`git` basics

What is `git`?

- version control system
  - Keeps track of *changes* in text files which are stored in a `git` **repository** aka **repo**
    - Designed to track files such as `*.md`, `*.cpp`, `*.py`, `*.tex`, etc.
    - Does not *track changes* in binary files, `*.docx`, `*.pdf`, etc.

How does `git` track changes?

- `git` is essentially a database of SHA-1 hashes
  - Every file has a unique hash
  - Change a file, even by one character, -> new hash

Example

You track a file called `file.txt` with the following contents:

```
lemons, appes
```

`git` stores the file with the hash `19d86005e6736978f8ff20392eb1feb88cdad77b`

Example Continued

If you change `file.txt`

```
lemons, apples
```

`git` see the file has been modified. It's new hash is `a0254d0a3e9a29ea85e204525b445c49b385a139`

Optional Exercise

Copy and paste the following commands, one by one, into your terminal. (You do not have to be in a git repo.)

```
echo "lemons, appes" > file.txt
git hash-object file.txt
echo "lemons, apples" > file.txt
git hash-object file.txt
```

You should see the same hashes as in the slides

What is `Github`?

- A cloud based `git` repository system
  - Online storage for `git` repos.
    - Enables multiple collaborators to work on one project
    - Enables a back up of a repo in the cloud
- In `git` parlance, the `Github` hosted version of a repo is the **remote**

NOTE: Github is not the only online repository system. Some alternatives are `gitlab.com`, `bitbucket.org`, `sourcefourge.net`

How to create a local `git` repository?

```
mkdir git_repo          # Or use existing folder
cd git_repo
git init
```

- This creates a `.git` directory in `git_repo`.
- All your `git` information is stored in here.
  - **If you delete this you have lost all your `git` history.**
- Similarly, if you accidentally run `git init` in the wrong folder just delete the `.git` folder with `rm -rf .git`.

Tracked, untracked and staged files in `git`

- Files in `git` are either `tracked` or `untracked`
- When you create a new file in a `git` repository, that file is `untracked`
  - Likewise, if you run `git init` in a folder that already has files, all those files are `untracked`
- To track a file you need to `add` it
- After you `git add` a file, it now called a `staged` file

`git status`

To check whether a file is `staged` or `unstaged`, `tracked` or `untracked` you can run `git status`

```
$ git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
```

Example

```
touch file.txt    # This creates an empty file called file.txt
git add file.txt
```

- `git add` **stages** your file for `commit`
- If you accidentally `add` a file you can unstage it:
  `git restore --staged <file_name>`
- To add multiple files
  - `git add *` will add all files except hidden files
  - `git add -A` will add all files including hidden files

`git commit`

- Once a file is `staged` it can be commited
- A `commit` is snapshot of your repo. It's like a save point
- If you save your code in a stable state and later make a mistake which breaks everything, you can always go back to a `commit` that works

`git commit`

- If you type `git commit`, a text editor will open for you to write a `commit` message
- If you don't have a default editor set up, this will open `vi`
  (If you haven't used `vi` before type `:q <Enter>` to quit)
- You can learn how to use `vi` (recommended) or change the default editor to something else in your `git config`
- This will set `nano` as the default editor:
  `git config --global core.editor nano`

`git commit` messages

- You ***MUST*** write a commit message (`git` won't let you commit without one).
  - You ***SHOULD*** write a commit message that is useful and explains why you have changed something
  - You ***SHOULD*** commit your changes often, don't leave it to the end of the day/week/month to commit changes to git

Good commit message example

```
A brief description of the change

* A longer description of why you have made the change
* If the change fixes something what does it fix.
```

- Writing Good Commit Messages A Practical Guide
- How To Write Good Git Commit Messages
- How To Write A Good Git Commit Message

Bad commit messages example

```
7ba3ab6 jghjdfkghddkjgh
c7ea2f2 YAY IT WORKS!!!!!
b98cd9d doesn't work :(
```

- The only time you should write a short message is for a trivial thing like a typo fix
- For a short commit message use `-m` for message:
  `git commit -m "Fixed typo in README"`
- If you write a bad commit message, change it with
  `git commit --amend`

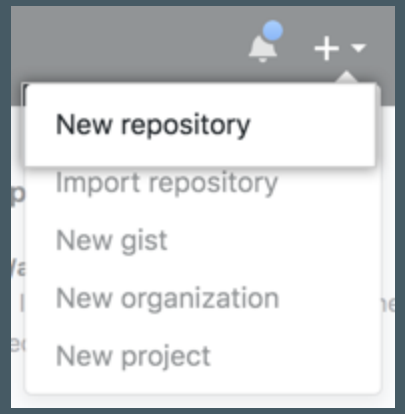
`git log`

```
$ git log
commit 283f718de0d76b30e0d072b69cce54789479134c (HEAD -> main)
Author: Your Name <you@example.com>
Date:   Sun Feb 6 18:45:57 2022 +0000

    Initial commit
```

- Every `commit` has a hash
- The command `git log` lists all the commits in a repo

Adding an existing local repository to `Github`



- Click on the + in the top right hand corner and click `New repository`
- Give the repo a name
- Choose `Public` or `Private`
- Click `Create repository`

