

# Version Control with Git

*Working in the Terminal*

**Fathom Training**  
@fathomdata

now you know

# Intentions

## After this course:

1. You will understand the importance of Version Control.
2. You are motivated to use Git and GitHub.
3. You will be comfortable with using Git commands.

# Outline

1. What is Version Control?
2. Why Version Control?
3. What are Git and Github?
4. Why Git and GitHub?
5. Git's special features

Break

6. Explore Git through the terminal
  - Committing, pushing and merging
  - Cloning
  - Rebasing

# Online Participation

## Important notes:

- Please keep yourselves muted unless you are asking a question
- Questions via text or voice are great
- This session will be recorded for internal use at Fathom
- Feedback is very much desired – what can we do better?

**Time to learn!**

# YOU now

- Start a new project
- Edit the project and save multiple versions
- Some versions are overridden by new version
- Some data are lost

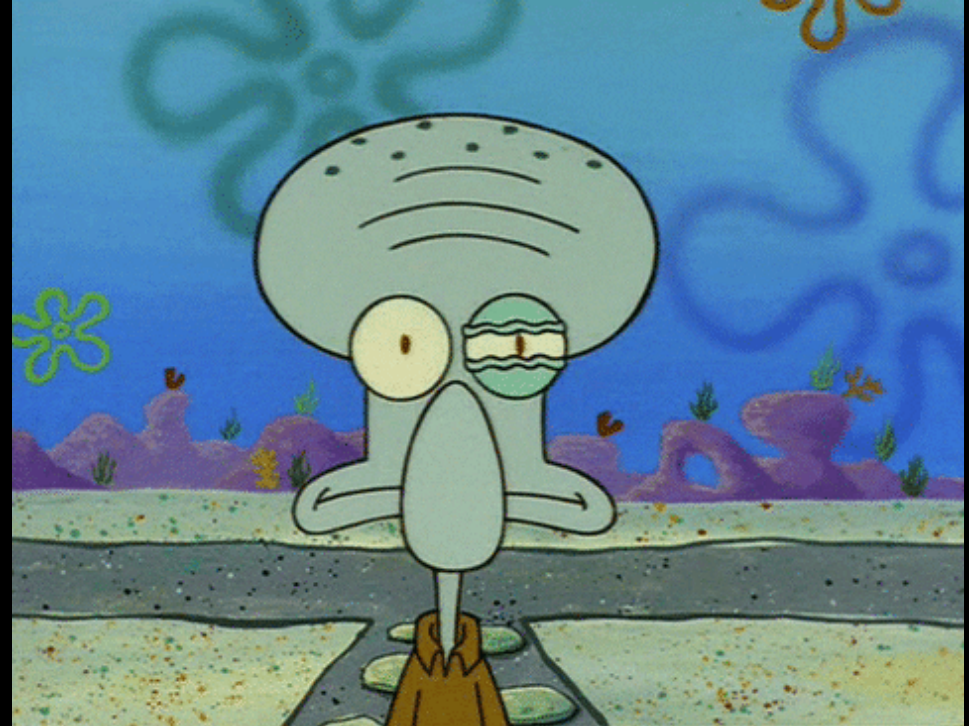
Six months later.....

Name

|   |   |
|---|---|
|    | Super Cool Report v1.xlsx                       |
|    | Super Cool Report v2.xlsx                       |
|    | Super Cool Report v3.1.xlsx                     |
|    | Super Cool Report v3.xlsx                       |
|    | Super Cool Report v4.xlsx                       |
|    | Super Cool Report v4a.xlsx                      |
|    | Super Cool Report v4b.xlsx                      |
|    | Super Cool Report v5.xlsx                       |
|   | Super Cool Report vFinal.xlsx                   |
|  | Super Cool Report vFinal_1.xlsx                 |
|  | Super Cool Report vFinal_2.xlsx                 |
|  | Super Cool Report vFinal_Final.xlsx             |
|  | Super Cool Report vFinal_Final-UPDATED.xlsx     |
|  | Super Cool Report vFinal_Final-UPDATED_NEW.xlsx |

# The FUTURE YOU

- Project needs updating
- Open the old project
- Lost the data
- Error when running code!
- File named final version is not the last file modified



# The Danger of Using File Date and Time

- Unreliable
  - What do they really represent? Modified? Opened?
  - Did you modify the wrong file before?
  - Was the real latest file deleted?
- Other Problems:
  - What changed between files?
  - What issues are there?
  - How do teams collaborate?





# Version Control

- What is VC?
  - Often referred to as "source control"
  - Help teams manage changes to projects over time
  - Tracking and managing changes to files
- Why use VC?
  - Collaboration: Simultaneously work on the same project
  - Storing Version: Record the development of a project
  - Restoring Versions: Restore older versions of a file
  - Understanding: Descriptive commit message
  - Backup: Saving your work in remote repository

# Git

## VC with Git and GitHub

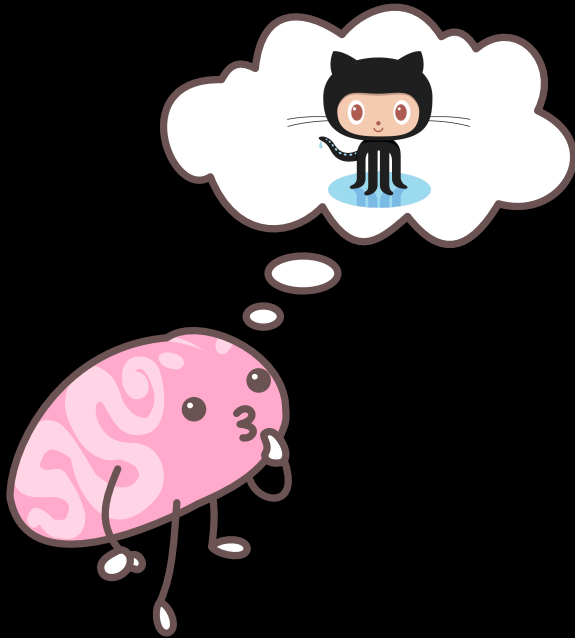
We highly recommend making use of the Git [cheatsheet](#)

Our intention is to arm you with the ability to:

1. Organise your code locally
2. Create incremental changes and commit these
3. Push your changes to a remote repository
4. Pull changes from a remote repository to your machine
5. Tell Git which files to exclude from VC

