Git and Github - More on Git Commands

LESSON

Generate a quiz about this unit

Summarize this unit

Have any doubt about this content? Ask!

Learning goals

After this lesson, you will be able to:

- Initialize Git in your project,
- Check the status to see what has changed,
- Add your changed files to the staging area with the `git add` command,
- Take a snapshot of changes you have made using `git commit`,
- View previous commits you have made using `git log`.
- Explain what GitHub is and what it is used for,
- Explain what a *remote repository* is,
- Push and pull code between your *local* and *remote* repositories.

Part 1: Git

Creating a Repository

Let's start by creating a directory to play with Git. Navigate to the desktop and create a folder called *git-practice*:

```
# We assume you are already in the desktop folder

mkdir git-practice

cd git-practice

touch data.txt

code .
```

Copy

Explain this code

Add some text to `data.txt`. Feel free to change the data:

```
Name: Ironhacker
Age: 25
Favorite Color: Yellow
```

Copy

Explain this code

`git init` and the hidden `.git` folder

We can use Git to keep track of changes in files and folders on your local machine, but how is this done?

Don't assume this happens all the time since that is not the case. To be able to track the changes you made in your file, the first thing you have to do is **to initialize this folder as a git repository**. *We will explain this.*

Inside the `git-practice` folder, run the following command:

```
git init
```

Copy

Explain this code

You should have `git` installed on your computer. You can check it out by running the following command in the terminal:

```
git --version
```

Copy

Explain this code

The output should be your git version, something like for example `git version 2.20.1`. In general, it's still fine if you see other version numbers that are bigger or smaller than this one. Depending on your operating system, the version number that is compatible with your system may be slightly different. If you can run the command above and see the version number, Git is installed on your system.

`git init` is the command to signal to Git that the folder you are currently in will now be a Git repository. From this point forward, Git will track all changes to the files and folders inside that folder.

However, remains the same question - how does it keep track of these changes?

Let's run an `ls` command to show hidden folders and files inside of the folder:

```
ls -a

# => .         ..         .git      data.txt
```

Copy

Explain this code

❗ `# =>` denotes a return value. You do not have to type this into your terminal.

In addition to data.txt, we have a new folder called `.git`. We did not create this directory ourselves - Git did when we ran the `git init` command.

The `.git` folder is where Git keeps track of all of the changes you make and *much* more. There is no point in going into the specific details of the folder now. However, we will reference it as the lesson progresses.

💡 If for some reason you run `git init` in a folder you didn't intend to make a git repository, you can simply remove the `.git` folder using `rm -rf`:

```
git init # => to initialize folder as Git repository

rm -rf .git # => to remove git (tracking)
```

Copy

Explain this code

Be careful, though. `rm -rf` is a dangerous command that can erase everything on your system if misused.

Where to create a `.git` folder/ where to run `git init`?

(*We will use the naming git repo/repository, but you can see it the same as git folder/directory. Repo/repository is more common though.*)

It may be easier to begin this by telling you where to **not create it** and why.

Most of the time, you will want to create the Git repository in a specific project folder.

You do not want to create a Git repository in a high-level folder, such as `Documents`, or your home(`~`) directory. Why? Git keeps track of the folder and *all subfolders of that folder*. This means that if you create a Git repository in your home(`~`) directory, you will be tracking all of the changes to files and folders on your local user's computer.

This is bad because:

- it is unnecessary;
- when you go to create a Git repo later on inside of a project, you may run into issues;
- when you eventually want to store your Git repos on GitHub later, you may accidentally push sensitive information. 👎

At Ironhack, you will be creating *many* projects and folders to track with Git. Take a look at the following tree to give you an example of where you should be creating Git repos:

```
├── project-1
│   ├── index.html
│   └── style.css
├── project-2
│   ├── index.html
│   └── style.css
├── project-3
│   ├── index.html
│   └── style.css
├── project-4
│   ├── index.html
│   └── style.css
└── project-5
    ├── index.html
    └── style.css
```

In this folder, we have five projects, all with an *index.html* and *style.css*. We suggest creating your git repository **in each project**, as so:

```
├── project-1
│   ├── .git
│   ├── index.html
│   └── style.css
├── project-2
│   ├── .git
│   ├── index.html
│   └── style.css
├── project-3
│   ├── .git
│   ├── index.html
│   └── style.css
├── project-4
│   ├── .git
│   ├── index.html
│   └── style.css
└── project-5
    ├── .git
    ├── index.html
    └── style.css
```

The following are examples of how *not* to structure the directory:
❌

```
├── .git
├── project-1
│   ├── index.html
│   └── style.css
├── project-2
│   ├── index.html
│   └── style.css
├── project-3
│   ├── index.html
│   └── style.css
├── project-4
│   ├── index.html
│   └── style.css
└── project-5
    ├── index.html
    └── style.css
```

Copy

Explain this code

❌

```
├── .git
├── ironhack
│   ├── .DS_Store
│   ├── project-1
│   │   ├── index.html
│   │   └── style.css
│   ├── project-2
│   │   ├── index.html
│   │   └── style.css
│   ├── project-3
│   │   ├── index.html
│   │   └── style.css
│   ├── project-4
│   │   ├── index.html
│   │   └── style.css
│   └── project-5
│       ├── index.html
│       └── style.css
└── prework
    └── git-practice
        ├── .git
        └── data.txt
```

Copy

Explain this code

Viewing Changes: `git status`

Let's return to our `git-practice` folder. We have already run `git init` and Git is keeping track of our changes. We currently have one file in the folder - *data.txt.*

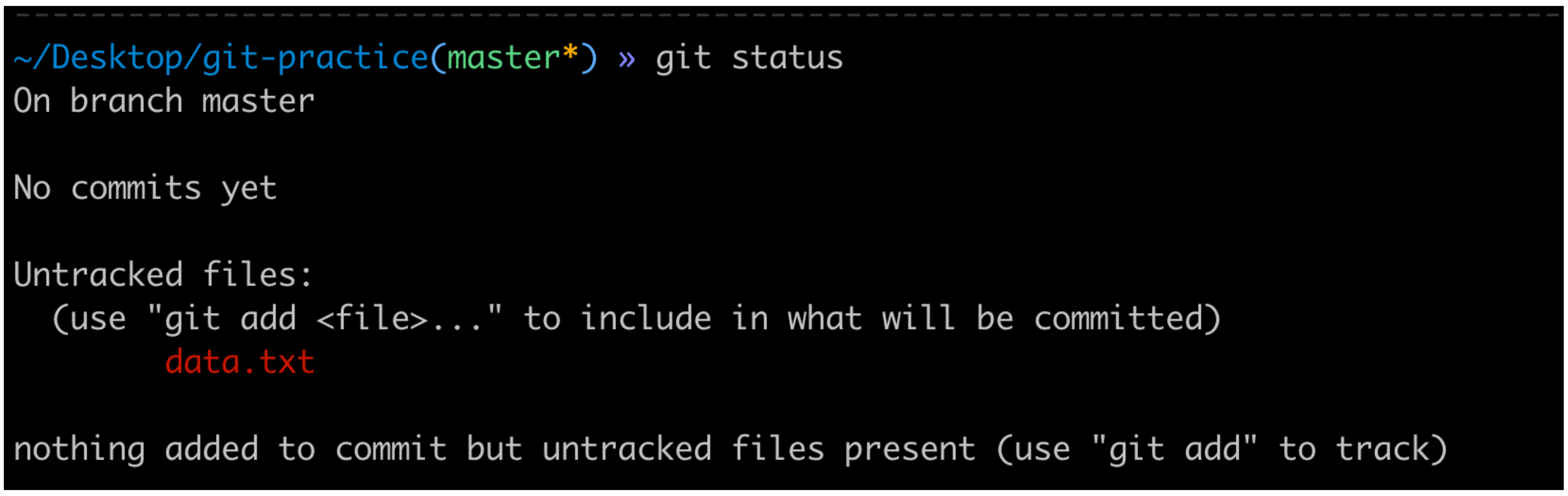
How can we tell what Git is keeping track of? The answer is a command that you will use very often while working on projects: **`git status`**.

`git status` tells us what files and folders are being tracked and what their current status is according to Git.

```
git status
```

Copy

Explain this code



```
---------------------------------------------------------------------------------
~/Desktop/git-practice(master*) » git status
On branch master

No commits yet

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

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

Git is telling us that we should pay attention to right now is the one that says **Untracked files**. `data.txt` is *untracked* because we have created it on our file system, but have not told Git to watch it yet.

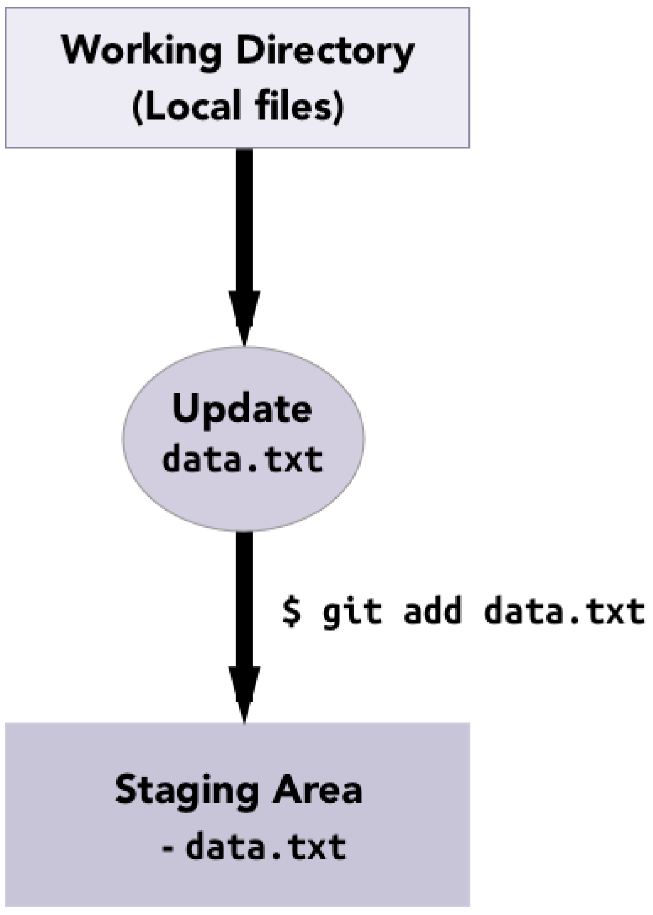
We will return to and reference Git status as we continue the lesson, but let's figure out how to add a file for Git to track.

Staging Changes: `git add`

Staging a Single File

Currently, `data.txt` is on our file system, but not being tracked by Git. Files and folders in this state are referred to as our **working directory**. To track these files with Git, we must *add* them to our *staging area*.

> A staging area is a place where we have notified Git that something will be tracked.

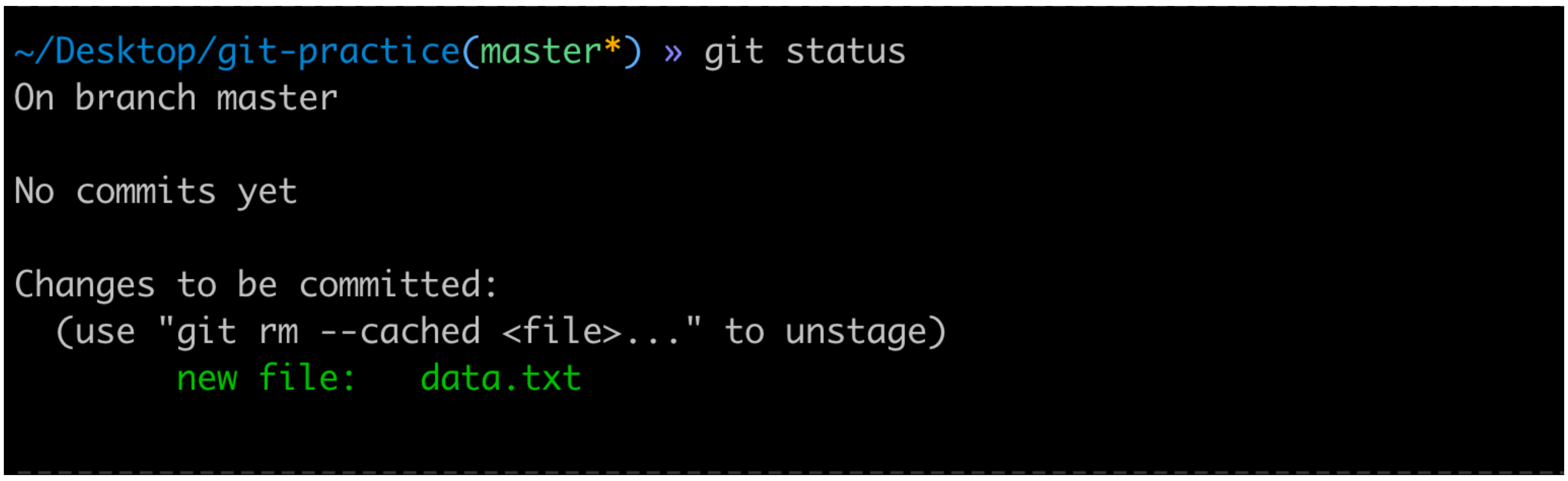


To add a file to the staging area, let's use the `git add` command:

