Git Workshop Notes

Snehit Sah

Contents

1 About the book

This book is provided in digital format free of cost. You are free to use the content as long as you follow the CC-BY-SA 4.0 license terms.

This book is open source at flyingcakes/git-workshop-notes. Head to the repo for suggesting corrections or improvements. Alternatively, you can also mail me at hi@snehit.dev.

This book was originally a simple text file I wrote for myself when hosting sessions. I kept adding bits and pieces until it grew larger than what I'd personally call "speaker notes". I decided I should share these notes with the attendees. Using Pandoc, I generated a PDF and shared it.

The book still remained closed source though. Very recently, I realised that a book about Git that I wrote, being closed source doesn't make sense. So I rearranged it, and published to GitHub under the MIT license. The repo contains everything you need to generate a PDF on your system.

1.1 PDF Download

The latest version of the book is available at files.snehit.dev/Git_Workshop_Notes.pdf.

2 Introduction to Git

2.1 Basics

Git is an open source distributed version control system.

VCS (version control system) refers to a system which can keep track of all the code that is added to a project. Often, you want the history of the project to be preserved for various reasons. Say, the latest version of your software has

major bugs, and as a result, you want to roll back some changes to an earlier date. This will only be possible if your code base is version controlled.

Git is also called *distributed* because the code base isn't stored on one server. Instead, each of the developers working on the project can have their own copies of the project. They can make changes independently on their machines, and then request for their changes to be merged back into the original software.

2.2 Why Git?

Git is one of the topics covered in MIT's Missing Semester. This course covers multiple useful topics that are often not formally taught, but they can be very useful to students.

Git is a vast and powerful software, but **you don't need to be an expert at Git.** You just need to know enough commands to maintain your repository and contribute to other repositories. Rest of the stuff, you can learn as you go.

There are two things that I believe are important for open source software programs, and Git helps you realize them.

1. Integrity of history : The project history is an important data that lets users trust the software. Anyone can see how the software developed or when the different components were added. It also lets people track new changes, so that external contributors can focus more on testing new code and also restrict themselves to auditing new code.
2. Long term maintainability : With git, you can associate commits with a descriptive commit message. This lets future maintainers know more about the code. Commits also carry author email - which can be useful in case a maintainer needs to contact a past maintainer to discuss some legacy code.

2.3 My approach to Git

I believe many Git workshops get the direction wrong. Their focus is on GitHub; and git is shown as a side utility. I call that the "easy way". Instructor will create a repository. They'll ask viewers to fork the repo, make some minor change via GitHub web interface, and create pull request. Such workshops end with a "grand message" that the viewer made their first PR. Its cool and all, but it's severely lacking in some important concepts.

I instead prefer bringing Git to the center stage. Git is very powerful, and knowing how to work with the command line utility is going to serve you quite far on your journey. GitHub has a cool self-explanatory UI - do you really need to focus more over learning that as compared to a powerful command line utility?

Another minor complaint with having all focus on GitHub is that new students fail to appreciate the existence of other git repository hosting services. Truth is

that once you learn Git, you can work with any hosting service.

2.4 A note about Git GUI programs

I'm not against GUI programs altogether. Once in a blue moon, even I fall back to using GUI. However, while learning it's better to use command line client only. That way you know what is happening, and it strengthens your understanding.

PS. Recorded lectures from The Missing Semester are free to view. You can check out their website here : https://missing.csail.mit.edu/.

3 Starting with Git

We already know that Git tracks your project. So, every time you change a file and commit it to the project (basically asking Git to mark a milestone you can return to later, if needed), Git is recording the **changes** you made to each file. Note that complete files are not duplicated. Only the changes you made to each file are stored by Git. It is intelligent enough to create the complete file by knowing the changes you made all through.

3.1 Installing

Downloads are offered at git-scm website. Windows users should download installer from the website. Users on *nix systems can use their package manager for the same.

https://git-scm.com/downloads

To check if git is working, open a terminal, and run

```
git --version
```

This should output the version name. An example output, on my system:

```
git version 2.33.0
```

I'm starting with shell commands now. Git commands should work on all systems, but other commands like `echo` will probably work only on *nix based systems. May also in Git Bash on Windows.