# Why Git?

- Allows users to track changes
- Collaborative team work
- Manages evolution of a project
- Allows branching



# Why GitHub?

- Provides home to a project on the internet
- Think about DropBox but a much better version
- It allows people to:
  - view and review your work
  - sync with a project
  - comment or edit your content
- Local corrupt content can be replaced by fresh content saved on Github

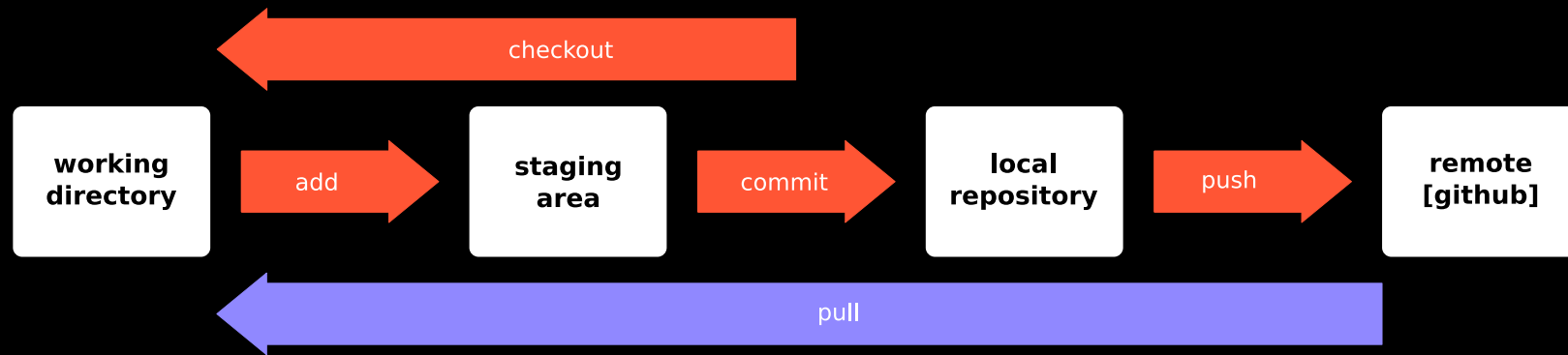
# Git support

- Git is language agnostic – can be used in all programming languages
  - Python
  - R
  - C
  - C++
  - Java
  - JavaScript
  - TypeScript and many many many more

# How Do I speak GitHub?

Common terms we will need to understand while using GitHub

- Repository (repo)
- Branch
- Commits
- Pull/Merge requests
- Issues
- Push/Pull



# Git Permission

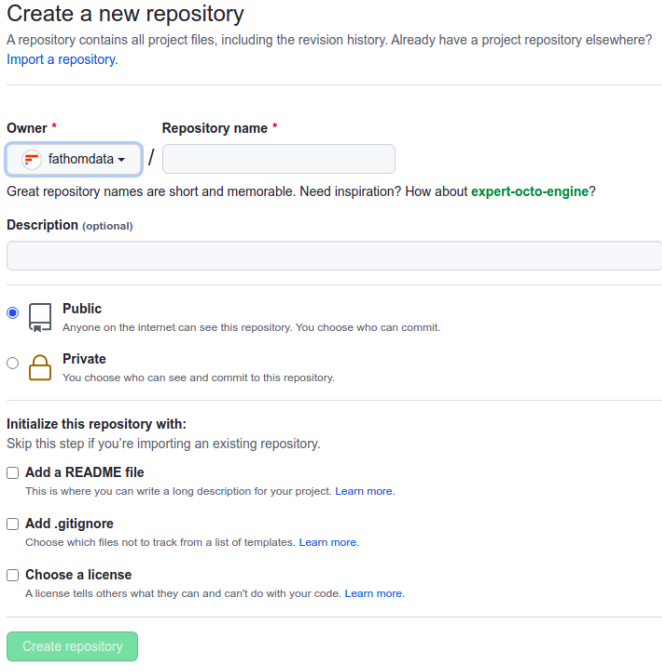
A repository can either be public or private

A public repository:

- Readable by the world
- Others are allowed to push comments

A private repository:

- Invisible to the world
- The owner can grant read/write access to others




The screenshot shows the 'Create a new repository' page on GitHub. At the top, it says 'Create a new repository' and provides a brief explanation: 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)'. Below this, there are two main input fields: 'Owner' with a dropdown menu showing 'fathomdata' and 'Repository name' with an empty text box. A hint below these fields says 'Great repository names are short and memorable. Need inspiration? How about [expert-octo-engine](#)?'. There is a 'Description (optional)' text area. Below the description, there are two radio button options for visibility: 'Public' (selected) and 'Private'. The 'Public' option is described as 'Anyone on the Internet can see this repository. You choose who can commit.' and the 'Private' option is 'You choose who can see and commit to this repository.' Below these options, there is a section titled 'Initialize this repository with:' with the instruction 'Skip this step if you're importing an existing repository.' This section contains three checkboxes: 'Add a README file' (with a sub-note 'This is where you can write a long description for your project. [Learn more.](#)'), 'Add .gitignore' (with a sub-note 'Choose which files not to track from a list of templates. [Learn more.](#)'), and 'Choose a license' (with a sub-note 'A license tells others what they can and can't do with your code. [Learn more.](#)'). At the bottom of the form is a green 'Create repository' button.



Note: during a pull/merge request, the owner of a remote repo has to approve changes before any files can be modified.

# A real life example

Let's explore GitHub. The goal here is not to teach GitHub yet, but rather to understand the end goal. As an example, let's explore the Git [repository](#).

- Explore the following: 
  - The repository
  - The commits
  - The branches
  - Browse the forks
  - View the contributors history



**Time to learn!**

# GitHub account

In case you haven't already done so, create a free GitHub account [here](#)

## Username advice

- Incorporate your actual name
- Shorter is better than longer
- Pick a username that you will be comfortable revealing to your boss

## Free private and public repositories

- Unlimited private repositories
- Support up to three external collaborators

# Configuring Git

Git comes with a `git config` command allowing you to set configuration that controls how Git operates.

