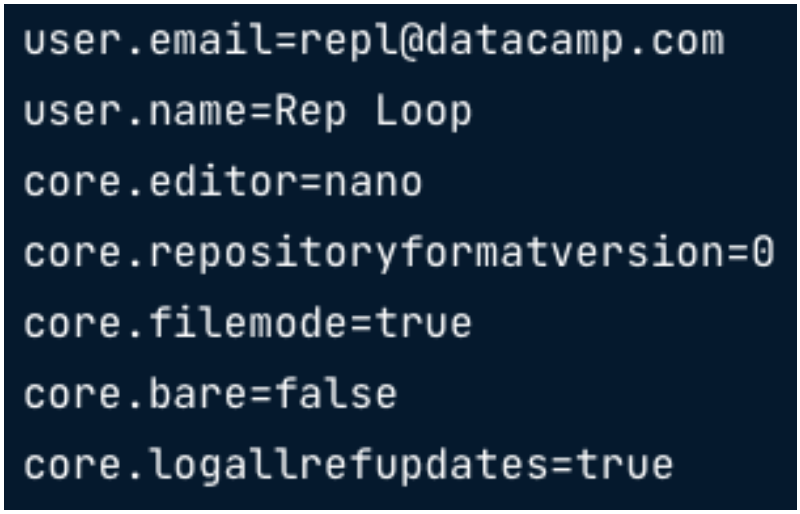
this will give us a list of all untracked files
git clean -f
this deletes these files FOREVER

Example
$ git log -2 report.md
$ git checkout 36b761e4 report.md
$ git add report.md
$ git commit -m "Restoring version from commit 36b761"

Configuring Git
customizable settings
three levels of settings:
- —local
- —global
- —system

*local settings take precedent over global settings which takes precedence over system settings

git config —list
gives us a list of all customizable settings



certain git commands require your credentials
so including them in your configuration saves time

How to change these settings?
git config —global setting value
change email address
git config —global user.email mp@mpmail.com
change username
git config —global user.name ‘Matt Po’

*quotes are needed if you have spaces

Using an alias
*typically used to shorten a command
example
git config —global alias.ci ‘commit -m’
we can now commit files by executing > git ci
can create a custom alias for any command
another example
git config —global alias.unstage ‘reset HEAD’
**be sure not to overwrite existing commands

Tracking aliases
found in a .gitconfig file
git config —global —list
ouput>
alias.ci=commit -m
alias.unstage=reset HEAD

git ignore
create a file using nano
nano .gitignore
within the nano text editor we add *.log
the * is a “wildcard” meaning git will ignore any files ending with .log
commonly ignored files > API keys, system files, software dependencies

Why ignore?
Ignoring files in Git is useful for several reasons:

1. **Preventing Unintentional Commits**: Sometimes, there are files that you don't want to be tracked by Git, such as temporary files, logs, or compiled binaries. Ignoring them prevents accidentally committing them to the repository.

2. **Reducing Repository Size**: Large files or files generated by the development process (like build artifacts) can bloat the repository size. Ignoring them helps keep the repository smaller and more manageable.

3. **Maintaining Security and Privacy**: There may be sensitive information in your project, such as passwords or API keys, that should not be shared in a public repository. Ignoring specific files ensures they are not accidentally committed.

4. **Improving Performance**: Ignoring generated files or folders can speed up operations like cloning, pulling, or pushing from and to the repository.

5. **Focusing on Relevant Changes**: When you're reviewing changes in version control, you want to see the modifications that are relevant to the code and its functionality. Ignoring non-essential files helps you focus on what matters.

6. **Standardizing Development Environments**: Ignoring environment-specific files (like IDE settings or operating system-specific files) helps in maintaining a consistent development environment across team members.

7. **Facilitating Collaboration**: Ignoring files that are generated by the build process means that each developer can build the project independently, without needing to share generated files in the repository.

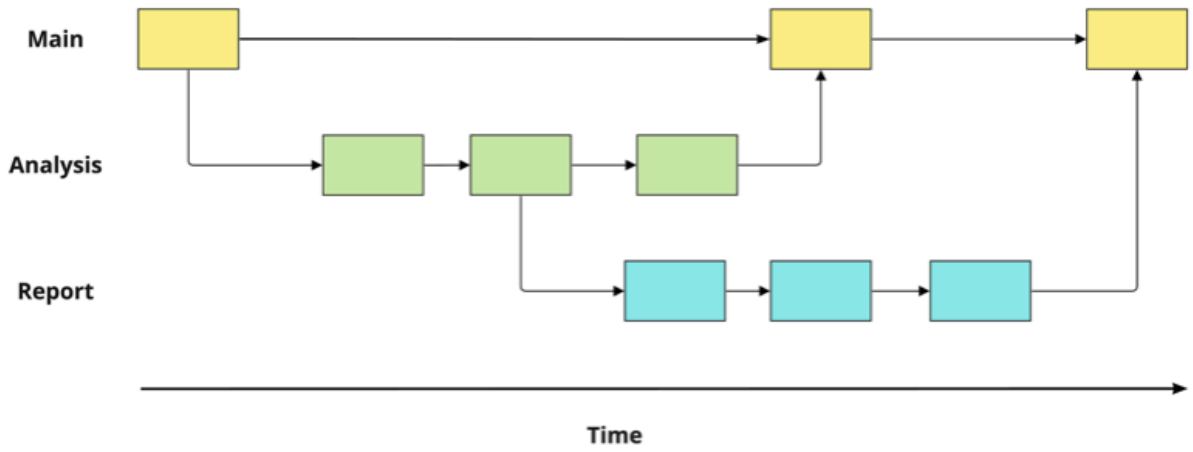
8. **Avoiding Merge Conflicts**: Ignoring files that are generated automatically can help avoid unnecessary merge conflicts, as these files are not meant to be manually edited.