3.2 Setting up Git

Git needs some info to create commits. At the very least, it needs to know your name and email. Run the following commands to set them up.

```
git config --global user.name "Your Name"
git config --global user.email "email@domain.com"
```

You can verify them too. I have shown the commands with their outputs.

```
[snehit@wired ~]$ git config --global user.name
Snehit Sah
[snehit@wired ~]$ git config --global user.email
snehitsah@protonmail.com
```

You may also change the default init branch to something other than `master`.

```
git config --global init.defaultBranch main
```

You can replace `main` with any other name you like. This step is not necessary. Do this only if you don't want your default branches in new repositories to be named `master`.

3.3 Initializing a repository

Create a new folder. Change working directory to this new folder. And then initialize a new repository.

```
mkdir my-project
cd my-project
git init
```

Now, you can open your code editor in this folder. Any files you add here will automatically be noticed by git. I run the following command to launch VS Code in my repo.

```
code .
```

3.4 Adding files

You can either create files via a text editor like VS Code, or even the classic notepad, or you can use a terminal utility to create files. I'll use `echo` to create a simple python script.

```
echo "print('Hello World')" > test.py
```

It creates a file `test.py` with the code `print('Hello World')`.

Now, lets check if git notices this file or not. Run `git status` in the same folder, and you should get such an output.

```
[snehit@wired my-project]$ git status
On branch main

No commits yet

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

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

There is quite some information here. The first line tells we are on this branch called `main`. Don't worry if you don't know what branches are. I'll discuss it soon.

Next is a section called `Untracked files`. This lists the file that are there in folder, but git isn't yet tracking them for changes. Our file, `test.py` is listed there. Git is also telling us a command to start tracking them… So lets move on to that.

3.5 Adding files to staging

Run this command to start tracking `test.py`.

```
git add test.py
```

`git add` serves two purposes. If a file is untracked, then it will start tracking it and add to staging area. If a file is already being tracked, but has some changes, then git will simply just add the file to staging area.

What is the staging area, you ask? It's the files whose change will be recorded and stored when you run git commit the next time.

Before we move on to the next command, let's see the output of `git status` after running the above `git add` command.

```
[snehit@wired my-project]$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   test.py
```

Our file `test.py` is now listed under a different section - `Changes to be committed`.

If you have multiple files in a project, you can simply use the following command to add all changed and untracked files.

```
git add -A
```

Be careful while running this. It may add any extra files you created in the repository which you actually don't want in commits.

3.6 Committing changes

Once you have files in the staging area, you `commit`. This is an important step. Note that commits are the mechanism by which git is able to preserve history. You can see commits as milestones, which anyone can check out.

Commits are accompanied by at least a title, and an optional message. Try to keep the message/title short, yet informative. For larger projects, it becomes necessary to have informative commit messages so that reviewers know what the commit does.

Run this in the terminal

```
git commit
```

This should land you in a text editor, usually vi(m), where you add commit title and message. This is what you should see.

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
#
# Initial commit
#
# Changes to be committed:
#   new file:   test.py
#
```

You get the output you saw with `git status` one more time, so that you can be sure that you are committing all the files you want. On the first line, give a commit title. Leave a blank line, and from the third line, you can write the description.

Here is an example. Note that I did not remove the lines that were already there. They are starting with a '#' and git will ignore them.

```
Add test.py with basic code

New beginnings - This file prints
Hello World. Tested with Python3.7
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
#
# Initial commit
#
# Changes to be committed:
#   new file:   test.py
```

```
#
```

Save and close the file. Git should create a commit now, and print some message in the terminal.

```
[snehit@wired my-project]$ git commit
[main (root-commit) 5de911e] Add test.py with basic code
 1 file changed, 1 insertion(+)
 create mode 100644 test.py
```

It's called `root-commit` because it's the first commit in this repo. There is a fancy alphanumeric string following it. `5de911e` - these are the first few characters of the commit hash. Commit hash is a 20 character alphanumeric SHA1 hash that is generated using data from past commit, changes in current commit, timestamp, author info etc. These hashes are git's primary mechanism to detect any changes to past code.