Set your user name and email address. This is essential as every Git commit uses this information:

```
git config --global user.name "My username"  
git config --global user.email example@gmail.com
```

# A Quick Refresher

# Bash commands

- `cd`: change the current working directory
- `ls`: list content of a directory
- `mkdir`: create a directory
- `pwd`: print working directory
- `rmdir`: remove directory
- `rm -r`: remove a directory and its content's
- `touch`: create new file
- `grep`: search
- `man`: get help for a command
- `cat`: create a file
- `echo`: echoes its arguments

# Creating a local Git repository

Open up the terminal and let's get started!

Using the `cd` command, direct to the desktop and create the folder `version_control` and with the `git init` command, activate git:

```
cd Desktop
mkdir version_control
cd version_control
git init
```

```
fathom-trainer:~$ cd Desktop
fathom-trainer:~/Desktop$ mkdir version_control
fathom-trainer:~/Desktop$ cd version_control/
fathom-trainer:~/Desktop/version_control$ git init
Initialized empty Git repository in /home/amieroh/Desktop/version_control/.git/
fathom-trainer:~/Desktop/version_control$ echo "this is a git diff test example" > diff_test.txt
fathom-trainer:~/Desktop/version_control$ ls
diff_test.txt
```

# Adding a file

Create a .txt file:

```
echo "this is a git diff test example" > diff_test.txt  
ls
```

We can use the `git status` command to see which files git knows exist: 

```
git status
```

# Staging

To stage a file is simply to prepare it for a commit.

- Staging commands:
  - `git add`: Add file to the staging area
  - `git diff`: Changed but not staged
  - `git diff --staged`: What is staged but not yet committed
  - `git reset [file]`: unstage a file while retaining changes
  - `git commit -m`: commit your staged content as a new commit
  - `git log`: allows you to view information about previous commits



# Setting up your SSH key

With SSH keys, you can connect to GitHub without supplying your username at each commit

Generating a new SSH key:

1. Open the terminal.
2. Generate a new set of keys with:

```
# mkdir $HOME/.ssh  
ssh-keygen -t ed25519 -C "your_email@example.com"
```

3. Press Enter, the default file location will be applied
4. At the prompt, when asked to enter a secure passphrase, Press Enter. When asked to repeat the passphrase Press Enter again

# SSH key

Adding your SSH key to the ssh-agent:

1. Start ssh-agent in the background.

```
eval "$(ssh-agent -s)" # for Mac and Linux  
eval `ssh-agent -s`  
ssh-agent -s # for Windows
```

2. Add SSH private key to the ssh-agent.

```
ssh-add ~/.ssh/id_ed25519
```

# SSH key

Adding SSH key to your account on GitHub:

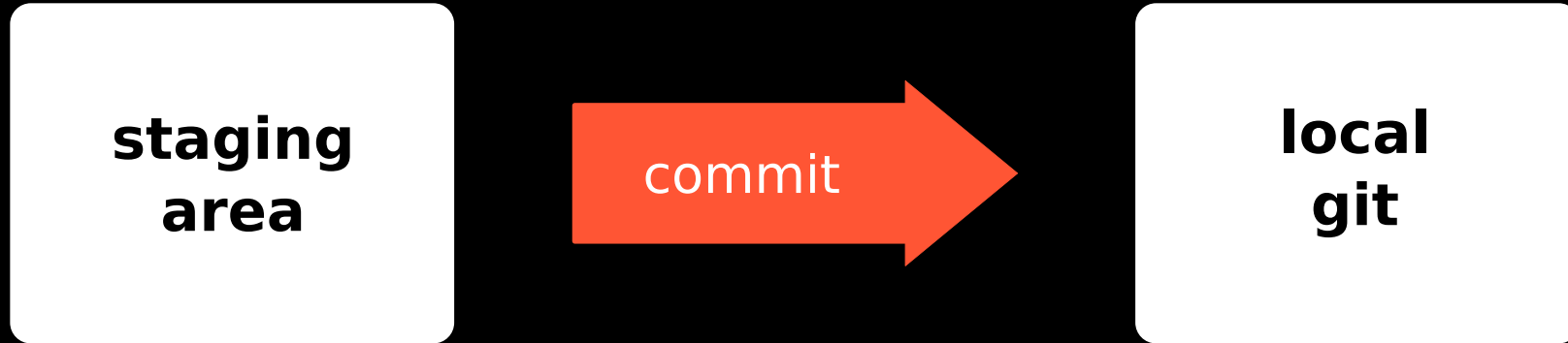
1. Copy the SSH public key.

```
cat ~/.ssh/id_ed25519.pub # Linux  
clip < ~/.ssh/id_ed25519.pub # Windows
```

2. Click on your GitHub profile and click **Settings**.
3. In the user settings sidebar, click **SSH and GPG keys**.
4. Click **New SSH key**.
5. Paste your key into the **Key** field.
6. Add SSH key.

# Committing

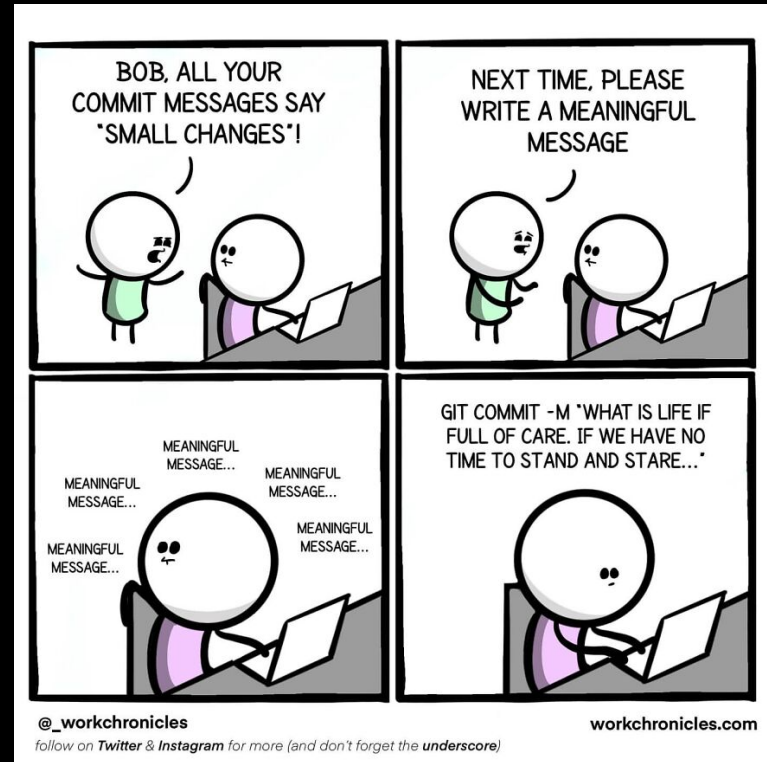
A commit takes a snapshot of your work at a specified point in time.