```
git add data.txt
git status
```

Copy

Explain this code



```
~/Desktop/git-practice(master*) » git status
On branch master

No commits yet

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

Now instead of saying “Untracked Files”, Git is telling us that `data.txt` is staged and in the section, *changes to be committed.*

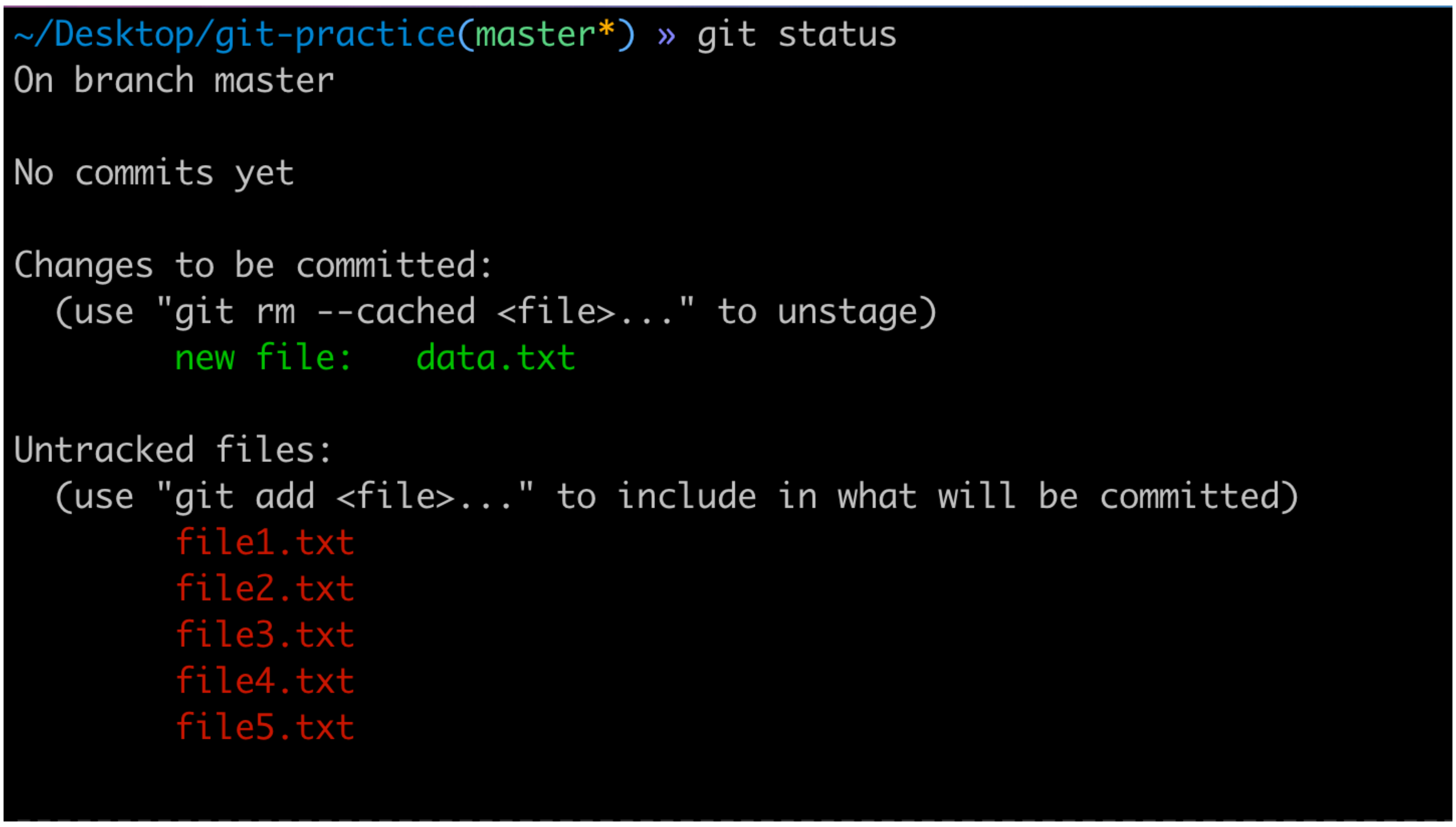
We will discuss committing shortly, but for now, we can see that Git is now tracking our file.

Staging Multiple Files

Often in projects, we will want to stage multiple files at the same time. This can be done in a few different ways.

Let’s add some new files to play around with this:

```
touch file1.txt file2.txt file3.txt file4.txt file5.txt

git status
```

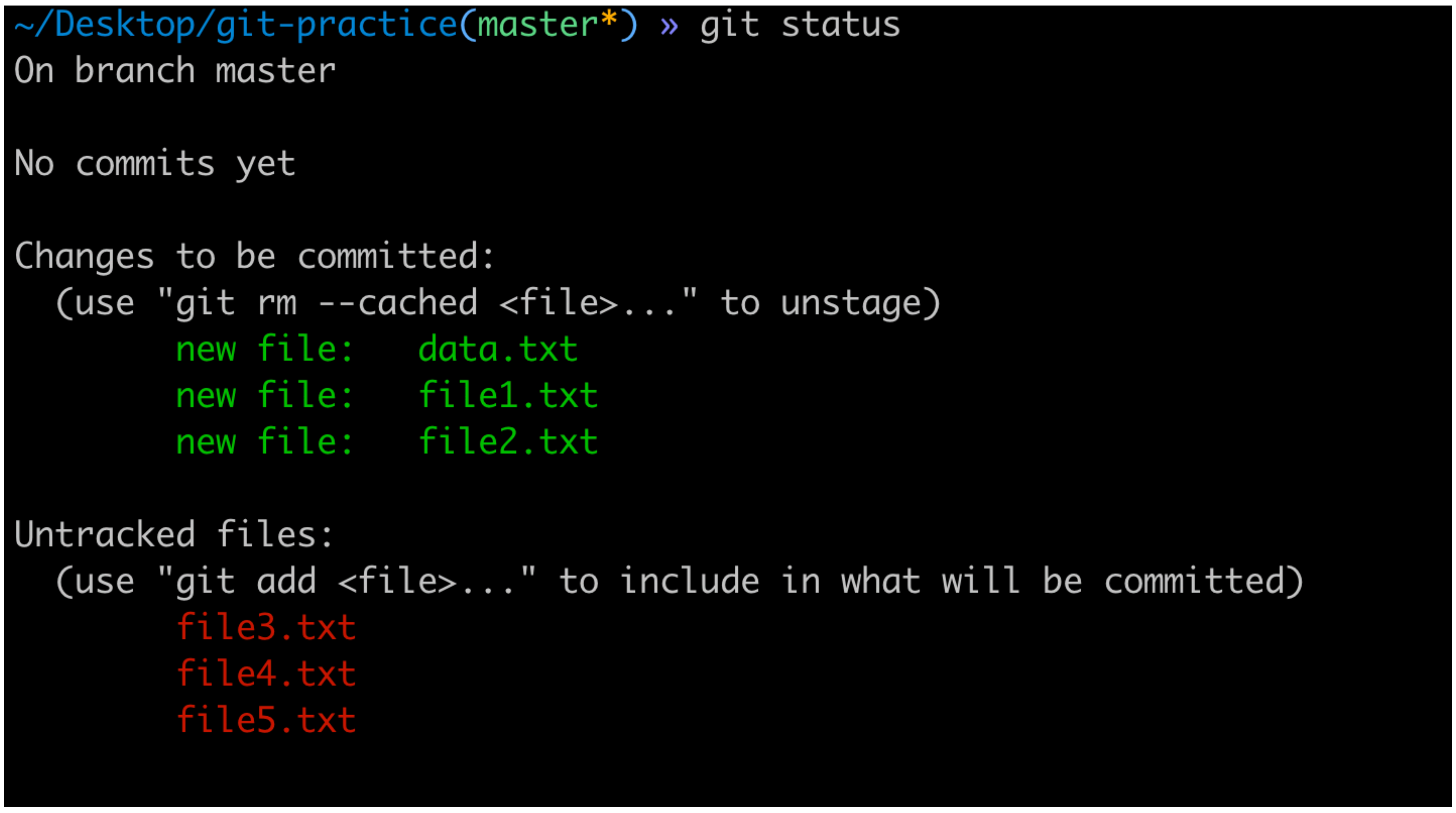


```
~/Desktop/git-practice(master*) » git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   data.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt
        file2.txt
        file3.txt
        file4.txt
        file5.txt
```

Currently, `data.txt` is *staged*, but the other five files are not.

Let’s stage `file1.txt` and `file2.txt`.

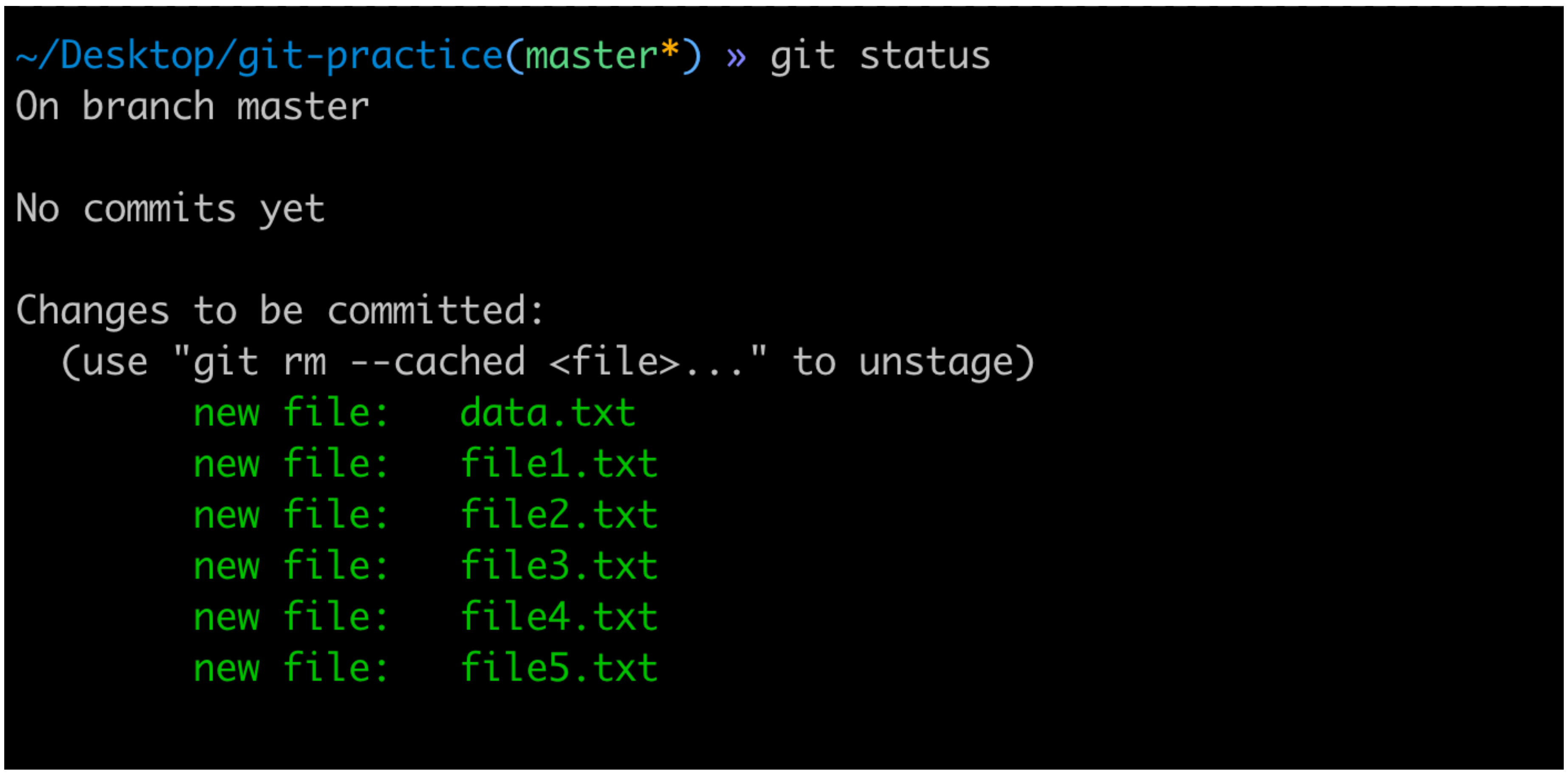
```
git add file1.txt file2.txt

git status
```



```
~/Desktop/git-practice(master*) » git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   data.txt
        new file:   file1.txt
        new file:   file2.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file3.txt
        file4.txt
        file5.txt
```