If you don't want to add a description to your commit - maybe because the change is very minor and self-explanatory - you can use the shorthand to just add a commit title.

```
git commit -m "Fix typo in code"
```

3.7 Checking logs

Since we have made a commit, it's a permanent part of the repository now. You can see logs to verify the commit is there.

```
[snehit@wired my-project]$ git log
commit 5de911e18db087d3c3a57031dab49637e316608f (HEAD -> main)
Author: Snehit Sah <snehitsah@protonmail.com>
Date:   Thu Sep 9 12:12:00 2021 +0530

    Add test.py with basic code

    New beginnings - This file prints
    Hello World. Tested with Python3.7
```

This gives us information about the commit we just made. As we make more commits, the log will grow lager. In the first line, you have the complete commit hash. (in the last section, we only saw first 7 characters). We also have information about the author, commit date/time and message. If you are not interested in author info, and just want to see commit messages, we can use the shorter format with `git log --oneline`

```
[snehit@wired my-project]$ git log --oneline
5de911e (HEAD -> main) Add test.py with basic code
```

This is more useful to know how the project grew, since it omits author info, dates and commit description.

3.8 Amending last commit

Sometimes it happens that you forgot to add a file to staging before commit, or there's a tiny change you need to make, which should have been a part of the last commit. In such situations, git provides an easy way to amend last commit. Assuming you made a commit, do some edits to code. Then do `git add` as usual. While committing, include the `--amend` flag to instruct git to update the last commit instead of creating a new one.

```
git commit --amend
```

If you look at the log, you will see the commit hash changes. This signifies that *some* changes were made. Later on when you upload your code on the internet, these hashes will allow other collaborators ensure they have the same code as you. If you edit a past comment and upload it, then other collaborators will get to know about it. In general, it's not a good idea to amend/edit commits if you have published them onto the web. Do this only if the commits to be edited are only on your local machine. One exception is that when you accidentally commit and publish sensitive info, like access keys, password etc.

PS. In case you are wondering if other collaborators will match commit hash letter by letter to verify integrity - the answer is an obvious "no!". Git does that automatically.

3.9 Recap

Try creating another file, and make some changes to `test.py`. Check status. Add both files to staging, and then make a commit. Check log to verify the commit was made.

You can use any text editor for the task. I'll quickly give the relevant shell commands for those who want to follow along in the shell.

```
echo "print('New file')" > new_file.py
echo "print('Another line')" >> test.py
```

Output of `git status` at this stage.

```
[snehit@wired my-project]$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   test.py

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

no changes added to commit (use "git add" and/or "git commit -a")
```

Continuing on to add files and commit them.

```
git add new_file.py
git add test.py
git commit -m "Update test.py and add new code"
```

Output of `git log`

```
[snehit@wired my-project]$ git log
commit 9c804c05610ca86f720abda62b297e9d2e6a725f (HEAD -> main)
Author: Snehit Sah <snehitsah@protonmail.com>
Date:   Thu Sep 9 12:34:49 2021 +0530

    Update test.py and add new code

commit 5de911e18db087d3c3a57031dab49637e316608f
Author: Snehit Sah <snehitsah@protonmail.com>
Date:   Thu Sep 9 12:12:00 2021 +0530

    Add test.py with basic code

    New beginnings - This file prints
    Hello World. Tested with Python3.7
```

3.10 Using gitignore file

You may have some files in your project directory which you don't want git to track. For example, upon testing C/C++ apps, you have a binary that shouldn't be there with the code. To prevent them from accidentally being committed, you can add them to gitignore file. You'll first have to create a file named `.gitignore` in the root of your repo folder. Then you list the file paths that should not be tracked.

```
echo "*/a.out" >> .gitignore
```

This will prevent git from tracking any file with name `a.out`. The asterisk at beginning is a wildcard that will select any path where `a.out` exists. To add folders, append a forward slash after path.

```
echo "build/" >> .gitignore
```

4 Working with remotes

Up until now, we have been working on our local system. Git however is built to facilitate collaboration of multiple developers. To let other developers work on our code, we need to publish this code to a website.

Now, you can upload your code anywhere. Zip it up and put on cloud storage like Google Drive or Dropbox. Or send the files as an attachment to your friend.