# Committing

A commit takes a snapshot of your work at a specified point in time.

- Changes that describe which files were added, modified and deleted
  - A human-readable commit message
  - Be concise
  - Describe the *why*





You might think that you are working alone on a project, but keep in mind that you have one very important collaborator: **Future you!**

**Commit often / Push regularly**

# Committing

Use `git add` to add the file to Git staging then commit: 📄

```
git add .  
git commit -m "My first commit"  
git status  
git log --all
```

```
fathom-trainer:~/Desktop/version_control$ git add .  
fathom-trainer:~/Desktop/version_control$ git commit -m "My first commit"  
[master (root-commit) 8a9d952] My first commit  
 1 file changed, 1 insertion(+)  
 create mode 100644 diff_test.txt  
fathom-trainer:~/Desktop/version_control$ git status  
On branch master  
nothing to commit, working tree clean  
fathom-trainer:~/Desktop/version_control$ git log  
commit 8a9d952b72e8c19e27bc484252a1ff626631594e (HEAD -> master)  
Author: AmierohAbrahams <amierohabrahams@gmail.com>  
Date:   Mon Oct 18 09:08:35 2021 +0200  
  
    My first commit
```



# Git diff

Executing this command will change the content of the diff\_test.txt file:

```
echo "this is a diff example" > diff_test.txt  
git diff
```

```
fathom-trainer:~/Desktop/version_control$ echo "this is a diff example" > diff_test.txt  
fathom-trainer:~/Desktop/version_control$ git diff  
diff --git a/diff_test.txt b/diff_test.txt  
index 6b0c6cf..b37e70a 100644  
--- a/diff_test.txt  
+++ b/diff_test.txt  
@@ -1,1 @@  
-this is a git diff test example  
+this is a diff example
```

1. This line displays the input sources of the diff
2. This line displays some internal Git metadata
3. These lines are a legend that assigns symbols to each diff input source
4. We have -1 +1 meaning line one had changes

# Git log

Display all commits:

```
git log --all
```

View the most recent commits:

```
git log -3
```

Filter commits By Author or Committer:

```
git log --author <name>  
git log --committer <name>
```

# Git log

Filter commits by X days ago:

```
git log --before <date>  
git log --after <date>
```

Filter commits by date range:

```
git log --after <date> --before <date>
```

View all changes for each commit

```
git log -p
```

# Git log

View summary of changes for each commit:

```
git log --stat
```

View one line per commit:

```
git log --oneline
```

Format the Git Log Output

```
git log --pretty=format:"<options>"
```

# Creating a Remote Repository on GitHub

1. Login to your Github account
2. Select the '+' icon. Then select 'New Repository'
3. Give your repository a name — ideally the same name as your local project
4. Click 'Create Repository'

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*

fathomdata

Repository name \*

version\_control

Great repository names are short and memorable. Need inspiration? How about [musical-adventure](#)?

Description (optional)

☒ **Public**

Anyone on the internet can see this repository. You choose who can commit.

☐ **Private**

You choose who can see and commit to this repository.

### Initialize this repository with:

Skip this step if you're importing an existing repository.

☐ **Add a README file**

This is where you can write a long description for your project. [Learn more.](#)

☐ **Add .gitignore**

Choose which files not to track from a list of templates. [Learn more.](#)

☐ **Choose a license**

A license tells others what they can and can't do with your code. [Learn more.](#)

Create repository

# Linking your local project folder

The screen you should be seeing now on Github is titled '**Quick setup — if you've done this kind of thing before**'.

Copy the link, this will be used when we commit from the terminal to our GitHub repository.

### Quick setup — if you've done this kind of thing before

or

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

#### ...or create a new repository on the command line

```
echo "# version_control" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/fathomdata/version_control.git
git push -u origin main
```

#### ...or push an existing repository from the command line

```
git remote add origin https://github.com/fathomdata/version_control.git
git branch -M main
git push -u origin main
```

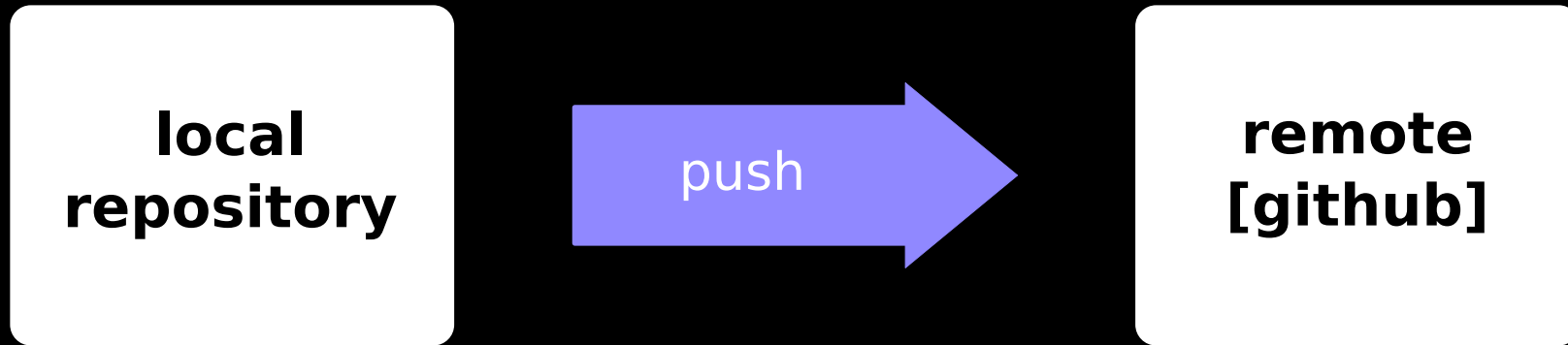
#### ...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

💡 **ProTip!** Use the URL for this page when adding GitHub as a remote.

# Pushing to a remote repository

Upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repository.