On second thought, I would like to stage all of the files in my project:

```
git add .

git status
```



```
~/Desktop/git-practice(master*) » git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   data.txt
        new file:   file1.txt
        new file:   file2.txt
        new file:   file3.txt
        new file:   file4.txt
        new file:   file5.txt
```

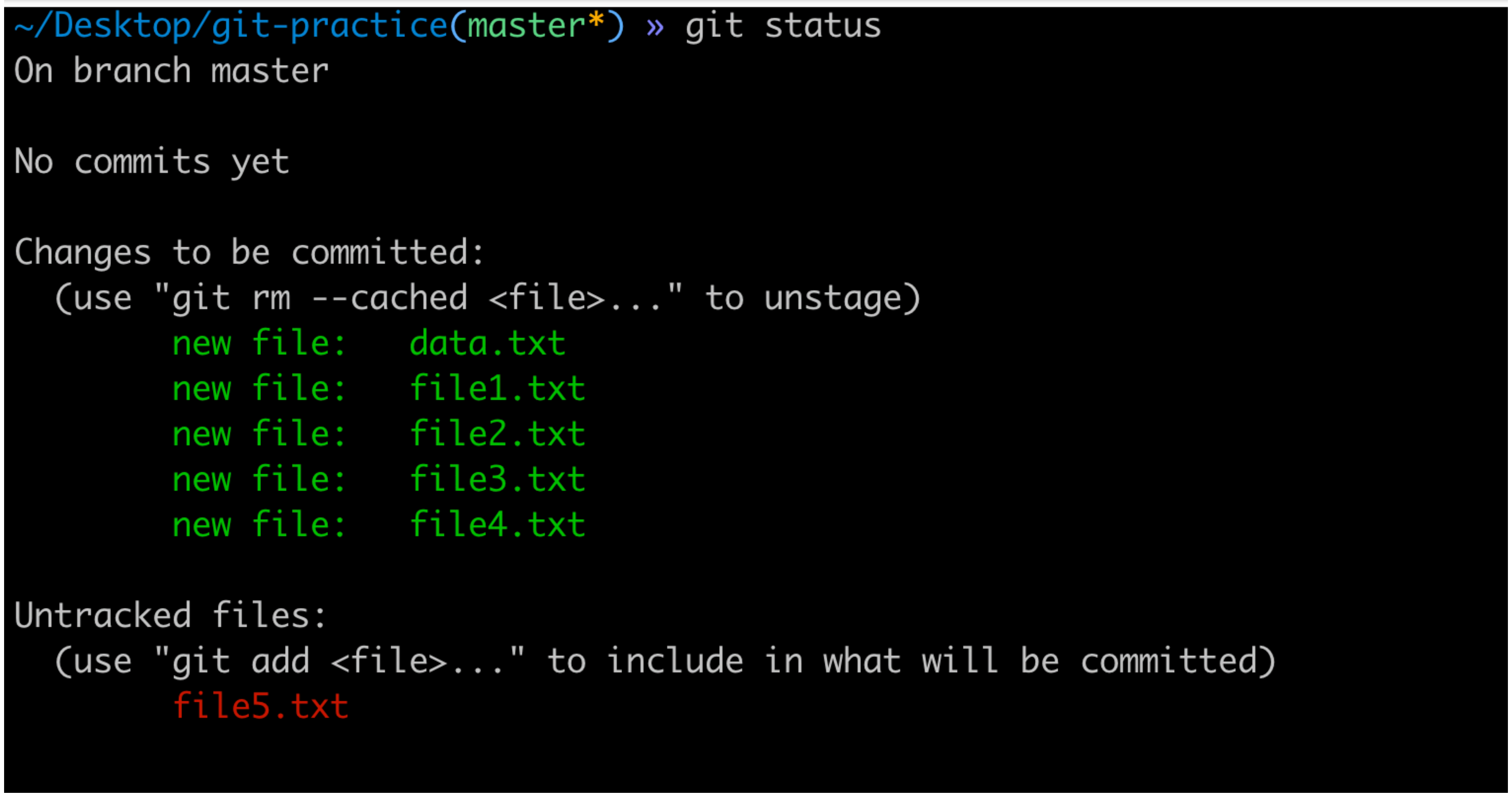
Everything is staged!

Un-staging Files - `git reset`

It looks like we actually don't want to add `file5.txt` to our staging area. We have made a mistake, so let's fix it. We can use the `git reset` command to remove a file from the staging area.

```
git reset file5.txt

git status
```

Explain this code



```
~/Desktop/git-practice(master*) » git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   data.txt
        new file:   file1.txt
        new file:   file2.txt
        new file:   file3.txt
        new file:   file4.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file5.txt
```

Now, `file5.txt` is back in our working directory and is no longer staged!

Saving Changes: `git commit`

What is a *Commit*?

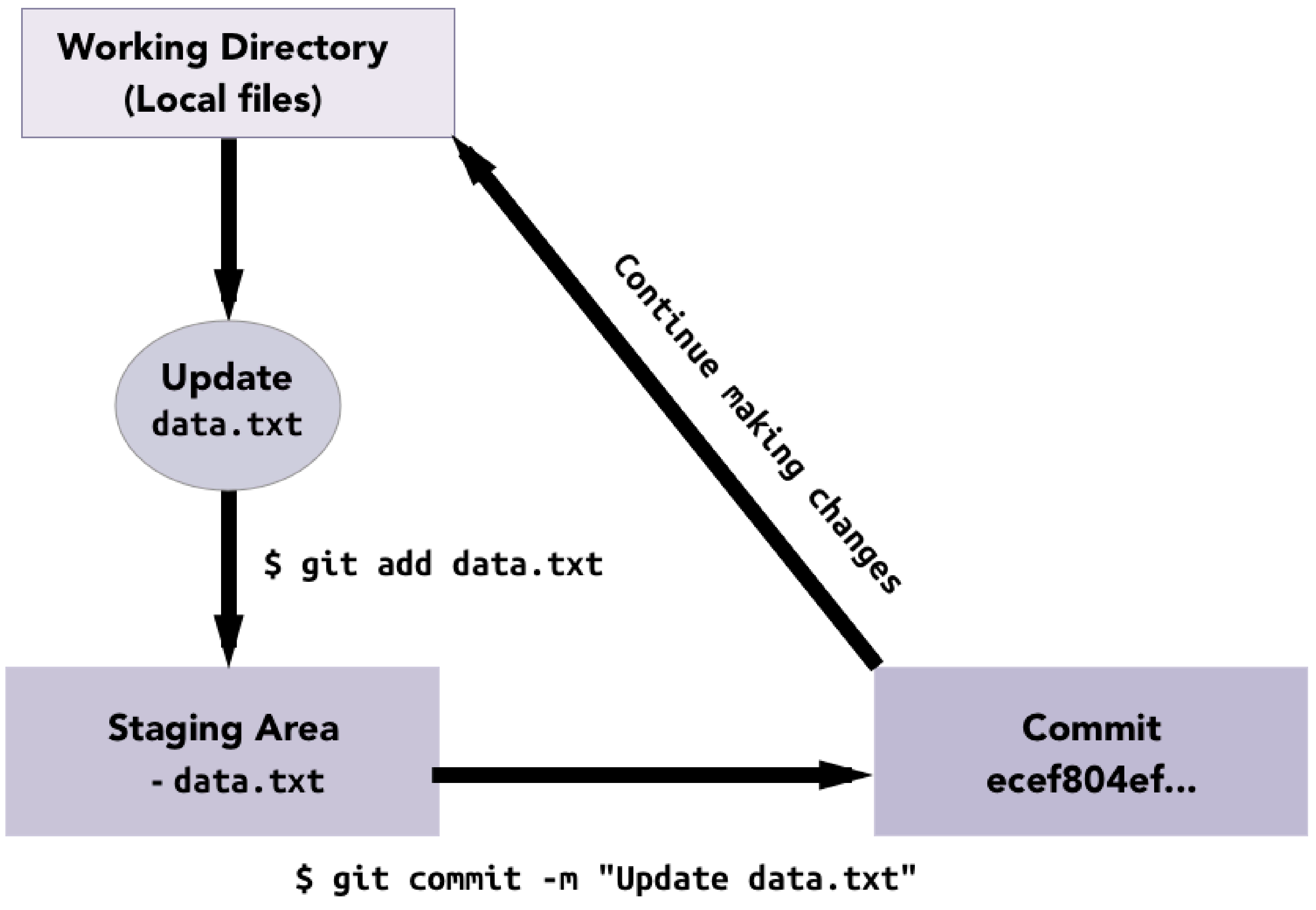
The question you may have been asking before this point is: *staging for what?* The answer lies in the screenshots. You have noticed files in the staging area are labeled with **"Changes to be committed"**.

A *commit* in Git **is a snapshot of the state of the files and folders in your project, as well as the content in them**.

Your projects are going to be evolving and changing with time. As you add content and modify them, its *state* changes. As their state changes, commits will capture that point in time.

💡 **State** in computer science is a term used to describe the status of an object, folder, files, etc. and its attributes or contents.

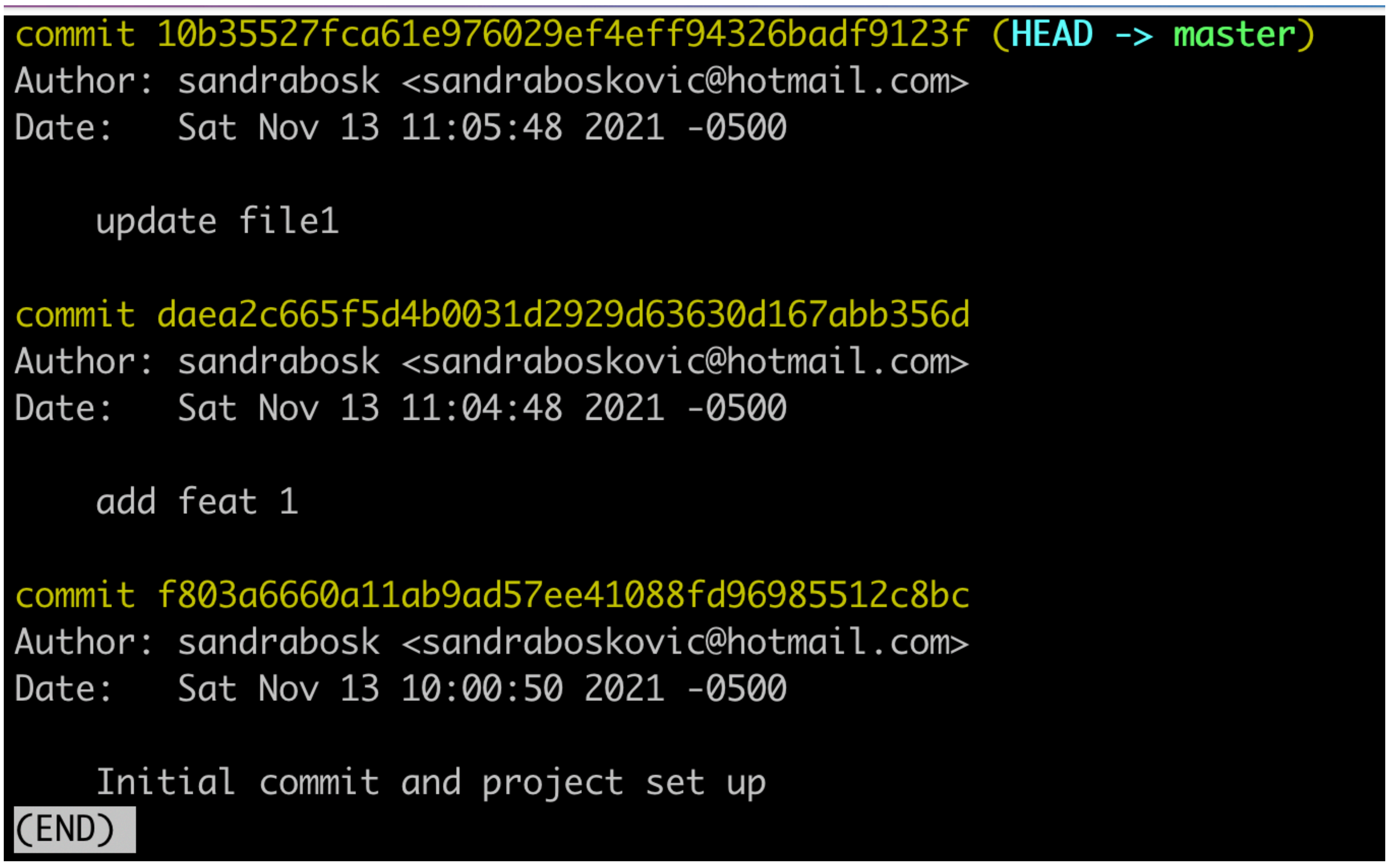
If `data.txt` contains: `"Hello"` and we modify that file to say: `"Goodbye"` the *state* has changed.