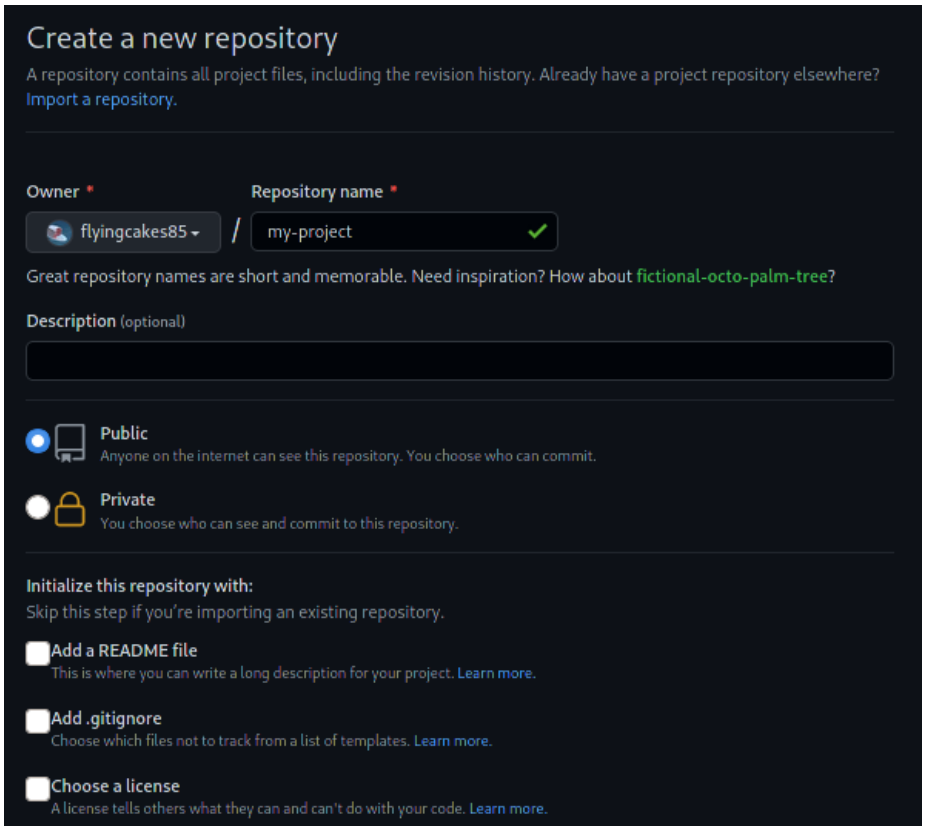
There are many ways, but they won't preserve the commit history and messages (unless you include the hidden `.git` folder). Moreover, it will be difficult to use collaboration features of git.

It is much better to use a hosting service that is specially made for Git repositories. One of them is GitHub and that's the one which we will be using for this tutorial.

Head over to their sign-up page and create an account if not done already. https://github.com/join

4.1 Connecting a remote repository

Create a new GitHub repository at https://github.com/new. Give it a name. Description is optional. Make sure that all checkboxes under initialization are unchecked.



Create the repository. On the page you are redirected to, you have an HTTPS link under *Quick setup*.

Link is of the form `https://github.com/<username>/<repo-name>.git`.

Copy this link.

In your terminal, you will now tell git to use this remote link for publishing on the internet. You use the `git remote add` command to do this.

```
git remote add origin https://github.com/flyingcakes85/my-project.git
```

Here, `origin` specifies the remote name. You can give it any name. `origin` is a common name for the primary remote location. Make sure you use your repo link in the command.

Before you push your branch, you first need an access token. Think of it as an alternate password to your account. For security reasons, using your actual password is not supported on GitHub. SSH is the easier way, but setting it up is out of the scope of this tutorial.

Head over to this link : https://github.com/settings/tokens. Click on "Generate new token". It may ask your password again.

Add a note to identify the token later. Check the first box which says "repo". It should automatically check the 5 boxes under it. Scroll down and click "Generate token". You will be redirected back to the token page where you can see your newly created token. Copy it to a file and store it somewhere you can access. While pushing form the terminal, you will use this token instead of your password.

Finally, you can push your branch to the remote.

```
git push -u origin main
```

It will ask your username and password. Enter your GitHub username only (not the complete link) and for password, use the token we just generated. Now on in this tutorial, I won't mention that you have to use this token, so keep it in mind.

`-u` is a shorthand for `--set-upstream`. It tells git that `origin` is the upstream for `main` i.e. the remote location where the `main` branch should be published. You might be wondering why we need to tell this? Git can work with multiple remotes, and often, different branches need to be published to different remotes. For any future push on this branch, you don't need to mention upstream, and simply running `git push` should do the work for you.