# Pushing to a remote repository

Connect to your remote repository and then push to GitHub:

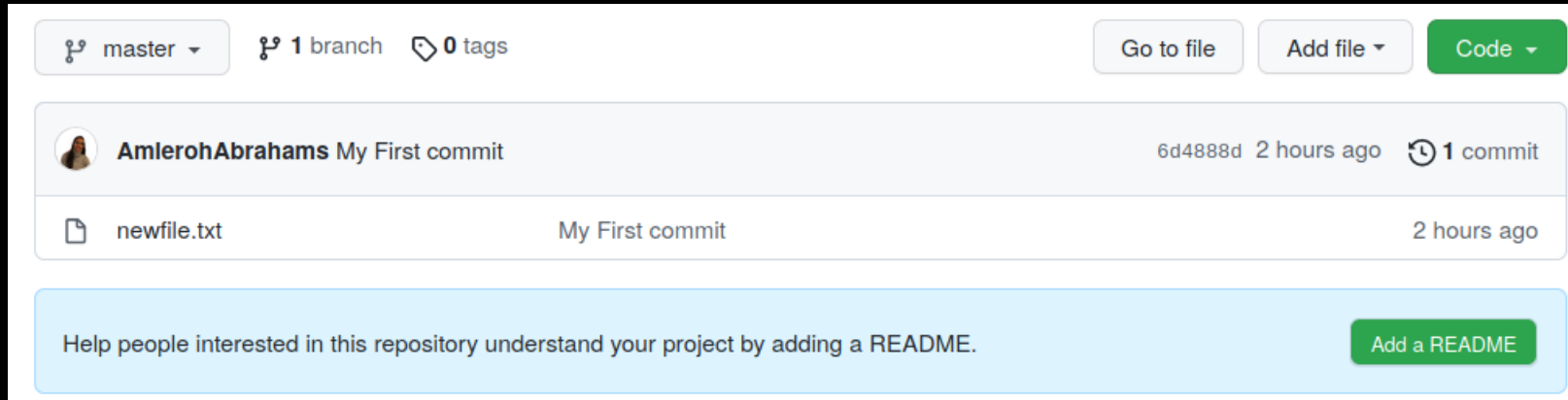
```
git remote add origin <remote repository URL>  
git remote -v  
git push origin master
```

The commit message should explain **why** we made the changes.



# Pulling

- Now that we have committed and pushed change, let's create a README.md.
- A README.md file typically contains information that explains the purpose of the repository – like 'a biography of the repository.



```
git pull origin master
```

# Ignoring files

.gitignore file is a text file that tells Git which files or folders to ignore in a project.

First, create a password.txt and a .gitignore file:

```
echo "#58876" > password.txt  
touch .gitignore  
echo 'password.txt' >> .gitignore
```

# Ignoring files

The password.txt file becomes a hidden file in our folder. Let's commit and push to GitHub and view the outcome:

```
ls -a  
git add .  
git commit -m "Exploring .gitignore"  
git push
```



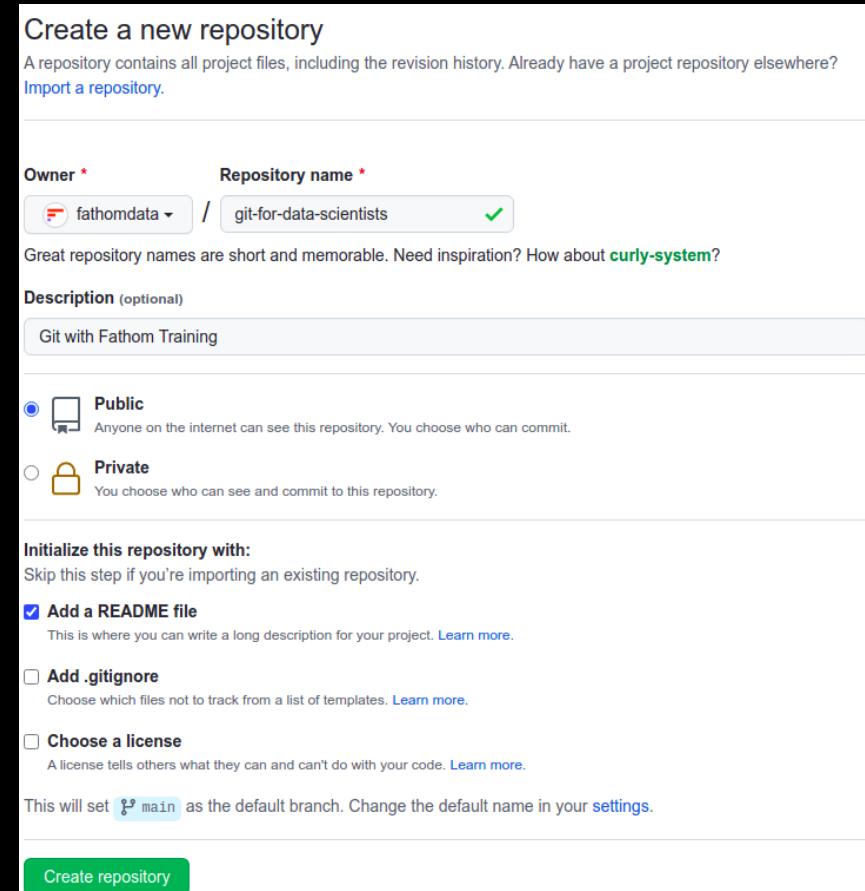
Now that we have had a brief introduction with the terminal and committing, lets **create a new repository on Github** and explore **cloning, branching, merging conflicts** and more.

# Git repository

Go to [GitHub](https://github.com). Click the "New repository" tab then click the green tab "New".

How to complete this:

- Repository name: git-for-data-scientists.
- Description: Git with Fathom Training.
- Public.
- Yes, initialize this repository with a README.