Adding an existing local repository to `Github`

- To link your local repo to your Github repo, type the following in your terminal (make sure you are in your git repo):

```
git remote add origin
https://github.com/<user_name>/<repo_name>.git
```

- If your repo is private you will have to either create a personal access token
- Or set up an ssh key and use the ssh url:

```
git remote add origin git@github.com:<user_name>/<repo_name>.git
```

Adding an existing local repository to `Github`

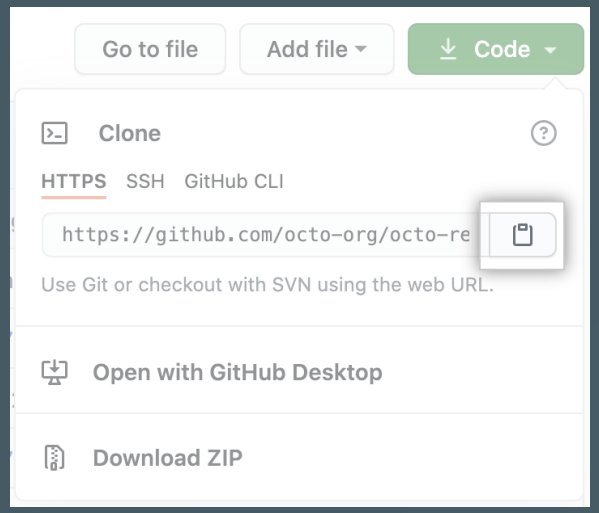
- `git remote add origin <url>` tells git that you have a cloud based repository and you want to link it to your local repo
- `origin` is the conventional name for a remote, but you can call it whatever you want and even have multiple remotes.

Adding an existing local repository to `Github`

```
# You must have made at least one commit for this to work
git branch -M main          # Optional: changes the branch name to main
git push -u origin main
```

- The default branch created by `git` is called `master` by default. For reasons Github have moved to calling the default branch `main`. This is optional, the default branch could be called anything
- `git push` is the command to upload the local repo to the remote

`git clone`



- To download a `Github` repo to your computer, you `clone` it with `git clone`
- You can find the `https` or `ssh` url by clicking on the green code button on the repos main page

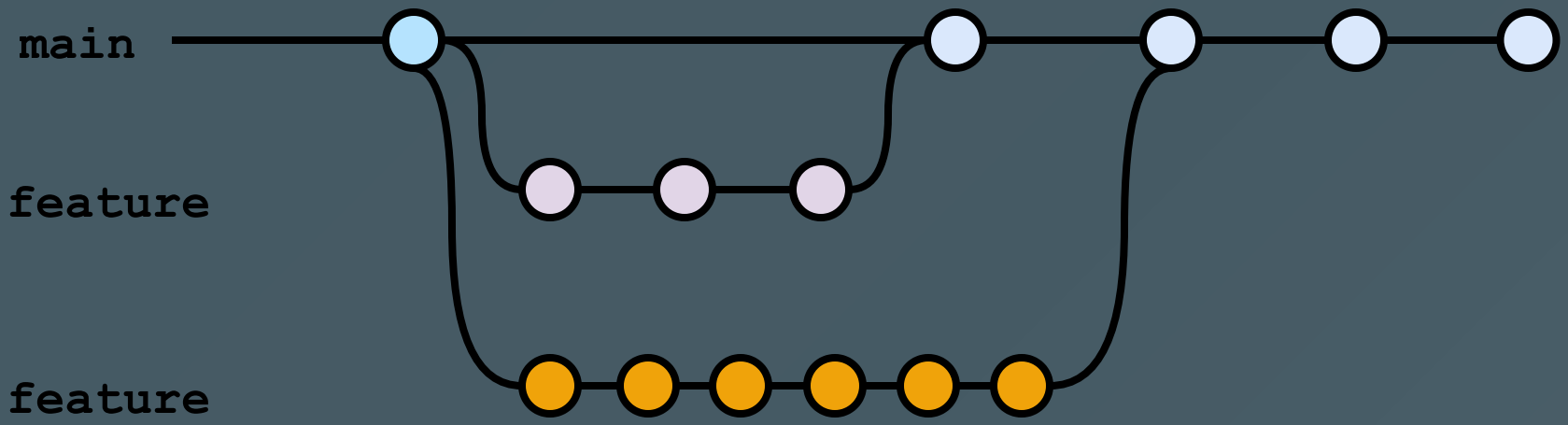
Some `git clone` examples

- `git clone <url>` clone the repo to a folder which has the same name as the repo
- `git clone <url> <directory_name>` clone the repo to a folder with the name `<directory_name>`
- `git clone --branch <branch_name> <url>` clone a single branch