9. **Improving Clarity**: By ignoring files that are not directly related to the codebase, the repository's structure becomes cleaner and more focused on the project's source code.

10. **Compliance with Version Control Best Practices**: Ignoring certain files aligns with best practices for using version control systems and helps maintain a clean and organized repository.

Overall, using Git's ignore mechanisms (such as `.gitignore` files) helps ensure that version control is focused on tracking the source code and related assets that are essential to the project's development process.

Branches
systematically track multiple versions of files
branches helps us avoid the excessive creation of subdirectories
visualizing this:



When merging two branches the commits are called parent commits
source > the branch we want to merge from
destination > the branch we want to merge into

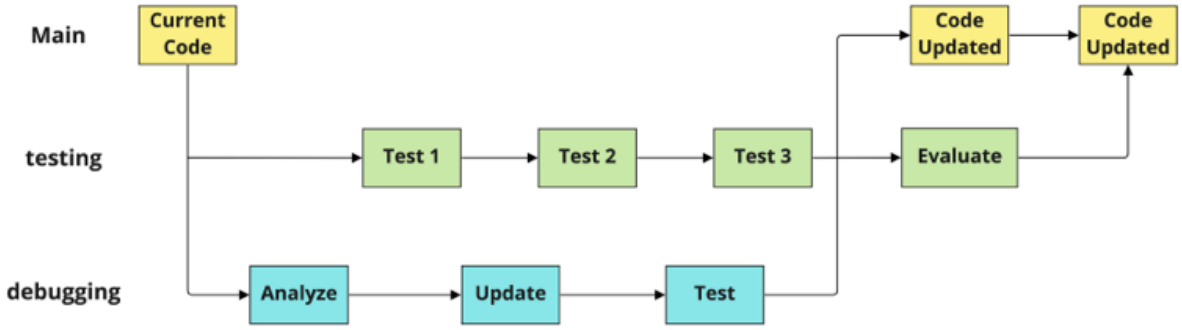
Identifying branches
git branch
output>
shows a list of branches
an * will be next to the branch that we are currently in

Creating a new branch
git checkout -b newbranchname

How to check differences between branches?
git diff branch1 branch2
example
git diff main summary-statistics

Example - modifying a file, staging, committing, then creating and switching to a new branch
$ echo 'Mean age is 32 years old, with a standard deviation of 6.72' >> summary_statistics.txt
$ git add summary_statistics.txt
$ git commit -m "Adding age summary statistics"
On branch summary-statistics
nothing to commit, working tree clean
$ git checkout -b report
Switched to a new branch 'report'

The power of branches



the testing branch allows us to work out a new feature on our project
can build it out
test it
validate it
once we confirm it works and is suitable for the project we can then merge it back into the main branch
the debugging branch allows us to analyze the logs, make changes, refactor our code, and test it out

Switching branches
run git checkout without the b-flag
git checkout branchname
example
git checkout debugging

Think of main branch as the ‘ground truth’
each subsequent branch is a in progress task
when those tasks are completed, they should be merged back into the main branch

How to merge branches?
git merge source destination
example
git merge summary-statisitics main
output>
shows the last two commit hashes from each branch at the top
shows the type of merge (example ‘fast-forward’ merge, which means that additional commits were made on the off-shoot branch and now the main branch is being brought up to date)

Example - switched into new branch, did edit, staged, committed
$ git checkout reportSwitched to branch 'report'
$ echo "80% of participants were male, compared to the industry average of 67%."

>> report.md
$ git add report.md
$ git commit -m "Add gender summary in report"
[report a441929] Add gender summary in report
 1 file changed, 1 insertion(+)

What is a conflict in Git?
occurs when a file in different branches has different contents that prevent them from automatically merging into a single version
example - we have a todo list where we added tasks while on the main and then added an additional task while on a branch called ‘update’
after working on ‘update’ branch, we switch back to the main branch
there we finish two of the tasks on the todo list, we then remove them from the todo list
it now looks like this in our tree:

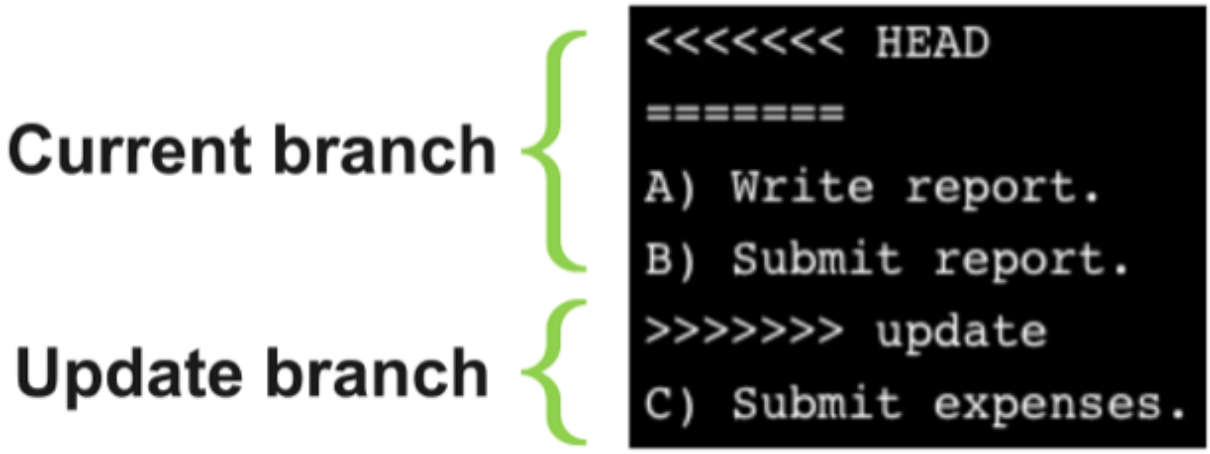
Main branch `todo.txt`

```
C) Submit expenses.
```

Update branch `todo.txt`

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

we have different versions in the main and update branches
this is a conflict!
now we can’t merge these two branches
how to resolve this?
open the conflicted file in the nano text editor



understanding this
the first line, arrows pointing to the left indicates that lines beneath it contain the

file’s contents in the latest commit of the current branch
line with equal signs refers to the center of the conflict
*if the equal signs come after some content, this would mean our files have different content on the same lines in different branches
arrows pointing right, indicates additional content within the other branch
in nano text editor make the changes
then put to staging
git add filename
git commit -m
then attempt the action that lead to the git conflict message

Creating a new repo
git init newprojectname

Can also convert a project
from an existing directory we can also type:
git init
this will convert the directory to a Git repo
in this case it will start the repo with files from that directory
Git will tell us to stage and commit these files
**key do not create nested repositories
this will confuse Git down the road, as two .git directories will be formed
Git will not know which directory to update in the future

Example - create new repo, change into that directory, add/modify file
$ git init anxiety_workplace
Initialized empty Git repository in /home/repl/projects/anxiety_workplace/.git/
$ cd anxiety_workplace
$ nano todo.txt
#in nano text editor add message (save with Ctrl+O then hit ‘enter’ to confirm, exit with Ctrl+X)

Working with remotes
‘remotes’ means remote repos
local repo refers to our computer
remote repo refers to a cloud based system (such as GitHub)
benefits are that our project is not lost if our computer breaks down
we and others can access our project from anywhere

Cloning locally
cloning is copying existing remotes to our local computer
git clone path-to-project-directory new_project_name
example

git clone /home/john/repo our_first_local_repo

Cloning a remote
cloning a remote from GitHub to our local computer
git clone [URL]
example
git clone https://github.com/datacamp/project

Identifying a remote
Git remembers where the original project is from
Git stores a remote tag in the new repo's configuration
can pull this tag:
git remote
above output>
datacamp
git remote -v
#stand for verbose
this will give us more info
like the URL and whether it was fetched or pulled

Creating a remote
Git automatically names the remote 'origin'
we can add more remotes
git remote add name URL
example
git remote add george https://github.com/george_datacamp/repo

Fetching from a remote
we want to compare files from local repos to remote repos
here the local repo is the work in progress and remote is the ground truth
to start we need to fetch versions from the remote
git fetch origin main
#here we are defining the name of the remote and the local branch to fetch into
can fetch into any branch
example
git fetch origin report

After fetching, the contents of the remote will be in our local repo
now we need to synchronize the contents between the two repos
to do this we perform a merge of the remote into the local repo's main branch
git merge origin main
ouput>
shows use commit hashes, type of merge, what files were changed

**Remote is often ahead of local repos
*fetch then merge is a common workflow
Git simplifies this process by combining both commands into a ‘pull’
git pull origin main
*remember to commit all local changes prior to merge or all unstaged and staged changes will be overwritten

Pushing to a remote
opposite of a pull
push content from a local repo to a remote repo
save changes locally first
git push remote local_branch
example
git push origin main
*key to synchronize before push
which entails pulling the most recent remote
Git will open a text editor at this request since there are different commits on the local and remote
add a message for the reason of the pull
can avoid leaving a message with this:
git pull —no-edit origin main
*this is not recommended
finish with
git push origin main

Git cheat sheet
https://www.datacampo.com/cheat-sheet/git-cheat-sheet