The screenshot shows the GitHub 'Create a new repository' page. At the top, it says 'Create a new repository' and provides a brief explanation of what a repository is, along with a link to 'Import a repository'. Below this, there are two main sections: 'Owner' and 'Repository name'. The 'Owner' is set to 'fathomdata' and the 'Repository name' is 'git-for-data-scientists', which is marked with a green checkmark. A note below these fields suggests that repository names should be short and memorable, with a link to 'curly-system?'. The 'Description' field is optional and contains the text 'Git with Fathom Training'. Below the description, there are two radio button options for visibility: 'Public' (selected) and 'Private'. The 'Public' option is described as 'Anyone on the internet can see this repository. You choose who can commit.' The 'Private' option is described as 'You choose who can see and commit to this repository.' Below these options, there is a section titled 'Initialize this repository with:' which includes a link to 'Skip this step if you're importing an existing repository.' There are three checkboxes: 'Add a README file' (checked), 'Add .gitignore' (unchecked), and 'Choose a license' (unchecked). Each checkbox has a brief description and a link to 'Learn more.'. At the bottom, it states 'This will set `main` as the default branch. Change the default name in your settings.' and a green 'Create repository' button.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \* Repository name \*

fathomdata / git-for-data-scientists ✓

Great repository names are short and memorable. Need inspiration? How about [curly-system?](#)

Description (optional)

Git with Fathom Training

☒ Public  
Anyone on the internet can see this repository. You choose who can commit.

☐ Private  
You choose who can see and commit to this repository.

Initialize this repository with:  
Skip this step if you're importing an existing repository.

☒ Add a README file  
This is where you can write a long description for your project. [Learn more.](#)

☐ Add .gitignore  
Choose which files not to track from a list of templates. [Learn more.](#)

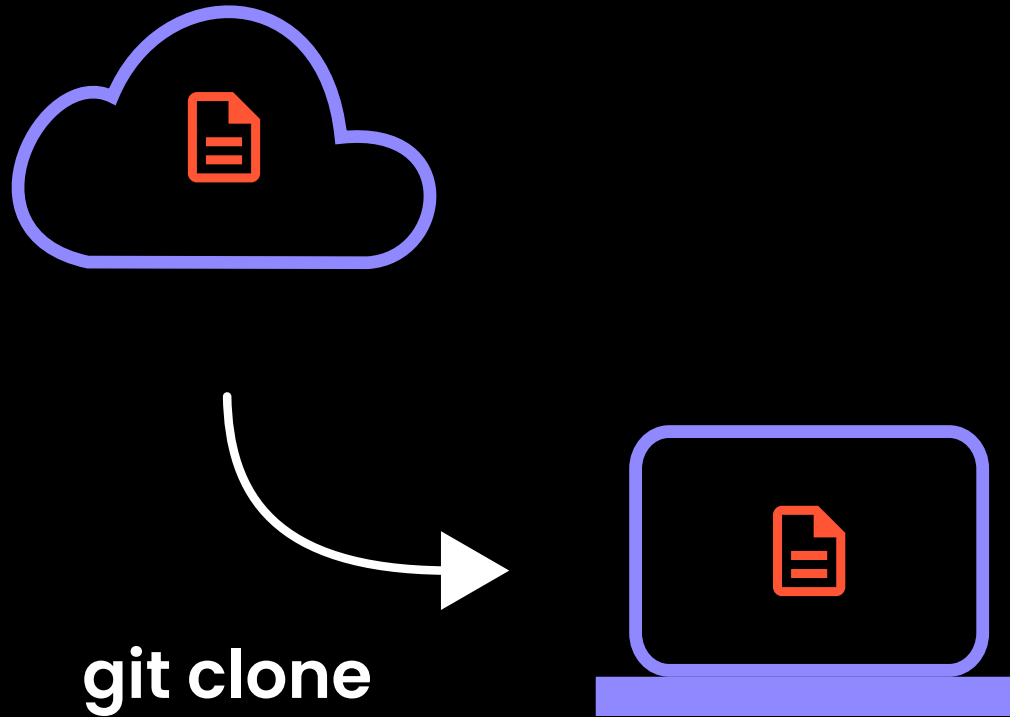
☐ Choose a license  
A license tells others what they can and can't do with your code. [Learn more.](#)

This will set `main` as the default branch. Change the default name in your [settings](#).

Create repository

# Cloning

Cloning a repository pulls down a full copy of all the repository data that GitHub has at that point in time.



# Cloning

The choice of clone link (SSH or HTTP) will determine the type of authorisation that will be required.

Clone your repository on the Desktop:

```
cd Desktop  
git clone {repository URL}
```

For example:

```
git clone git@github.com:fathomdata/git-for-data-scientists.git
```

# Editing the repository

Check the status of our project. But first, direct to the cloned folder:

```
cd git-for-data-scientists  
git status
```

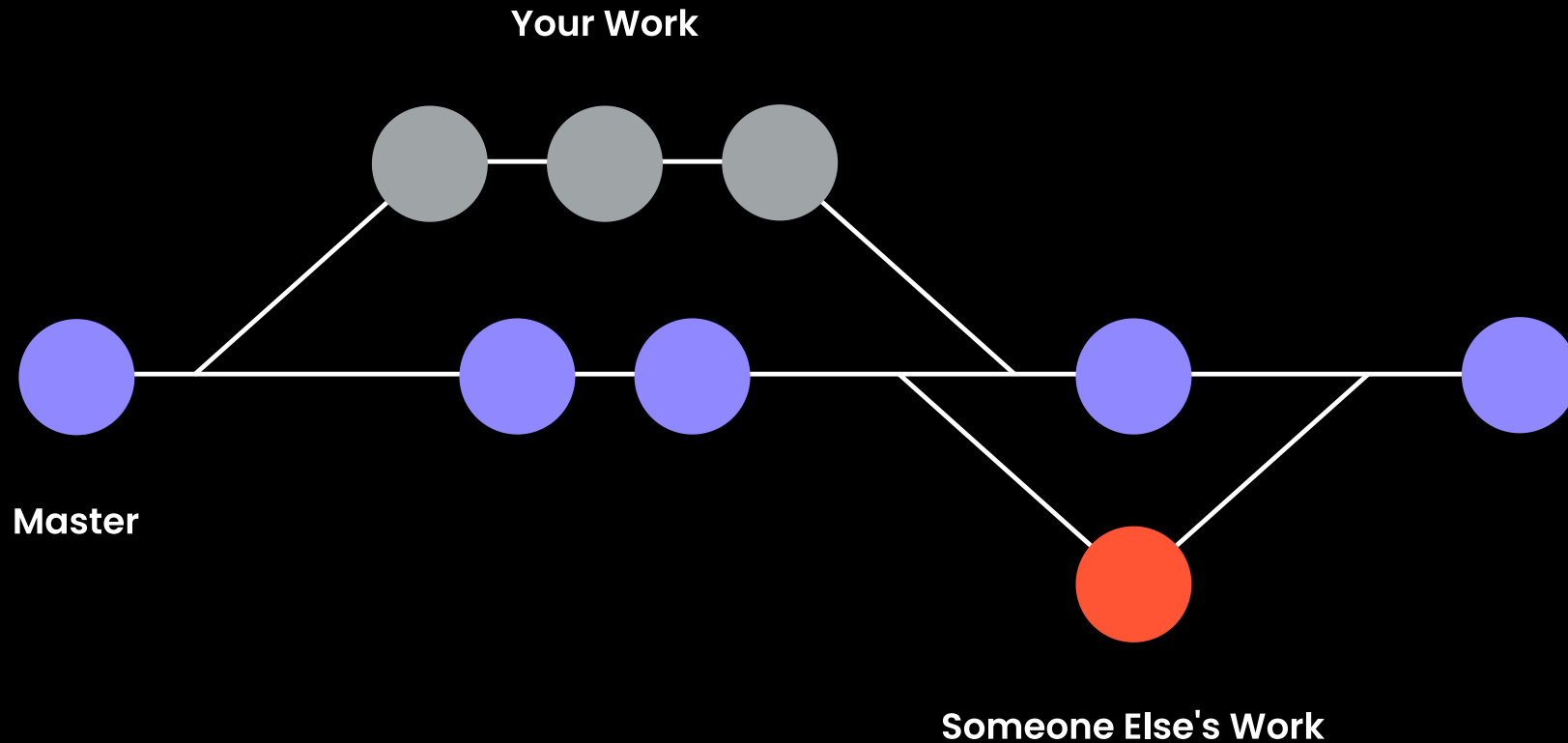
Add a file to the repository and then commit. Send these changes to your remote repository on GitHub:

```
touch demo_file.txt  
git add .  
git commit -m "My first commit"  
git remote -v  
git push origin main
```