Using commits in Git will:

- help you revert mistakes to a previous working version (kind of *undo* button thing);
- enable you to collaborate with others on your projects without colliding and
- keep track of *who* made changes and *when* they did it.

For example, check out this sample commit log of a project's evolution:

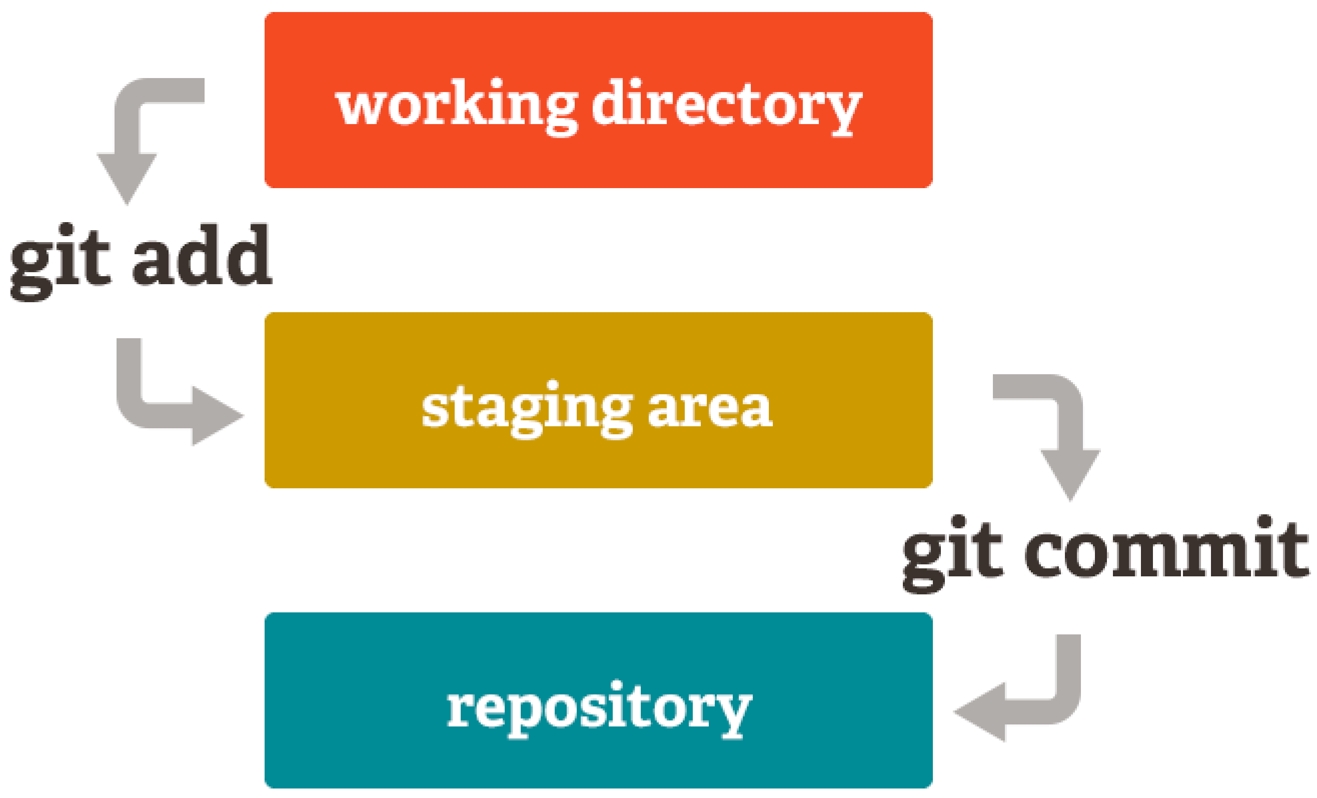


```
commit 10b35527fca61e976029ef4eff94326badf9123f (HEAD -> master)
Author: sandrabosk <sandraboskovic@hotmail.com>
Date:   Sat Nov 13 11:05:48 2021 -0500

    update file1

commit daea2c665f5d4b0031d2929d63630d167abb356d
Author: sandrabosk <sandraboskovic@hotmail.com>
Date:   Sat Nov 13 11:04:48 2021 -0500

    add feat 1

commit f803a6660a11ab9ad57ee41088fd96985512c8bc
Author: sandrabosk <sandraboskovic@hotmail.com>
Date:   Sat Nov 13 10:00:50 2021 -0500

    Initial commit and project set up
(END)
```

How to Make Commits

We can make commits using the `git commit` command. This will take any files in our staging area and create a new commit to our repository.

Let's take a look at the following diagram for some more visual insight:



We have already added files to our staging area from our working directory and now we need to move them permanently to our repository. The state of our project will forever be frozen in that snapshot.

```
git commit -m "Initial commit - Create files, add data"

git status
```

Copy

Explain this code

💡 `git commit -m` means we will provide a message with our commit. This will become more useful later, but for now, note that you should leave a **detailed and descriptive message** about what you have done in that commit.
Make a commit message in the present tense - as someone gave you a ticket on which they told you what to do. This is useful when you need to figure out what changes you have made in the past and when they were done.

What happened to our files? When we create commits, it removes the files from the staging area, because they have been committed. The only remaining file is `file5.txt`, which we did not add to the staging area.

Great. Now our files are committed and stored in Git, but how can we see commits that we have made previously and what we have done?

Revisiting and Viewing Commits: `git log`

At some point in the future, it is likely we will want to view all of the changes to our project. Besides, we may want to revert our project to a previous snapshot.

The `git log` command is used to view commits and data about those commits. Let's give it a run and see what happens:

Run the following to see the historical log of the commits that are made in the project:

```
git log
```

Copy

Explain this code

💡 Press `q` to exit the log dialog screen in your terminal.

Contained in this dialog is our commit and some data about that commit. Each commit has:

- **Commit SHA**(for example: `commit 905616a7252c247f3244bf6ca00faeeba324a26f`) - You can think of this as a unique ID for each commit. This can be used in the future to revert to this commit, remove the commit, or combine multiple commits.
- **Author** - The author attribute is useful when working with teams to see who did what.
- **Date** - Date when the commit was created.
- **Message** - The message we left when making the commit. This gives us some context as to what was done at that point. **This is why it is crucial to leave detailed and descriptive commit messages**.

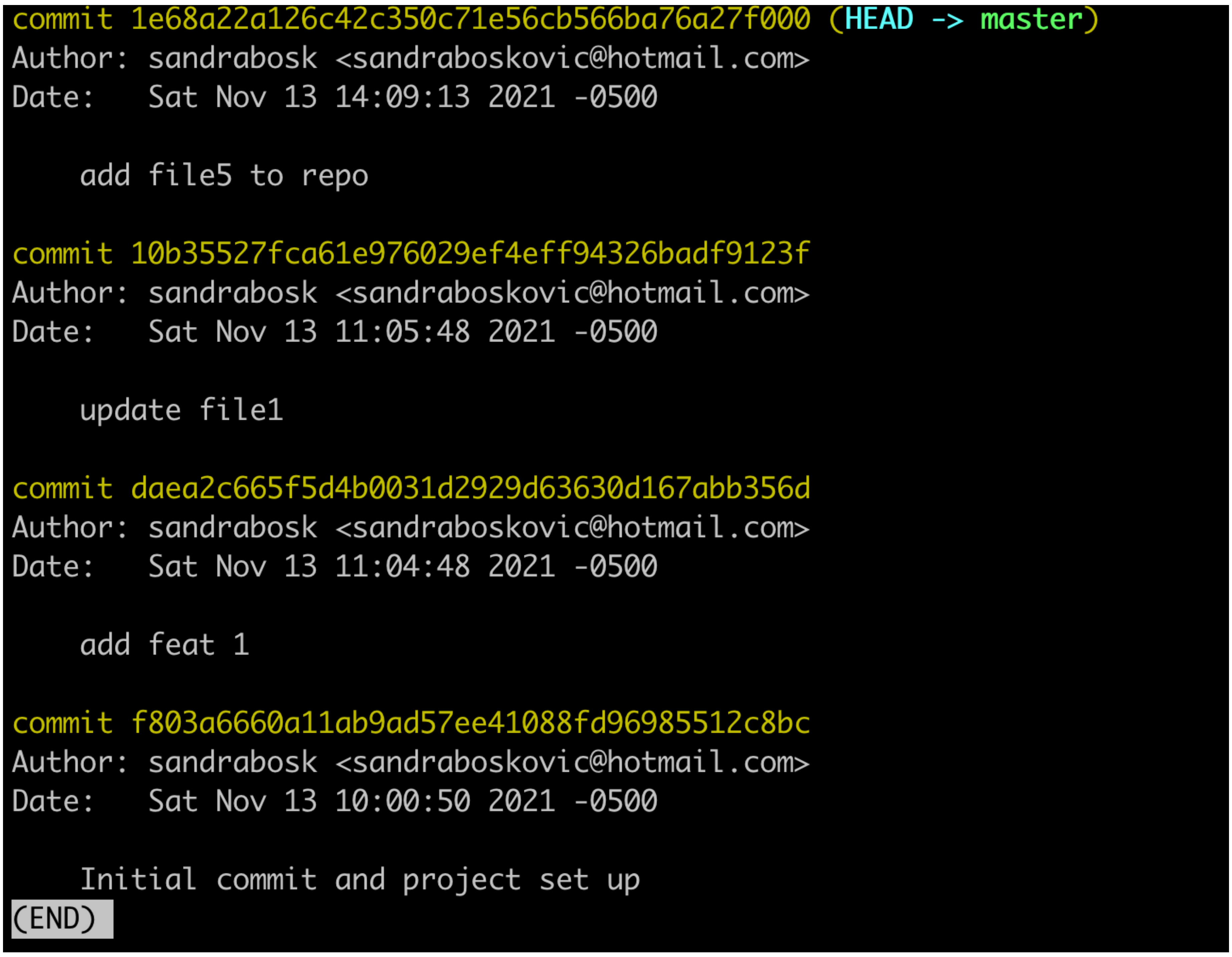
Let's make another commit and see how the log changes:

```
git add file5.txt

git commit -m "Add file5 to repo"

git log
```

Copy

Explain this code

Now you should see the last commit showing the message you just added - something along these lines:



```
commit 1e68a22a126c42c350c71e56cb566ba76a27f000 (HEAD -> master)
Author: sandrabosk <sandraboskovic@hotmail.com>
Date:   Sat Nov 13 14:09:13 2021 -0500

    add file5 to repo

commit 10b35527fca61e976029ef4eff94326badf9123f
Author: sandrabosk <sandraboskovic@hotmail.com>
Date:   Sat Nov 13 11:05:48 2021 -0500

    update file1

commit daea2c665f5d4b0031d2929d63630d167abb356d
Author: sandrabosk <sandraboskovic@hotmail.com>
Date:   Sat Nov 13 11:04:48 2021 -0500

    add feat 1

commit f803a6660a11ab9ad57ee41088fd96985512c8bc
Author: sandrabosk <sandraboskovic@hotmail.com>
Date:   Sat Nov 13 10:00:50 2021 -0500

    Initial commit and project set up
(END)
```

Our newest commit just showed up!

💡 Like most Git commands, `git log` is extremely customizable in how the output can be formatted. Check out the Advanced Git Log article for more details.