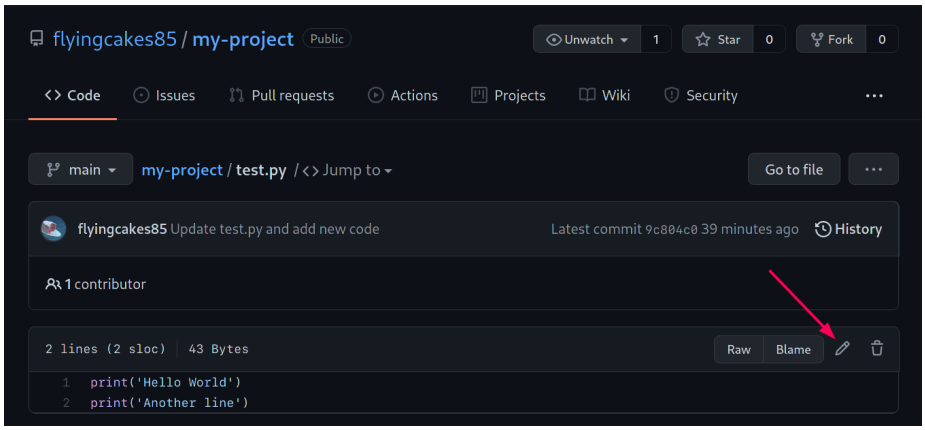
4.2 Pulling remote changes

Say someone else contributed to your repository. The changes they contributed will be stored on GitHub (or whichever remote you are using). You need to get the changes down on your machine. `git pull` is the command we use to get changes from remote to our local system.

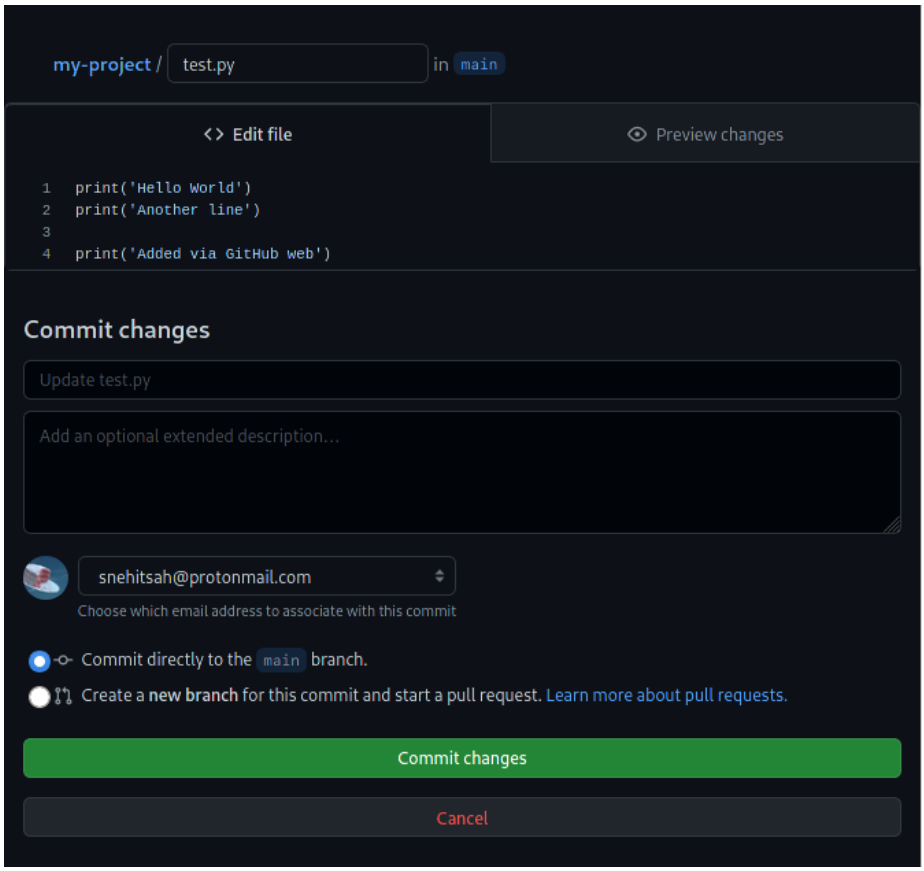
Since we may not be able to find a contributor right away, so let us make changes ourselves on the remote. GitHub web interface allows making changes to code

without needing to have repository on your local system. Head over to the repository on your GitHub account.