# Branches

Branches allow you to develop features, fix bugs, or safely experiment with new ideas in a contained area of your repository.



# Branches

Branching is when you take a detour from the main development stream. It allows a team to work simultaneously without overwriting each other's work.

To create a new branch, run `git checkout -b <my branch name>`:

```
git checkout -b development
```

# Demo:

Create a new text file and commit this on the newly created branch:

```
touch new_file.csv  
git add .  
git commit -m "new_file"  
git push origin development  
  
git log --all  
git log -3  
git log -p  
git log --stat  
git log --oneline
```

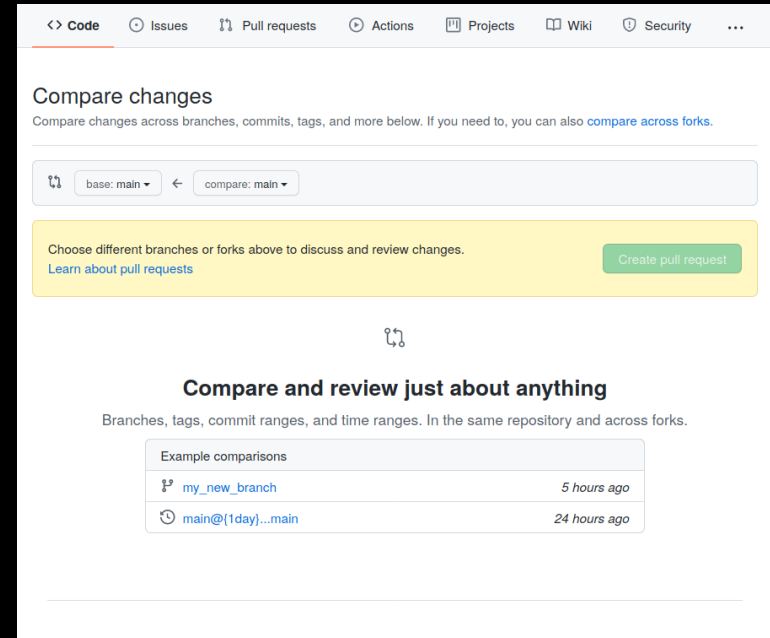


# Merging with pull requests

A pull request is a tool for proposing changes before merging.

To merge changes, you can:

- Click the pull request tab.
- Click the green "New pull request" button.
- Select the "Create pull request".



# Merging in the command line

By default, every Git repo's first branch is named `main`. Let's merge to the main branch:

```
git checkout main  
git branch  
git merge development
```

To create an upstream branch so that you can push all of the changes and set the remote branch upstream, you will push the feature by running:

```
git branch <name of branch>  
git push -u origin <name of branch>
```

# Exercise

1. Create a third branch
2. Add an empty txt file
3. Commit and push to Github



# Merge conflicts

Most of the time a merge will go smoothly. However if the same part of the file is changed on both branches, and differs between the branches, you will experience a merge conflict.

Let's create a merge conflict!

# Merge conflicts

1. Create a new directory called git-merge, change into that directory
2. Initialize it as a new Git repo.
3. Create a new text file (merge.txt) with content
4. Add merge.txt to the repo and commit it

```
cd Desktop
mkdir git-merge
cd git-merge
git init .
echo "Some content to play with" > merge.txt
cat merge.txt
git add merge.txt
git commit -am "committing the initial content"
```



# Merge conflicts

Create a new branch to use as the conflicting merge:

```
git checkout -b development
echo "different content" > merge.txt
cat merge.txt
git commit -am "edited the content to cause a conflict"
```

Next, create a new branch, overwrite the content in merge.txt and commit the new content:

```
git checkout master
echo "content to append" >> merge.txt
cat merge.txt
git commit -am "appended content to merge.txt"
git merge development
```

# Merge conflicts

This now puts our repo in a state where we have 2 new commits:

```
git status
```

The output from Git status indicates that there are unmerged paths due to a conflict. Let's examine the file and see what's modified:

```
cat merge.txt
```

# Merge conflicts

Let's resolve our merge conflict:

```
Some content to mess with  
content to append  
different content
```

Or, edit via the console:

```
vi merge.txt # Make the changes to the file in the terminal  
Esc :wq!
```

To finalise the merge, create a new commit:

```
git add merge.txt  
git commit -m "merged and resolved the conflict in merge.txt"
```

# Commands for resolving conflicts

Identify conflicted files using:

```
git status
```

Produce a log with a list of commits that conflict between the merging branches:

```
git log --merge
```

Exit from the merge process and return the branch to the state before the merge began:

```
git merge --abort
```

Reset conflicted files to a now good state:

# Git rebase

Rebasing a branch updates one branch with another by applying the commits of one branch ontop of the commits of another branch.