Quick Reference

Here you have the most important commands - don't try to memorize them but instead keep this reference open every time you go through this process (with time and repetition you will know these by heart):

Initialize a repository

```
git init
```

Copy

Explain this code

View changes in the repository

```
git status
```

Copy

Explain this code

Adding files

- single file

```
git add <file-name>
```

Copy

Explain this code

- all files

```
git add .
```

Copy

Explain this code

Committing file(s) to staging area

```
git commit -m"<commit-message>"
```

Copy

Explain this code

View changes history

```
git log
```

Copy

Explain this code

Quick Reference

Here you have the most important commands you should have learned in this lesson:

Add a remote repository

```
git remote add <alias> <github-url>
```

Copy

Explain this code

Pushing to Remote Repository

```
git push <alias> <remote-branch-name>
```

Copy

Explain this code

Pull changes from GitHub

```
git pull <alias> <remote-branch-name>
```

Copy

Explain this code

However, remember, before being able to push your code to the remote repository, you have to do all the steps we covered in the previous lesson:

1. `git init` ⇒ we run this command only once, only when initializing the git repo
2. `git add .`
3. `git commit -m"commit message in present tense"`
4. `git push remote-name master`

In the meantime, at any point, you can and should check the changes with `git status`.

Part 2: GitHub

What is GitHub?

GitHub is a tool to collaborate with others (DVCS) using Git. In short, GitHub allows us to host our Git repositories in the cloud so that others can share them.

GitHub also has features for:

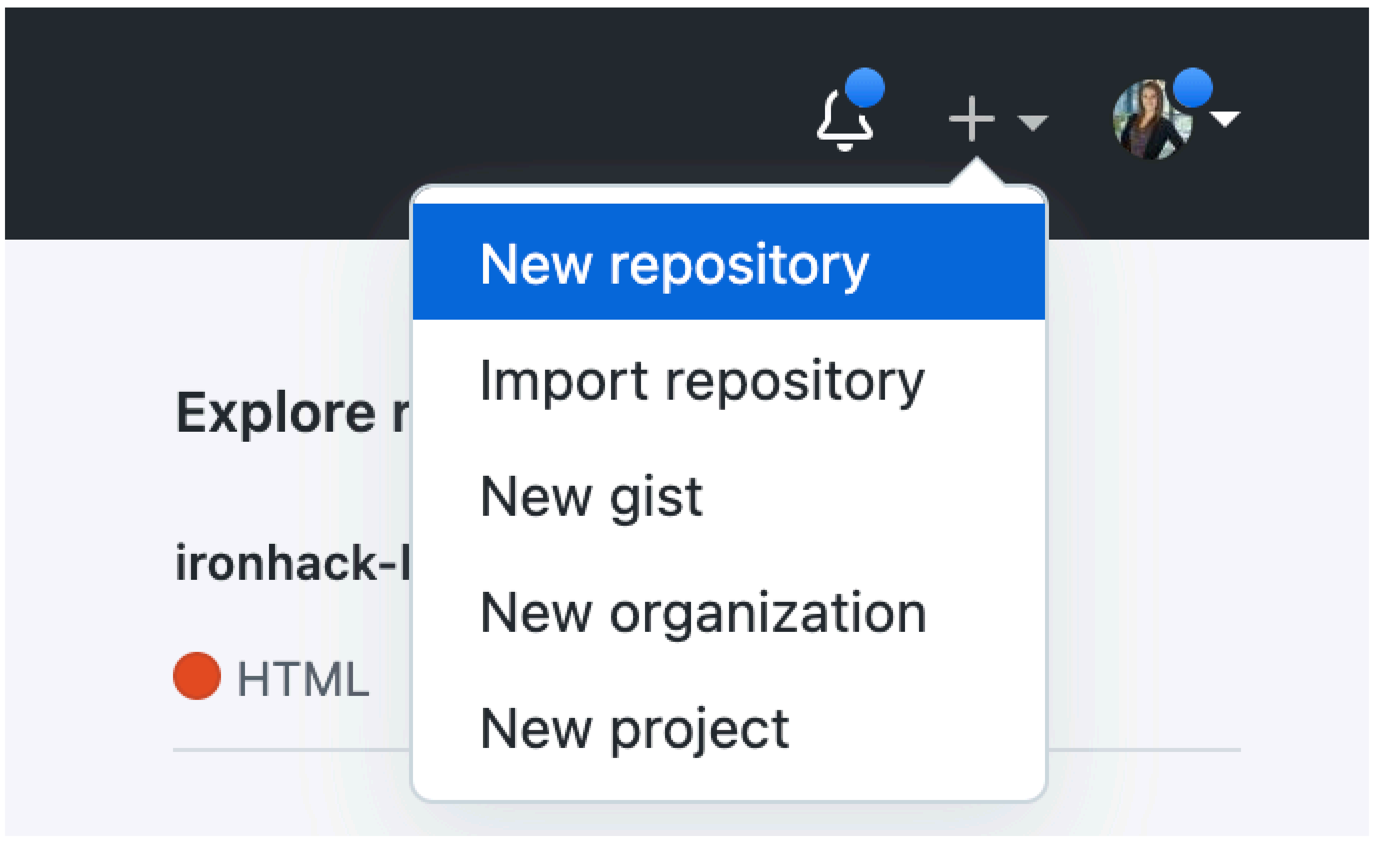
- **Managing Repos** - Team collaboration can be tricky. When multiple people are working on the same project, it is hard to find a source of truth and manage who can change the repository. GitHub has features to facilitate this.
- **Project Management** - With open source projects, it is often difficult to state the general direction of the product, transparently display which features are being implemented and the priority of those features.
- **Social Networking** - GitHub has features to search and find others, find repos with a specific language and follow/unfollow other coders.
- **Organizations and Team Management** - If you have a company or a project in which multiple people are working, you can create an *organization*, or a group of people with varying permissions and abilities on the repo.
- ...and many more.

Currently, we have a local Git repository. As it is, we can't share this repository with anyone, so let's change that.

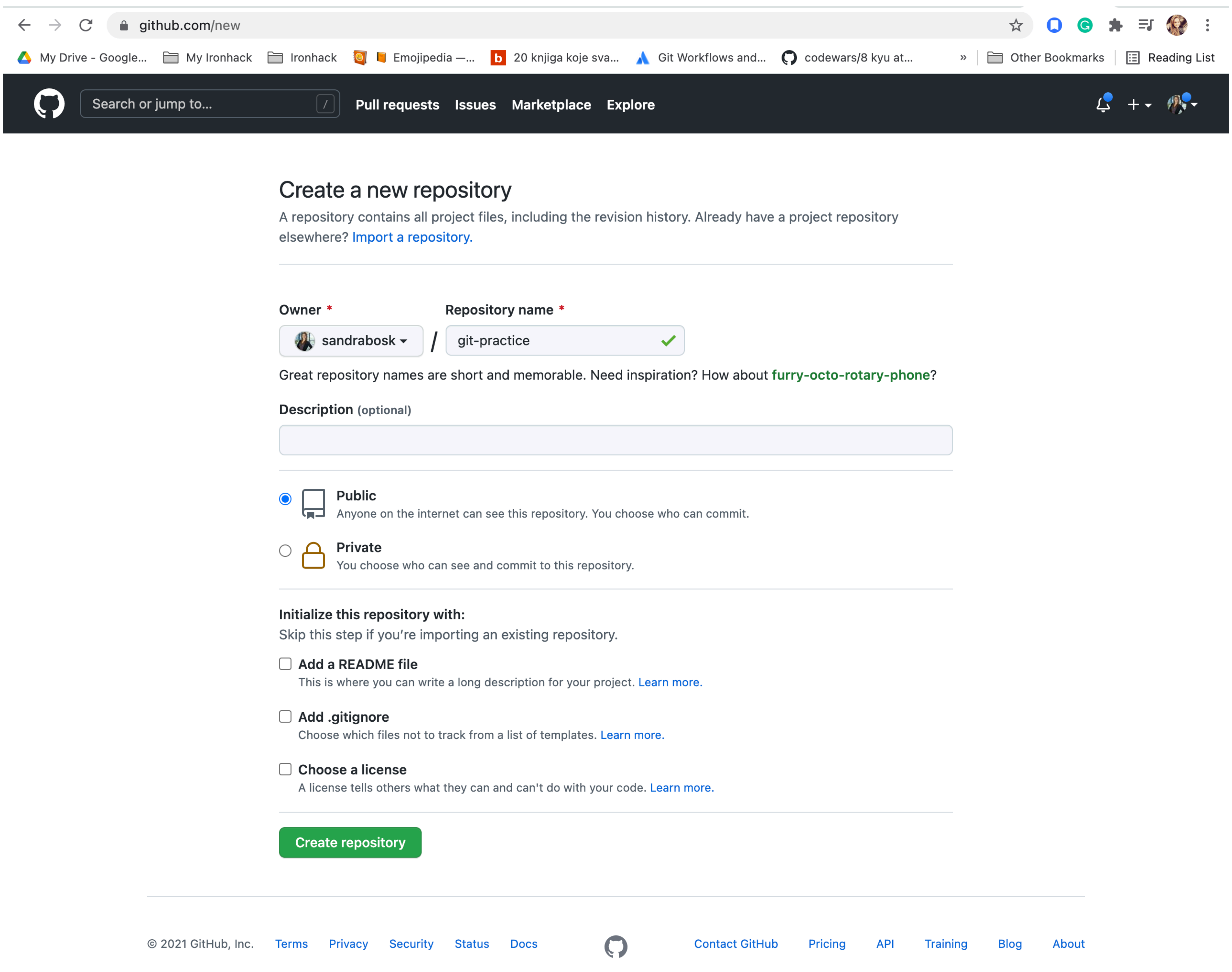
Creating a GitHub Repo

There are quite a few ways to create a GitHub repo, but let's talk about the simplest first. The easiest way to get up and running is to visit GitHub.

Click the plus sign in the upper right-hand corner of the page and then click "New Repository":



You will be presented with the following page. Fill in the data as it is in the screenshot:



github.com/new

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * sandrabosk / Repository name * git-practice

Great repository names are short and memorable. Need inspiration? How about furry-octo-rotary-phone?

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.

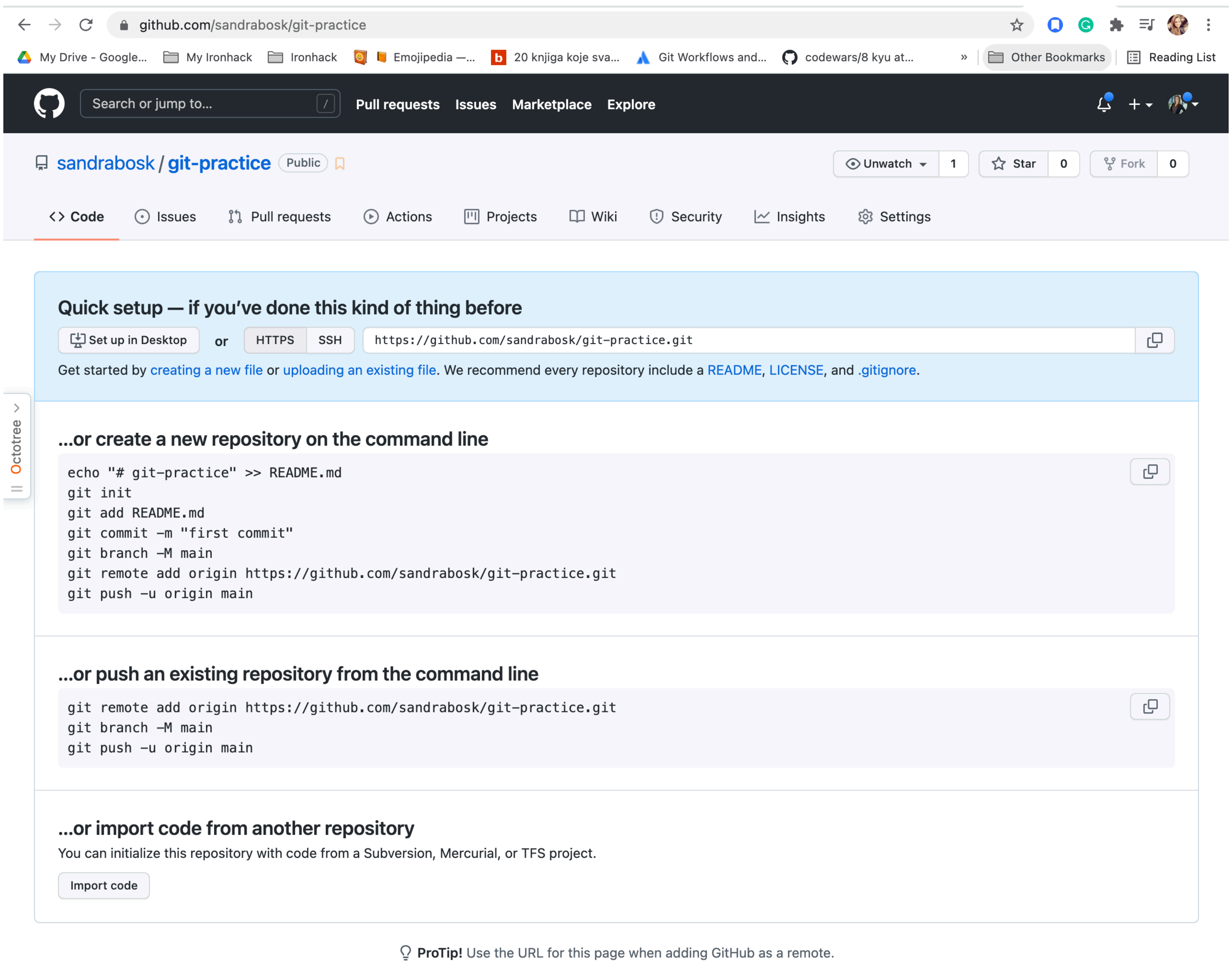
Create repository

© 2021 GitHub, Inc. Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About

A few things to note about this page:

- **Repository name** (*mandatory*) - This is the name of the repository as it will appear on GitHub. *This does not have to match the name of your local repository and can't have any spaces.*
- **Description** (*optional*) - Helpful for outsiders that may stumble upon your project on GitHub.
- **Public / Private** - If your repository is public, that means anybody can discover your project from Google or your profile. Real-world projects frequently will be private repos, so only those that are collaborators can view them. All your repositories during your Ironhack journeys should be public. That will allow your teacher/TAs to check your work and your future employers can see how you progressed.