`git` branches and workflows

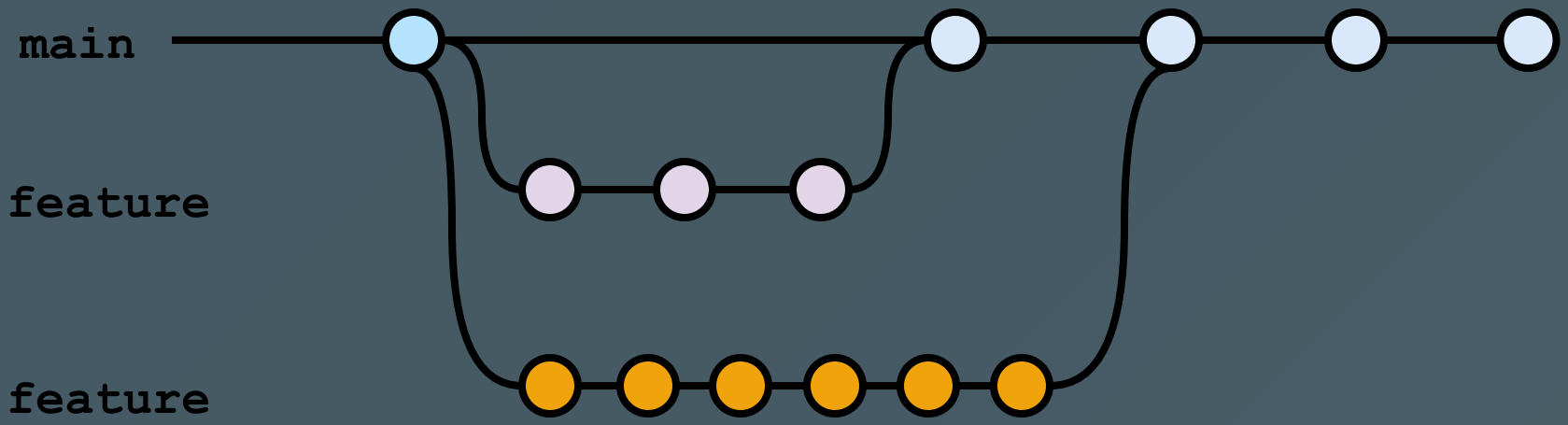
- When you initialise a repo you have one branch
- Using and committing to only one branch is a "basic" git workflow.
- This works well when only one person is working on a repo
  - A latex thesis

`git` branches and workflows



- When more than one person is working on a project it is a good idea to use more than one branch.
- Often these are called feature branches this workflow is known as a 'feature branch workflow'

`git` branches and workflows



- `main` is always in a stable state that works and is 'deployable'
- To work on something new, create a feature branch
- When the feature is complete - create a `pull request` on `Github`
- When someone else has checked the new code doesn't break anything it can be merged into main

`git` branches and workflows

- There are other git workflows
  - `gitflow`
  - `forking workflow`
  - `trunk based development`

What workflow you use isn't really important but it's a good idea to have one way of working and stick to it.

`git branch` and `git checkout`

```
git branch <branch-name> # create a branch
git checkout <branch- name> # switch to new branch
```

Equivalently, use the following one liner:

```
git checkout -b <branch-name>
```

`git checkout` can also checkout a commit:

```
git checkout <commit_hash>
```

`git branch` and `git checkout`

To list all the branches in a repo:

```
git branch -a
```

To delete a branch:

```
git branch -d <branch_name>
```

`git merge`

To combine one branch with another use `git merge`

```
git checkout <branch-to-merge-into>
git merge <branch- to- be -merged>
```

If you are working on a project with others don't merge into main locally and push your changes to git make a Pull Request (PR)

`Github` pull request

- `gitlab` calls this a merge request, which is a much better name

Merge Conflicts

- Sometimes changes are made in the same file in different branches
- If git can't work out how to merge these files it will warn of a merge conflict and refuse to merge until it is resolved
- Use `git status` to check where the issue is and then open the file(s) with conflicts

Merge Conflicts

```
<<<<<<< HEAD
Something in the current branch
=======
Something different in the branch to be merged
>>>>>>> Branch
```

- HEAD is the current branch
- Everything above = is from HEAD everything below is from the branch to be merged

Merge Conflicts

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   file.txt
```

- Delete the parts you don't want to keep and save the file
- Run git status and check what it says
- If it says all conflicts resolved: `git commit`

Avoid Merge Conflicts

- You can avoid merge conflicts by:
  - Not working on the same files as others
  - Keeping up to date with changes in the remote

`git pull`

- Your local repo has a main branch
- Your `Github` has a branch called origin/main
- If changes are made to origin/main, you need to merge them into the local main
- This is what `git pull` does
  - First it `fetches` your code from `Github`
  - Then it merges it into your local main

`git fetch`

If you don't want to `merge` the changes you can simply `fetch`

```
git fetch
```

and then merge them later

```
git merge origin/main
```

- This is safer because you won't accidentally lose your changes you should always `pull` (or `fetch`) before you `push`

If you lose a `commit` or want to undo a `merge`

```
$ git reflog
adc237  HEAD@{0}: merge origin/main into main
c98ade8 HEAD@{1}: fetch origin/main
2b91bcd HEAD@{2}: before the merge
$ git reset --hard HEAD@{2}
```

- `git reflog` keeps track of `git refs` (pointers to commits)
- `git reset --hard <REF>`
- Deletes all the changes and gets you back to where you were
- Omit hard if you don't want to delete things