- Why rebasing?
  - Useful to integrate recent commits without unnecessary merging
  - Maintain a cleaner, more linear and readable project history

Image a situation where the main branch has changed since you worked on a feature branch. You want to get the latest updates from the main branch into your feature branch and it should appear that you have been working off the latest updated main branch all the time.

# Git interactive rebasing

Take commits from a branch and replay them at the end of another branch

Let's interactively rebase to squash commits, on the command line:

```
cd Desktop
mkdir rebasing
cd rebasing
git init
touch a.txt
git add .
git commit -m "add a.txt"
```

```
fathom-trainer:~$ cd Desktop
fathom-trainer:~/Desktop$ mkdir rebasing
fathom-trainer:~/Desktop$ cd rebasing
fathom-trainer:~/Desktop/rebasing$ git init
Initialized empty Git repository in /home/amieroh/Desktop/rebasing/.git/
fathom-trainer:~/Desktop/rebasing$ touch a.txt
fathom-trainer:~/Desktop/rebasing$ git add .
fathom-trainer:~/Desktop/rebasing$ git commit -m "add a.txt"
[master (root-commit) 6b8213c] add a.txt
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 a.txt
fathom-trainer:~/Desktop/rebasing$
```

# Interactive rebasing

Now, switch to a new branch (feature branch) and create an empty file

```
git checkout -b feature
touch b.txt
git add .
git commit -m "adding b.txt"
# Repeat these steps and create c.txt and d.txt
git log # try other git log commands here
```

# Interactive rebasing

Now, let's initiate rebasing for all commits:

```
git rebase -i master
```



# Interactive rebasing

1. Now that we are in our default editor, remove *pick*, then type *s* (Only for c.txt and d.txt)
2. Save the file: Ctrl + X, y and then Enter
3. Place # in front of the commit comments we don't want to share (the "add c.txt" and "add d.txt")
4. Change the commit message of *add b.txt* to add b, c, and d.txt
5. Save and exit the file: Ctrl + X, y and then Enter

```

pick 8d1ccc8 adding b.txt
s aee4c58 adding c.txt
s 086c3d2 adding d.txt

# Rebase 5f826da..086c3d2 onto 5f826da (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .      create a merge commit using the original merge commit's
# .      message (or the oneline, if no original merge commit was

^G Get Help  ^O Write Out ^W Where Is ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace  ^U Paste Text ^T To Spell  ^_ Go To Line

```

```

fathom-trainer:~/Desktop/rebasing$ git rebase -i master
Successfully rebased and updated refs/heads/feature.
fathom-trainer:~/Desktop/rebasing$ git rebase -i HEAD~3
[detached HEAD 0bb7355] add b, c and d.txt
Date: Mon Oct 18 09:24:39 2021 +0200
3 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 b.txt
create mode 100644 c.txt
create mode 100644 d.txt
Successfully rebased and updated refs/heads/feature.

```

# Interactive rebasing

See a list of commits to confirm:

```
git log --oneline
```

Retrieve files from master branch:

```
git checkout master  
git rebase feature # Initiate rebasing
```

Delete the feature branch:

```
git branch -d feature
```

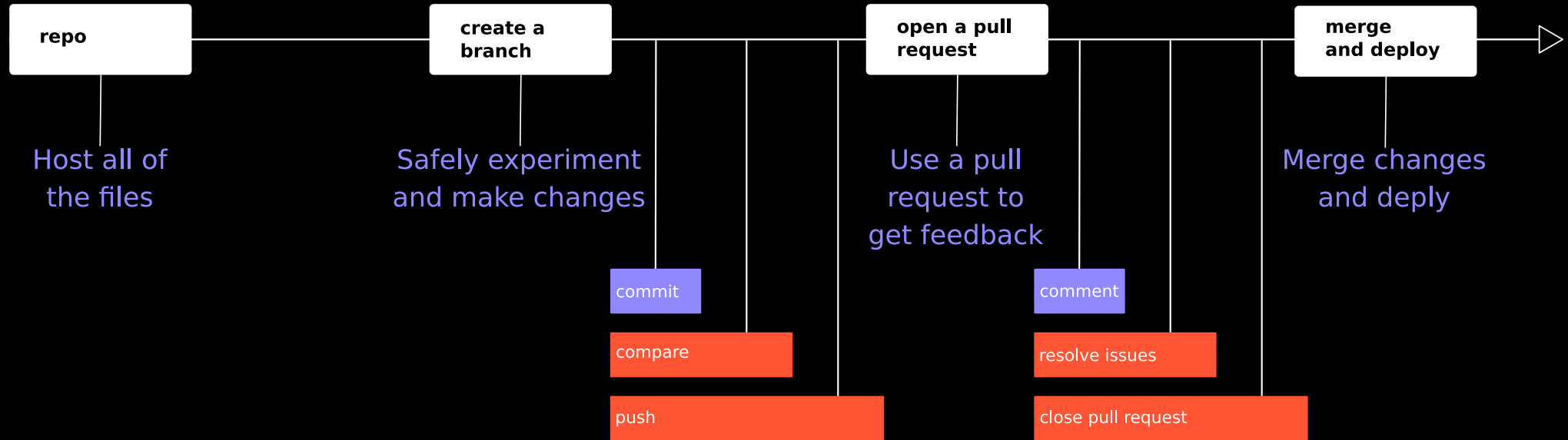
# Interactive rebasing

```
fathom-trainer:~/Desktop/rebasing$ git log --oneline
0bb7355 (HEAD -> feature) add b, c and d.txt
5f826da (master) add a.txt
fathom-trainer:~/Desktop/rebasing$ git checkout master
Switched to branch 'master'
fathom-trainer:~/Desktop/rebasing$ git rebase feature
First, rewinding head to replay your work on top of it...
Fast-forwarded master to feature.
fathom-trainer:~/Desktop/rebasing$ git branch -d feature
Deleted branch feature (was 0bb7355).
```

Incase of worry and panic, use:

```
git rebase --abort
```

# Summary



# Other helpful commands

Need help remembering what command you're supposed to run? To figure out how to use specific commands to clone, type: Press q to return to the terminal

```
git help  
git help clone  
git clone -h # Lists the arguments but doesn't open the man pages
```

Thank you!