Leave the rest as defaults for now and click "Create Repository". You will then be taken to a page that looks like the following:


github.com/sandrabosk/git-practice

sandrabosk / git-practice Public

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

Set up in Desktop or HTTPS SSH https://github.com/sandrabosk/git-practice.git

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

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

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

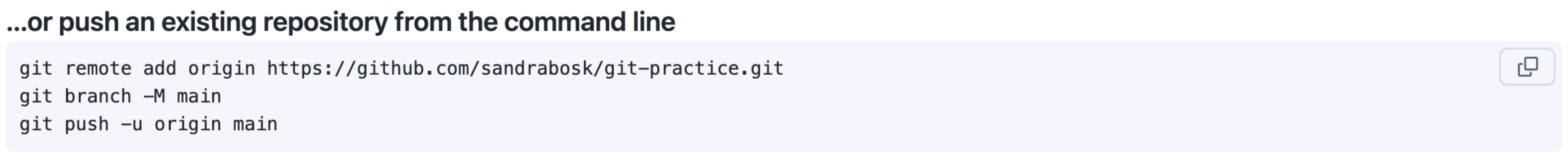
...or import code from another repository

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

Import code

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

What we will take special note of is the following section "**...or push an existing repository from the command line**", since we already have a local repo on our computers:


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

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

GitHub gives us some instructions here, but let's take a more in-depth look.

*In case you see the default branch name being different than in the screenshot, that is ok - sometimes you can see `master` and sometimes `main`. Both are in use.

Adding a Remote Repository: `git remote`

Our remote repository and local repository are two entirely different repos. When we make changes locally, we will have to *push* them to our remote repository - which lives on GitHub (the online cloud). When we want to grab changes that our collaborators made on their computers and pushed into the GitHub repo, we will have to *pull* them to our local repository.

⬆️ - **push** ⇒ take changes made locally to a remote repo that lives on GitHub ⬇️ - **pull** ⇒ take changes others made on the same repo and pushed to GitHub to your local repository

The first step in this is connecting the two repos. This can be done using the `git remote add` command while supplying a few options:

- **alias** - you create an alias in your system for the **remote** repository, which will be pointing to the GitHub repo. It is very common to call this alias `origin`, which we will advise you to do all the time except now. Since this is the learning process and we want you to understand that *origin* is a *remote* repo hosted on GitHub, we will call it *origin* (just for now).
- **GitHub repository URL** - a unique URL that GitHub provides for each repository.

Navigate to your `git-practice` folder on your computer if you are not already there and run the following:

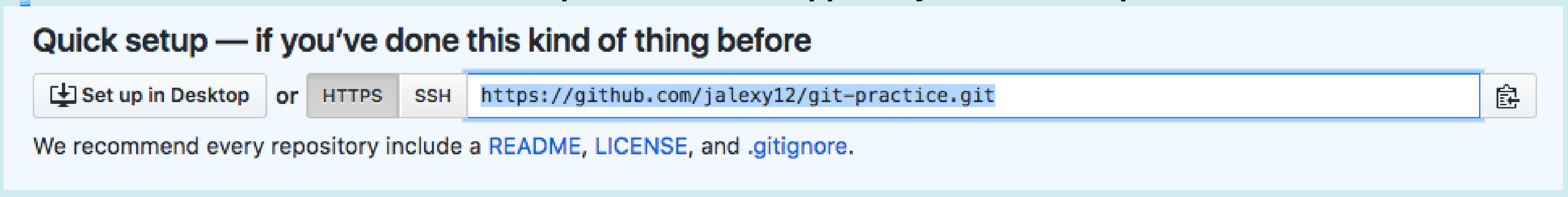
```
git remote add origin https://github.com/your-github-username/git-practice.git

# "origin" is a shorthand name for the remote repository where we will push the changes
# and where from we will pull the changes when our collaborators make them.
# "origin" is used instead of the original repository's URL - and thereby makes referencing much easier
```

Copy

Explain this code

To sum it up - we are using `origin` as an alias for our first GitHub project.

💡 **You can retrieve the link used in the previous code snippet to your GitHub repo here:**



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

Set up in Desktop or HTTPS SSH `https://github.com/jalexy12/git-practice.git`

We recommend every repository include a README, LICENSE, and .gitignore.

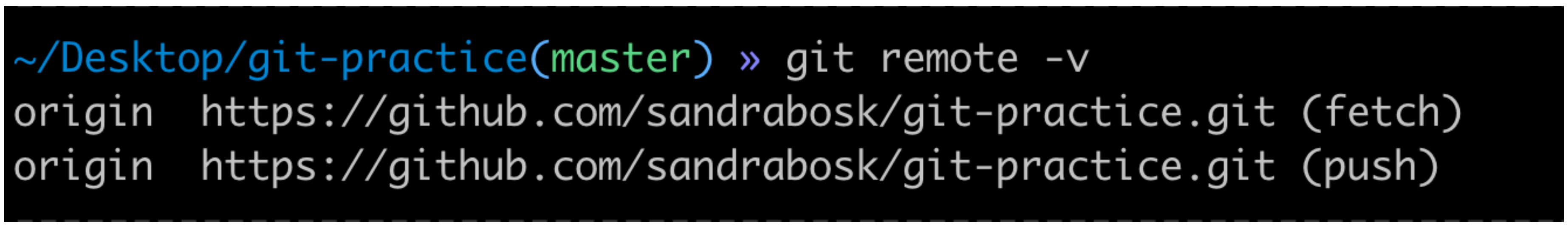
This is telling your local GitHub repo: "Add a remote repository called *origin* and have it point to `https://github.com/your-github-username/git-practice.git`".

We can view a list of remote repos attached to our current local repository by running `git remote` with no options:

```
git remote -v
```

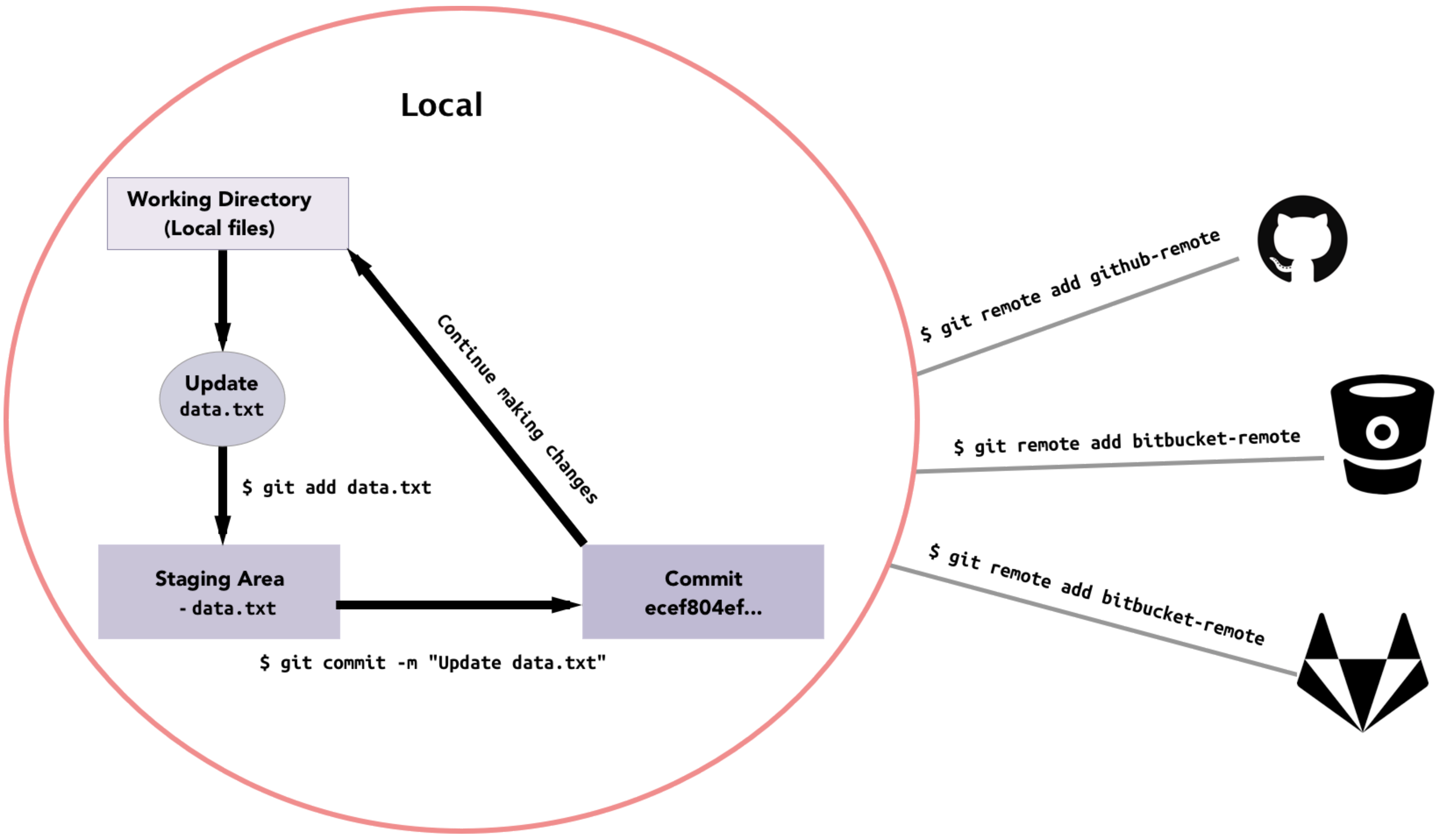
Copy

Explain this code



```
~/Desktop/git-practice(master) » git remote -v
origin  https://github.com/sandrabosk/git-practice.git (fetch)
origin  https://github.com/sandrabosk/git-practice.git (push)
```

You can have as many or as few remote links as you would like. GitHub is only one remote Git hosting service. There are others such as Bitbucket, Gitlab and many more.



Local
Working Directory (Local files)
Update data.txt
$ git add data.txt
Staging Area - data.txt
Commit ecef804ef...
$ git commit -m "Update data.txt"
Continue making changes
$ git remote add github-remote
$ git remote add bitbucket-remote
$ git remote add bitbucket-remote

Now that our local repository and remote repository are connected via a remote, let's push some code!