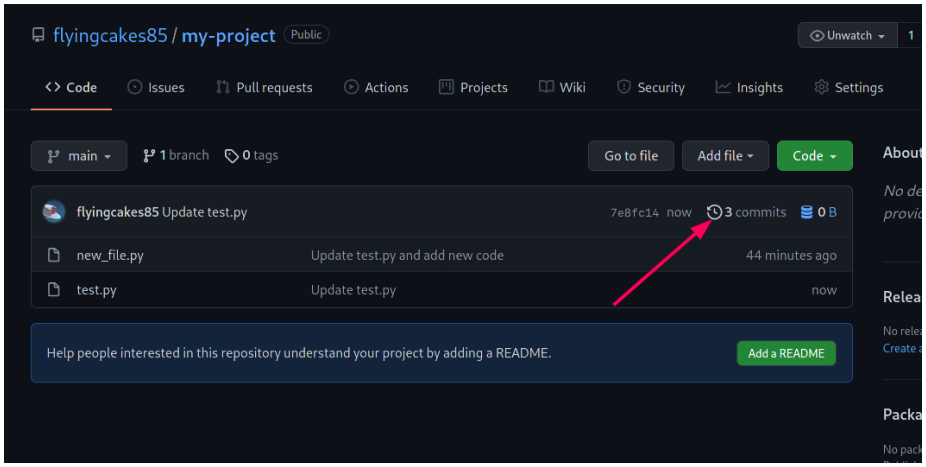
You will be able to see the files you added - `test.py` and `new_file.py`. Click on any of them, let's say `test.py`. You will see the code you wrote. On top right of the code, you can see a pencil icon that lets you make changes to file.



Add a some code and click "Commit changes".



Now, if you go to repository page, and click on the number of commits (see screen-grab below), you can see 3 commits listed. We made 2 commits earlier. The third commit was made just now via GitHub web.



Go back to the terminal and run `git log --oneline`. You will still see only 2 commits here. This mean we need to pull remote commits on to our local machine. Run `git pull` to do the same. Here is the output in my case

```
[snehit@wired my-project]$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 709 bytes | 35.00 KiB/s, done.
From https://github.com/flyingcakes85/my-project
   9c804c0..7e8fc14  main       -> origin/main
Updating 9c804c0..7e8fc14
Fast-forward
 test.py | 2 ++
 1 file changed, 2 insertions(+)
```

You can see the text `test.py | 2 ++`. This mean that there were 2 additions made to this file. Deletions are followed by `--`. Now, if you run `git log --oneline` in terminal, you can see third commit too.

```
[snehit@wired my-project]$ git log --oneline
7e8fc14 (HEAD -> main, origin/main) Update test.py
9c804c0 Update test.py and add new code
5de911e Add test.py with basic code
```

4.3 Listing remotes

You can use `git remote` to list remote names configured for current repo, or pass the `-v` flag to also list remote links.

```
[snehit@wired my-project]$ git remote
origin
[snehit@wired my-project]$ git remote -v
origin  https://github.com/flyingcakes85/my-project.git (fetch)
origin  https://github.com/flyingcakes85/my-project.git (push)
```

4.4 Using git clone

So far we created a repository on the command line and connected it to a remote. Many times, we already have a git repo online which we want to use on our local machine. You can simply use the `git clone` command to download a repository. Note that you can download any public repository with this command even if you did not create it.

```
git clone https://github.com/flyingcakes85/my-project.git
```

Often, the repository you want to download is large, and downloading it can take some time. To get around this, you can use `--depth=N` flag to fetch only

the past `N` commits. You will have all files from the repository, but you won't have complete commit history.