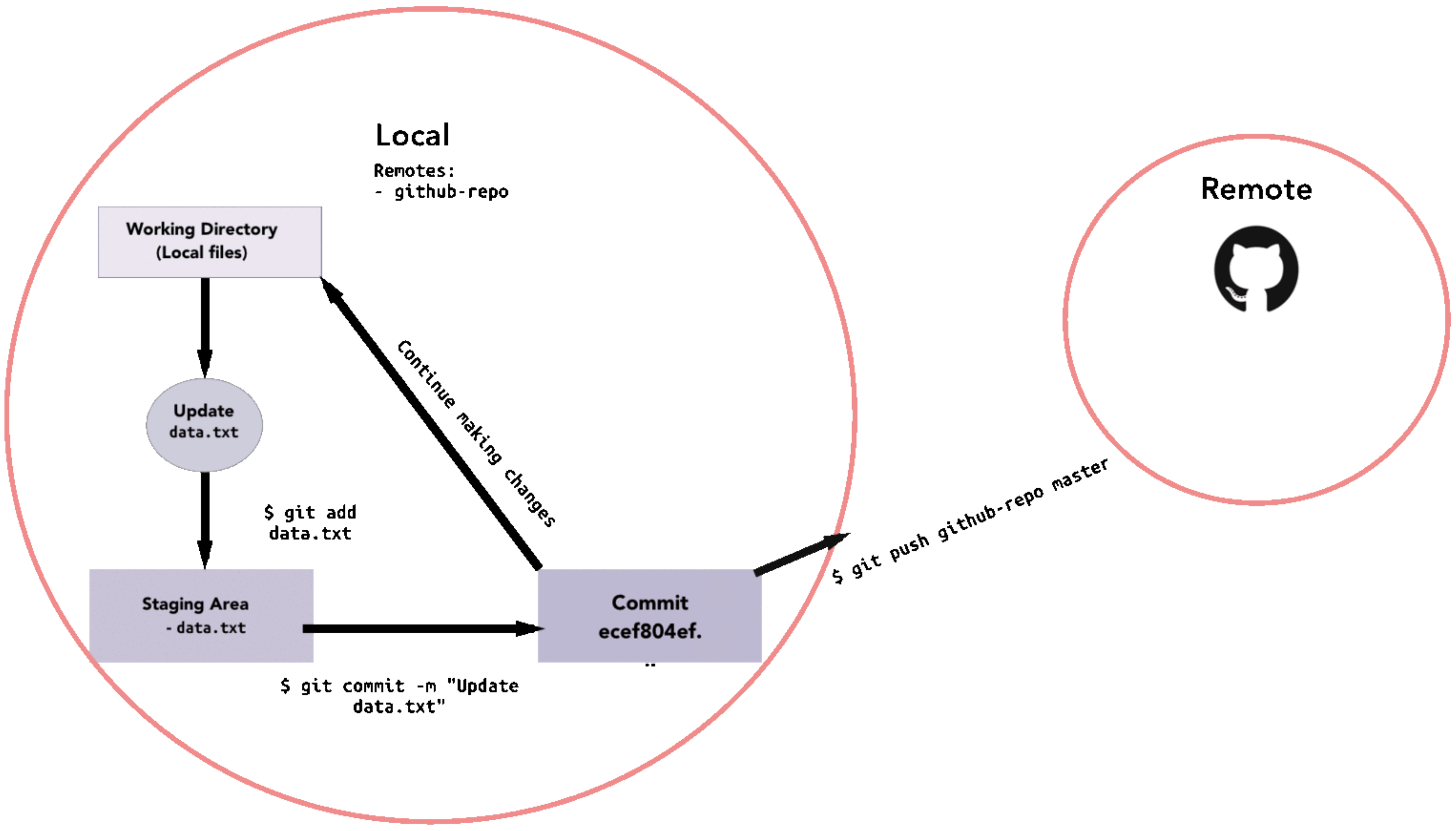
Adding changes to a remote repository: `git push`

Adding to a remote repository is pretty simple in most situations. We use the `git push` command, along with a couple of options to push our local repository to our remote repository (the first a couple of steps stay the same, you have to do: 1. `git add .`, 2. `git commit -m"some message"` and then push changes to remote repo):

Copy

```
# to add all the changes that are made in the repo
git add .

git commit -m "add a descriptive message"

git push origin master
```

Explain this code



- `git-push` - command to push code from a local to a remote repo;
- `origin` - an alias of a remote repository we want to push to. Eventually, we may have multiple remote repositories; therefore, we have to name them. However, you will most likely during the bootcamp use the default name which is `origin`, so this command will look like: `git push origin master`;
- `master` - the branch we are pushing to on GitHub. We will dive into branches later on.

The command line may prompt you for a username and password but hold for a second before proceeding. The next couple of lines is important before you go ahead and add your credentials.

Token authentication requirements for Git operations

> In July 2020, we announced our intent to require the use of token-based authentication (for example, a personal access, OAuth, or GitHub App installation token) for all authenticated Git operations. Beginning August 13, 2021, we will no longer accept account passwords when authenticating Git operations on GitHub.com.

> Source: The GitHub Blog - Token authentication requirements for Git operations

What does this mean? To put it in simple English, after the changes that happened in late 2021 in the GitHub authentication process we still *can* use passwords to login to GitHub but to be able to *push* changes to owned or forked repositories, we have to use Personal Authentication Tokens.

Let's do the setup and enable you to push changes you made in the repository locally to your remote repository on GitHub.

Create a Personal Access Token

To successfully create the Personal Access Token, follow the guidelines provided by the GitHub organization: Creating a token.

Hint 1: When on a step that requires you to set up the *Expiration*, click on the drop-down menu and select **Custom** to be able to use the calendar picker to set up the custom expiration date. Since you are using all your repositories for learning purposes, we recommend setting up the expiration for 6 months at least so you don't have to go through the process of updating the token too often. Once the token expires, you will be asked to generate a new token.

Hint 2: When asked to *Select scopes*, select **repo**. All the other options are considered more advanced and not necessary to be defined for now.

Hint 3: After you *generate the token*, make sure you copy it somewhere safe as you will need it at least 2 more times - to make the first push to your repo and to update your GitHub keychain credentials.

Creating and saving a personal token is the last thing you have to do in this step. Proceed to learn how to use it.

Using a token on the command line

Once you have a token, you can enter it instead of your password when performing Git operations over HTTPS. Follow the guidelines provided by the GitHub organization: Using a token.

As shown in the provided guidelines, you would **use a personal token when *cloning* a repo or *pushing* to a repo when you are asked to enter your password**.

To push the changes you made in the local repo to your GitHub repo:

```
git push origin master
```

Copy

Explain this code

```
Username: your_username
Password: your_token
```

Copy

Explain this code

If you are not prompted for your username and password, your credentials may be cached on your computer. Follow the steps listed below based on your machine:

- **Mac OS**

  You can update your credentials in the keychain to replace your old password with the token.

  Follow the guidelines provided here to successfully update your GitHub credentials on your keychain: Updating your credentials via Keychain Access.

Hint: When you open the keychain and in the search bar type “github.com” you should be able to update the credentials. You should **paste the token in the password field and save the changes** you made.

- **Windows OS**

  You can update your credentials in the Credential Manager to replace your old password with the token.

  1. Go to Credential Manager from Control Panel
  2. Windows Credentials
  3. Find `git:https://github.com`
  4. Edit - on *Password* paste your GitHub Personal Access Token
  5. Save the changes

- **Linux OS**

  Make sure you have a locally configured git client with a username and email:

```
git config -l
# user.name=your_username
# user.email=your_email
```

Copy

Explain this code

  If you don’t have it configured, follow the steps:

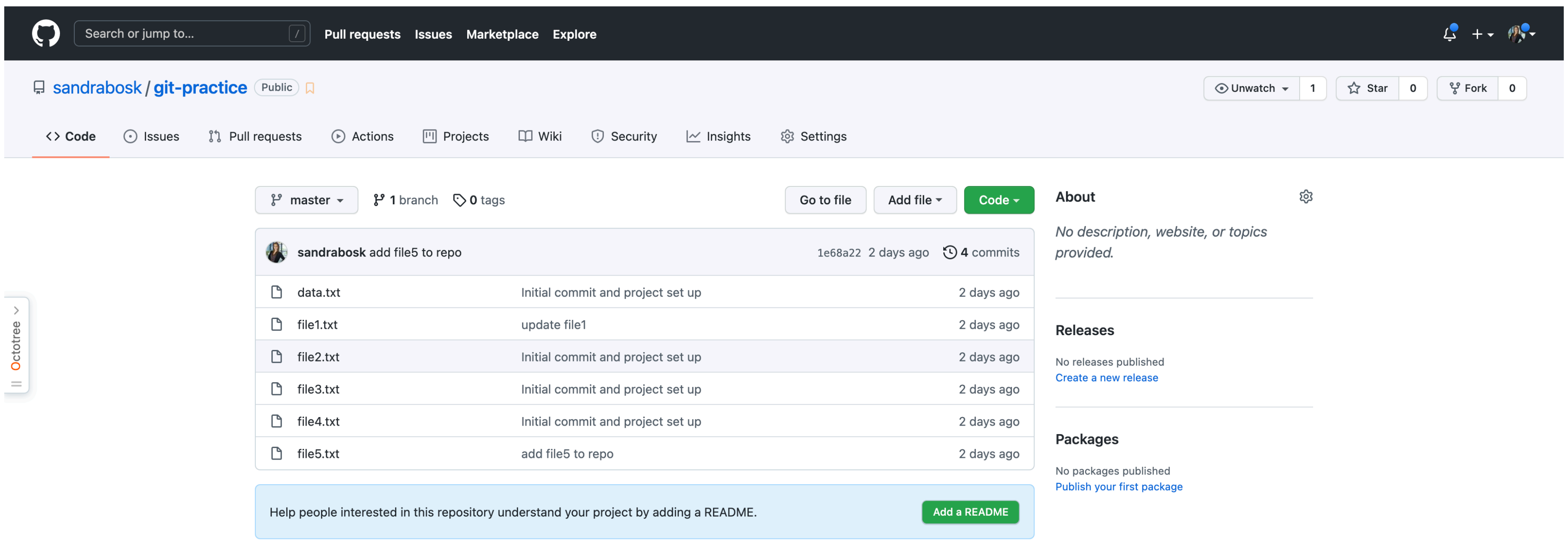
```
git config --global user.name "your_github_username"

git config --global user.email "your_github_email"
```

Copy

Explain this code

  Now when you try to clone the GitHub repo or push to it, you will be prompted to enter your username and password but instead of a password, you should paste the token.

Refresh GitHub and you should see the code that you have committed!



sandrabosk add file5 to repo 1e68a22 2 days ago 4 commits

data.txt	Initial commit and project set up	2 days ago
file1.txt	update file1	2 days ago
file2.txt	Initial commit and project set up	2 days ago
file3.txt	Initial commit and project set up	2 days ago
file4.txt	Initial commit and project set up	2 days ago
file5.txt	add file5 to repo	2 days ago

Help people interested in this repository understand your project by adding a README.

About

No description, website, or topics provided.

Releases

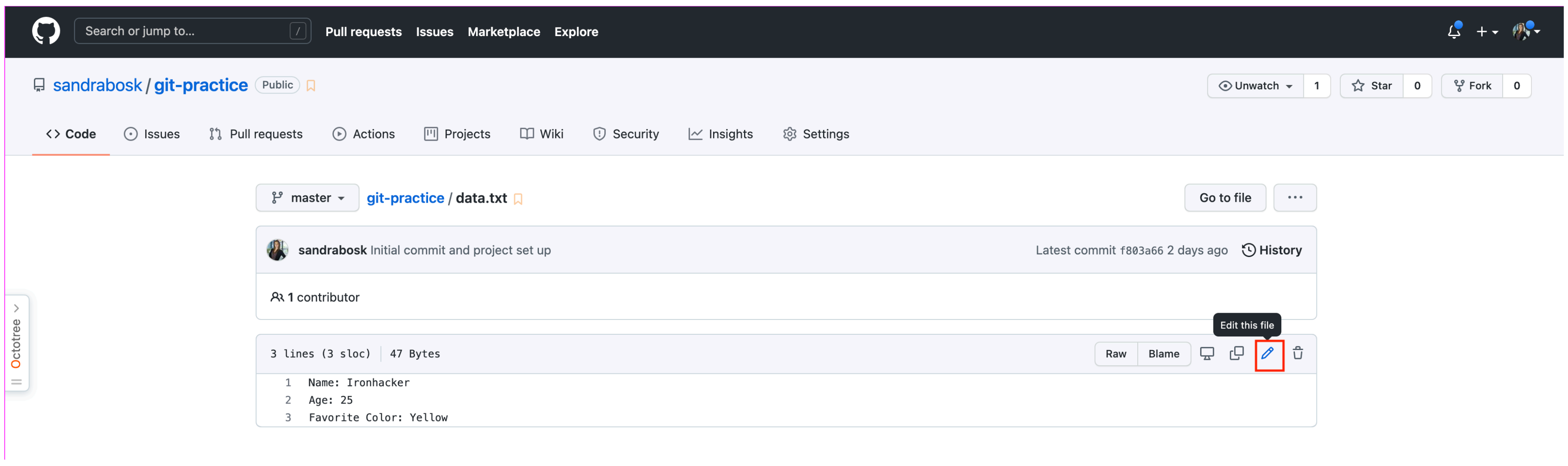
No releases published
Create a new release

Packages

No packages published
Publish your first package

Pulling From a Remote Repository: `git pull`

Occasionally GitHub repos will have changes that we don’t have on our local computer. This can happen when teammates push code to the repo, or when we update code on GitHub manually. In this situation, we have to use `git pull` to pull code to our local repository from GitHub. Let’s start by modifying a file on GitHub. On the repo’s page, click `data.txt`. On the following page, click the “Edit this File” button:



sandrabosk / git-practice

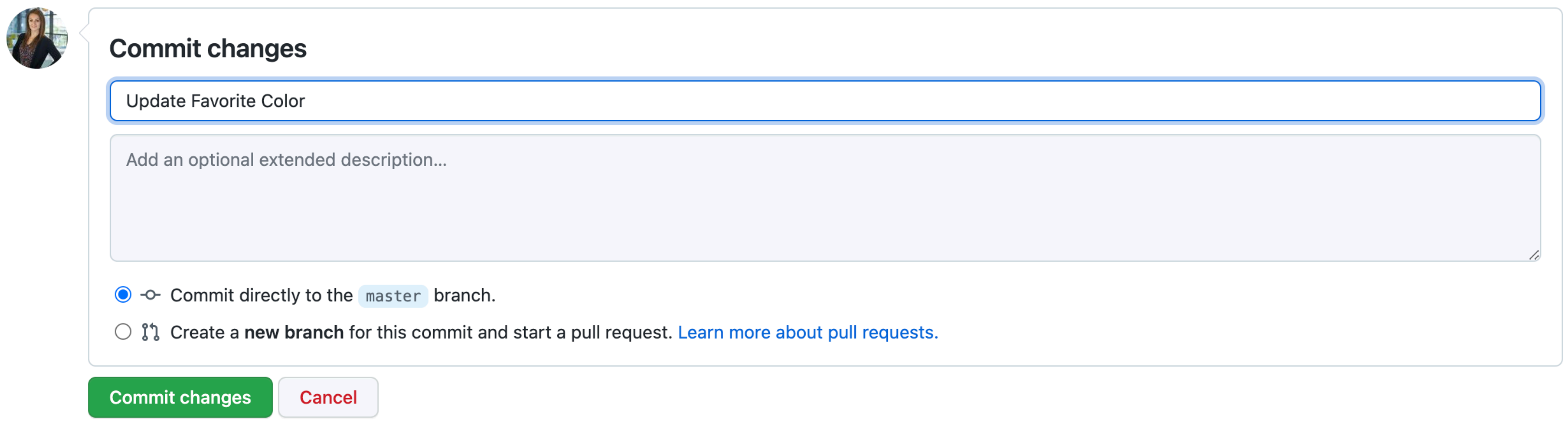
master · git-practice / data.txt

sandrabosk Initial commit and project set up — Latest commit f803a66 2 days ago

3 lines (3 sloc) | 47 Bytes

```
Name: Ironhacker
Age: 25
Favorite Color: Yellow
```

Modify anything you want in that file and then click “Commit Changes”:



Commit changes

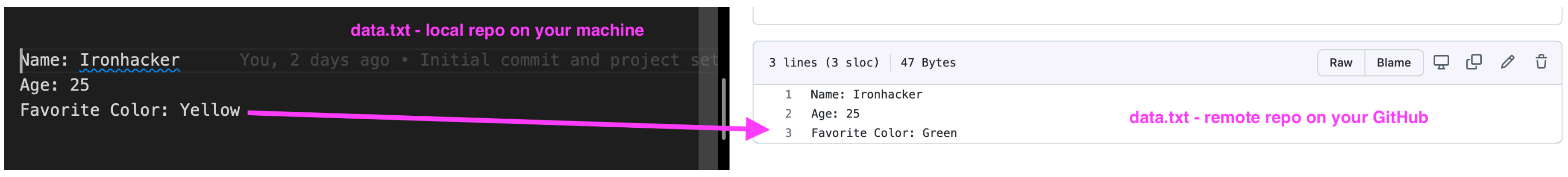
Update Favorite Color

Add an optional extended description...

[✓ Commit directly to the `master` branch.]

Create a **new branch** for this commit and start a pull request. Learn more about pull requests.

You have just made a commit to the repository on GitHub, which means that the file has gone through some change in the remote repository on GitHub, but if you open the file locally, nothing has changed.



data.txt - local repo on your machine
data.txt - remote repo on your GitHub

```
Name: Ironhacker        You, 2 days ago • Initial commit and project set
Age: 25
Favorite Color: Yellow
```

3 lines (3 sloc) | 47 Bytes

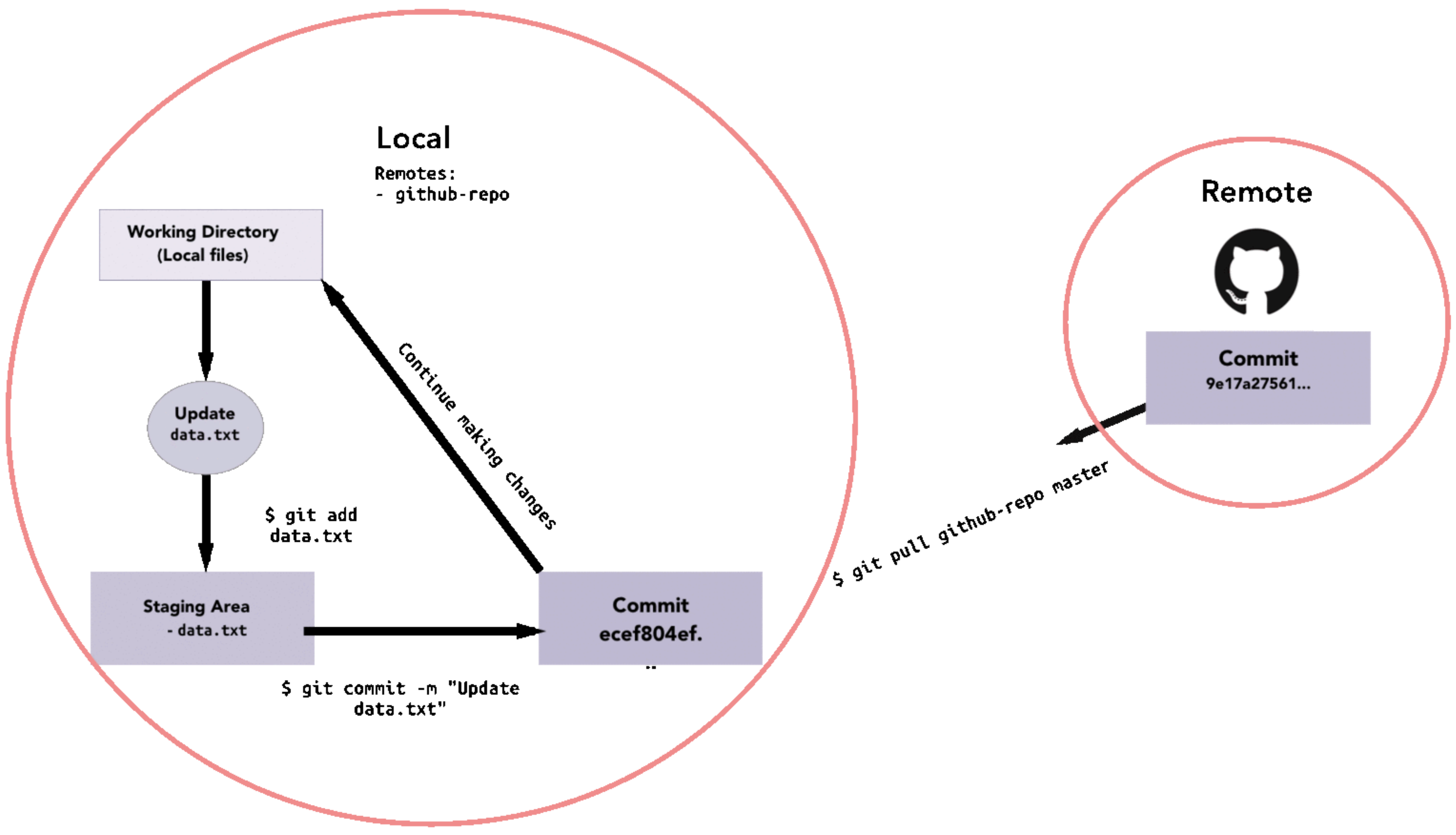
```
Name: Ironhacker
Age: 25
Favorite Color: Green
```

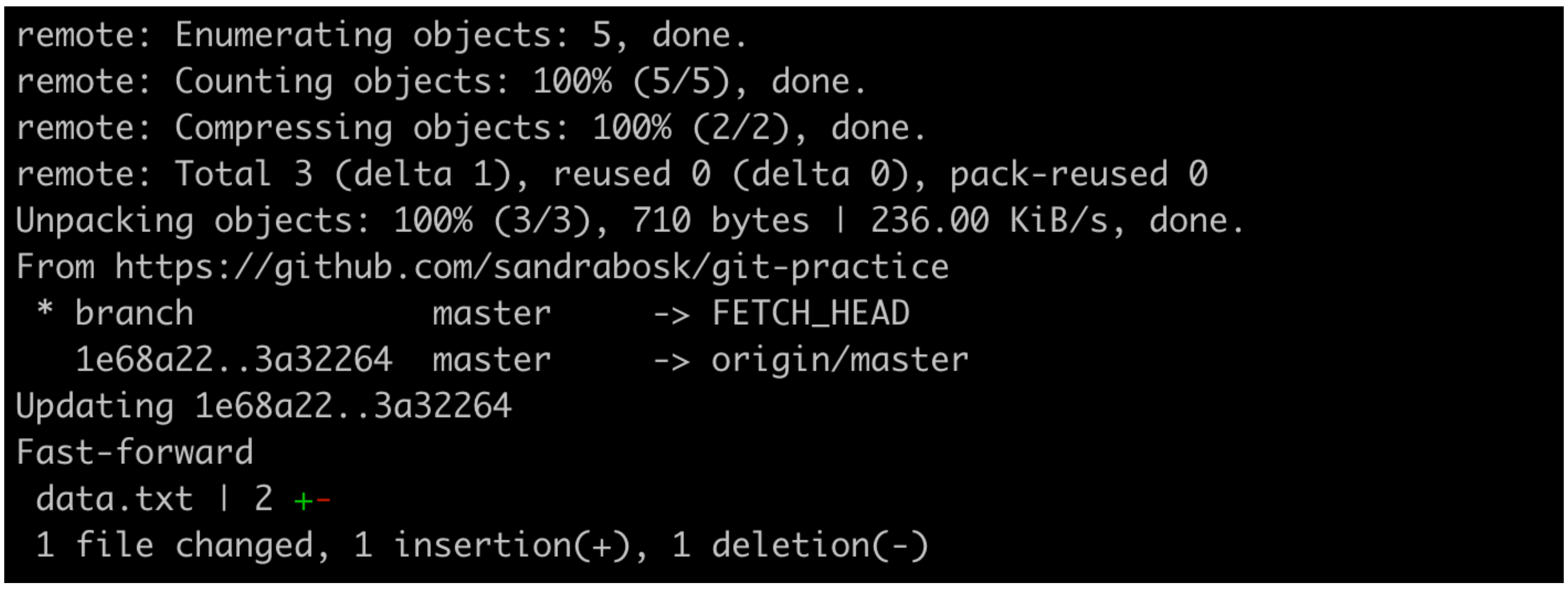
To pull our most recent commit from GitHub, we have to use `git pull`:

```
git pull origin master
```

Copy

Explain this code





```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 710 bytes | 236.00 KiB/s, done.
From https://github.com/sandrabosk/git-practice
 * branch            master     -> FETCH_HEAD
   1e68a22..3a32264  master     -> origin/master
Updating 1e68a22..3a32264
Fast-forward
 data.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

As you saw, the command `git pull` takes two arguments. The first being the **remote alias** and the second being the branch. If you take a look at `data.txt`, it should have your updates from GitHub!

Summary

In this lesson regarding Git, we have learned:

- how to initialize a Git repository,
- where to initialize a Git repository for an efficient workflow,
- how Git keeps track of changes in the `.git` folder,
- about *working* and *staging* areas and how to move files between them,
- how to view commits and what are the different pieces of data we can find in those commits.

These commands will be some of the most commonly used ones and will be crucial for your daily workflow. Get familiar with using them and don't be afraid to use `git status` all of the time!

In this lesson regarding GitHub we covered about:

- what GitHub is,
- how to push code from your local repository to the remote repo on GitHub,
- how to pull code from a GitHub repo to our computer (super useful when working with a team)

Extra resources

- **Git - Cheat Sheet**
- **Guide on how to set up Personal Access Token on different operating systems**

Any doubt? Ask our AI Chatbot!

Lesson Completed

Mark as not completed

How would you rate the content of this unit?