4.5 Editing published commits

Make some changes to a file and instead of a new commit, use `--amend` to edit the last commit.

```
echo "print('More content')" >> new_file.py
git add new_file.py
git commit --amend
```

Now, if you do a `git push`, you will get an error.

```
[snehit@wired my-project]$ git push
Username for 'https://github.com': flyingcakes85
Password for 'https://flyingcakes85@github.com':

To https://github.com/flyingcakes85/my-project.git
 ! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/flyingcakes85/my-project.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

There are two ways to get out of this situation - you either do a `git pull --rebase` to merge remote changes to your local copy first. Or you can do `git push -f` to force overwrite remote with your code.

If you were to do the latter, you will be changing the history of project. Anyone who has a copy of the project and does `git pull` will get a similar error.

The above error is also a common error when you made some change on your remote (via web interface) and then you made some commits on your local machine without pulling remote changes first. As I told, `git pull --rebase` should get you out of the fix in most cases.

Keep in mind, editing published commits is not recommended, and you should do this only when necessary.

5 Working with Branches

Git provides the ability to create branches to let developers work on new features or bug fix without disturbing the main code. Changes you make to a branch are stored separately from the other branches. You can switch between branches anytime. When you think you have made enough changes to a branch and the changes are working, you can merge the branch into your `main` branch.

5.1 Listing branches

`git branch` will show you your local branches, and adding the `-a` flag will also list remote branches.

```
[snehit@wired my-project]$ git branch
* main
[snehit@wired my-project]$ git branch -a
* main
  remotes/origin/main
```

The branch marked with an asterisk denotes the currently active branch. Any changes or commits you make are added on the currently active branch.

5.2 Creating and switching branches

`git branch <name>` creates a new branch. `git checkout <name>` will switch to the new branch so that further changes are make to the new branch.

```
git branch add-emotes
git checkout add-emotes
```

You can also use the shorthand to create and switch branch in one command. Just use the `checkout` command with a `-b` flag.

```
[snehit@wired my-project]$ git checkout -b add-emotes
Switched to a new branch 'add-emotes'
```

You can also *checkout* an older commit. This will replace the project files with the content that existed at that commit. Run `git log --oneline` to see the commit hashes.

```
[snehit@wired my-project]$ git log --oneline
f15f7dc (HEAD -> bugfix, origin/main, main, add-emotes) Update test.py
7e8fc14 Update test.py
9c804c0 Update test.py and add new code
5de911e Add test.py with basic code
```

Note the hash of whichever commit you want to checkout. Say, we want to go to the first commit. Its hash is `5de911e`.

```
git checkout 5de911e
```

This will land you to the first commit. If you see the project now, you will find only one file. Don't worry, our changes aren't lost. We can just switch to `main` branch to get our changes back.

```
git checkout main
```

5.3 Working on branches - merging and stash

You usually create a branch to work on bug or feature without disturbing the actual code. The benefit here is that if there are any urgent changes required to the main code, you can make commits on main without needing to halt working on bug or feature.

There are usually three situations.

5.3.1 Fast-forward merge

First checkout the newly created `add-emotes` branch.

```
git checkout add-emotes
```

Let's make some changes now.

```
echo "print(':-)')" >> emotes.py
git add emotes.py
git commit -m "Add emotes"
```

You test the code. It works and you are satisfied. So you decide to merge it into main. First checkout `main` branch then merge the other branch into it.

```
[snehit@wired my-project]$ git checkout main
Switched to branch 'main'
[snehit@wired my-project]$ git merge add-emotes
Updating f15f7dc..e74906b
Fast-forward
 emotes.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 emotes.py
```

Notice the phrase `Fast-forward`. It signifies that no commits were made on main from the time of creating branch `add-emotes` and the time when we merged it. Git could just "fast-forward" the `main` branch with commits from `add-emotes`. You ran use `git log --oneline` to verify that the commit was indeed applied on `main` branch too.

5.3.2 Recursive merge

Let's do this process again, but we will make a commit on `main` before merging. Again, checkout `add-emotes` and make some changes.

```
git checkout add-emotes
echo "print(':/')" >> emotes.py
```

Let's say, you don't want to commit now, as you want to make more changes. But there's an important fix to make on the `main` branch. You can go forward and switch branch, but it can become messy, because the changes will also show

up on main (they are uncommitted) and it will get confusing for you. So, you store the changes to a temporary directory and then make changes on main.

```
git stash
```

Upon running this, you will notice that the line we added to `emotes.py` is no longer there. Now make changes on main and commit.

```
git checkout main
echo "print('important bugfix')" >> test.py
git add test.py
git commit -m "Important Bugfix"
```

Now, let's switch back to `add-emotes` branch and complete our code. We use the `git stash pop` command to bring back the changes we had made earlier.

```
git checkout add-emotes
git stash pop
echo "print('B-)')" >> emotes.py
git add emotes.py
git commit -S -m "Add more emotes"
```

Now, let's switch back to `main` branch and try merging `add-emotes`.

```
git checkout main
git merge add-emotes
```

This will land you in your text editor, with the following text.

```
Merge branch 'add-emotes'
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

This is very much like making a commit. There was already a commit made on to `main` before we merged the branch. So, git cannot simply fast-forward. Save this file and exit. You should get this message.

```
Merge made by the 'recursive' strategy.
 emotes.py | 2 ++
 1 file changed, 2 insertions(+)
```

This time, it was not fast forwarded. Git used a different strategy called recursive.

5.3.3 Merge conflict

Finally, if you change the same line of the same file in both branch then try to merge them, you run into a merge conflict. Let's quickly add different text to the same file on both `main` branch and `add-emotes` branch.

```
git checkout main
echo "print('added via main')" >> test.py
git add test.py
git commit -m "Make program better"
git checkout add-emotes
echo "print('added via add-emotes')" >> test.py
git add test.py
git commit -m "Make program fancy"
```

If you see the contents of `test.py`, you won't see the line `print('added via main')`. If you checkout `main`, then you won't see `print('added via add-emotes')`. So, what we have got is different content at same location in two branches. This will lead to a conflict.

First checkout `main` branch.

```
git checkout main
```

Let's now merge it and see what happens.

```
[snehit@wired my-project]$ git merge add-emotes
Auto-merging test.py
CONFLICT (content): Merge conflict in test.py
Automatic merge failed; fix conflicts and then commit the result.
```

Git bailed out saying it cannot merge - as we expected.

When you get a merge conflict, you shouldn't need to panic. It's not an "error". It only indicates that git is unable to decide which copy of the code to keep, because the two branches have different changes for the same file location.

Git is telling us that there is a conflict in `test.py`. Let's open that file and see what's there.

```
print('Hello World')
print('Another line')


print('Added via GitHub web')
<<<<<<< HEAD
print('important bugfix')
print('added via main')
=======
print('added via add-emotes')
>>>>>>> add-emotes
```

Here, you can see the conflicting part between angled brackets. From `<<<<<<< HEAD` to `=======` are the contents that are in `main`. From `=======` to `>>>>>>> add-emotes` are the contents that are in `add-emotes`. You can freely remove one of them, along with the angled brackets and equal signs. You may also just remove the brackets and equal signs in case you want to keep changes from both

branches. Finally, you may remove it to add some other code altogether. It's all up to you.

Let's say, I remove the code from `main`. So my file should now look like this.

```
print('Hello World')
print('Another line')

print('Added via GitHub web')

print('added via add-emotes')
```

Now we can continue on to adding file and committing it.

```
git add test.py
git commit -m "Merge add-emotes into main"
```

This will create the merge commit.

5.4 Deleting branches

You may want to delete a branch if you have merged it into `main`, and it is no longer needed. Or maybe you decided against incorporating the changes from that branch. In these cases, you can delete the branch.

```
git branch -d <name>
```

If you had published a branch you now want to delete, run this command

```
git push origin --delete <branch name>
```

Run these commands with care, as you may accidentally delete one of the important branches.

6 Beyond the book

6.1 General development workflow

I'll briefly tell the workflow you follow when contributing to someone else's repo. You first fork the repository to your account. You clone your fork with `git clone <link>`. Likewise, you will then create a new branch which shall hold the commits you make. Furthermore, you can also directly make commits on `main`, but this is almost **never** recommended.

You make some changes then commit it. When you think you have made all the changes you wanted, you push the branch to remote on your repo, using `git push -u origin <branch name>`. On GitHub, you finally create a pull request from your branch to the `main` or `dev` branch of original repo.

Maintainers will often have a separate `dev` branch. This can be for various reasons - the simplest one being that they don't want everyone to be using the

latest changes by default as it can be buggy and unstable. Those who want to test new features can manually switch to dev branch and use. Once features from dev branch are tested, it is finally merged into `main`.

If available, read the `CONTRIBUTING.md` file on the original repo.

6.2 Referencing GitHub pull requests/issues

Say your pull request fixes a certain issue open on the original repository. You can add `Fixes #N` to your commit description. Here `N` is a number that shows just after the issue title. The numbers also show up on pull request titles. You can use the same `#N` format to reference a certain pull request or issue while adding commits on GitHub.