



TASK

Version Control and Git

[Visit our website](#)

Introduction

WELCOME TO THE VERSION CONTROL AND GIT TASK!

Knowing how to use version control is a crucial skill for any Software Developer working on a project, especially when working in a team of developers. The source code of a project is an extremely precious asset and must be protected. Version control software tracks all changes to the code in a special kind of database. Therefore, if a developer makes a mistake, they can compare earlier versions of the code to the current version to help fix the mistake while minimising disruption to the rest of the team. This task will introduce you to the basics of version control. It focuses on the Git version control system and the collaboration platform, GitHub.



A note from the
HyperionDev Team

A version control system is one of the most important tools for any software developer! Check out this [HyperionDev blog post](#) to see the other 4 tools essential for all developers.

WHAT IS A VERSION CONTROL SYSTEM?

Version control systems record modifications to a file or set of files so that you can recall specific versions of it later on. A version control system can be thought of as a kind of database. You are able to save a snapshot of your complete project at any time. Then, when you take a look at an older snapshot (or version) later on, your version control system shows you exactly how it differs from the current one.

Version control is independent of the kind of project, technology, or framework you are working with. For example, it works just as well for an Android app as it does for an HTML website. It is also indifferent to the tools you work with. You can use it with any kind of text editor, graphics program, file manager, etc.

WHY DO YOU NEED A VERSION CONTROL SYSTEM?

Below are some of the benefits of using a version control system for your projects:

- **Collaboration:** when working on a large (or even medium-sized) project, more often than not you will find yourself working as part of a team of developers. Therefore, you will have multiple people who need to work on the same file. Without a version control system in place, you will probably have to work together in a shared folder on the same set of files. It is, therefore, extremely difficult to know when someone is currently working on a file and, sooner or later, someone will probably overwrite someone else's changes.

By using a version control system, everybody on the team is able to work on any file at any time. The version control system then allows you to merge your changes into a common version, so the latest version of the project is stored in a common, central place.

- **Storing versions:** it is especially important to save a version of your project after making any modifications or changes. This can become quite confusing and tedious if you do not have a version control system in place. A version control system acknowledges that there is only one project being worked on, therefore, there is only one version on the disk you are currently working on. All previous versions are neatly stored inside the version control system. When you need to look at a previous version, you can request it at any time.
- **Restoring previous versions:** being able to restore older versions of a file enables you to easily fix any mistakes you might have made. Should you wish to undo any changes, you can simply restore your project to a previous version.
- **Understanding what happened:** your version control system requires you to provide a short description of the changes you have made every time you decide to save a new version of the project. It also allows you to see exactly what was changed in a file's content. This helps you understand the modifications that were made in each version of the project, even if you weren't the one who made them.
- **Backup:** a version control system can also act as a backup. Every member of the team has a complete version of the project on their disk. This includes the project's complete history. If your central server breaks down and your

backup drive fails, you can recover your project by simply using a team member's local repositories.

THE GIT VERSION CONTROL SYSTEM

In this course, we will be using the Git version control system. Git is the most widely used modern version control system. It is free and open-source and is designed to handle everything from small to very large projects.

Git has a distributed architecture and is an example of a distributed version control system (DVCS). This means that with Git, every developer's working copy of the code is also a repository that contains the full history of all changes, instead of having only one single place for the full version history of the project.

As well as being distributed, Git has been designed with performance, security, and flexibility in mind.

INSTALLING GIT

Before you start learning how to use Git, you must install it. Even if you already have it installed, you should ensure that you update it to the latest version. Below are the instructions on how to install Git on Windows, Mac and Ubuntu:

Installing Git on Windows

1. Go to <http://git-scm.com/download/win> to download the official build from the Git website. The download should start automatically.
2. After starting the installer, you should see the Git Setup wizard screen. Click on the Next and Finish prompts to complete the installation.
3. Open a Command Prompt.
4. Configure your Git username and email using the following commands:

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

Installing Git on Mac

1. Download the latest [Git for Mac installer](#)
2. Follow the prompts to install Git.
3. Open a terminal.

4. Verify that the installation was successful by typing the following command into the terminal:

```
git --version
```

5. Configure your Git username and email using the following commands:

```
git config --global user.name "Your Name"
```

```
git config --global user.email "youremail@email.com"
```

Installing Git on Ubuntu

1. Install Git by typing the following commands into your terminal:

```
sudo apt-get update
```

```
sudo apt-get install git
```

2. Verify that the installation was successful by typing the following into the terminal:

```
git --version
```

3. Configure your Git username and email using the following commands:

```
git config --global user.name "Your Name"
```

```
git config --global user.email "youremail@email.com"
```



Extra resource

If you're using a different Linux/Unix distribution, you can use [Git-SCM](#) to download Git using the native package manager on numerous platforms.

Before we dive into actually using Git, we need to understand a few important concepts.

REPOSITORIES

A repository can be thought of as a kind of database where your version control system stores all the files for a particular project. A repository in Git is a hidden folder called `.git`, which is located in the root directory of your project. Fortunately, you do not have to modify anything in this folder. Simply knowing that it exists is good enough for now.

There are two types of repositories, namely, local repositories and remote repositories. A local repository is located on your local computer as the `.git` folder inside the project's root folder. You are the only person that can work with this

repository. A remote repository, however, is located on a remote server on the internet or in your local network. Teams of developers use remote repositories to share and exchange data. These serve as a common base where everybody can publish and receive changes.

You can get a repository on your local machine in one of two ways:

- You can initialise a new repository that is not yet under version control for a project on your local computer.
- You can get a copy of an existing repository. For example, if you join a company with a project that is already running, you can clone this repository to your local machine.

COMMIT

A commit is a wrapper for a set of changes. Whenever someone makes a commit, they are required to explain the changes that they made with a short commit message so that, later on, people looking at the project can understand what changes were made.

Every set of changes creates a new, different version of your project. Therefore, every commit marks a specific version. The commit can be used to restore your project to a certain state as it's a snapshot of your complete project at a certain point in time.

THE WORKING DIRECTORY, STAGING AREA, AND GIT REPOSITORY

Your files can have three main states in Git: **committed**, **modified** and **staged**. If your file is committed, its data is safely stored in your local database. If it is modified, the file has changed but has yet to be committed to your database. If it is staged it means that you have marked a modified file to go into your next commit in its current version.

This leads to the main sections of a Git project: the **working directory**, **staging area** and **Git repository**. A working directory consists of files that you are currently working on. The staging area is a file that is contained in the Git repository. It stores information about what will go into your next commit. The staging area acts as the interface between the repository and the working directory. All changes added to the staging area will be the ones that actually get committed into the Git repository, which is where Git stores the metadata and object database for your project.

A version of a file is considered committed if it is in the Git repository. If it has been changed and has been added to the staging area, it is considered staged. If it has changed but has not been staged, it is modified.

BASIC GIT WORKFLOW

Below is the basic Git workflow:

1. Modify a file from the working directory.
2. Add these modified files to the staging area.
3. Perform a commit operation to move the files from the staging area and store them permanently in the Git repository.

GITHUB

GitHub is an online Git repository hosting service. It is free to use for open-source projects and offers paid plans for private projects.

GitHub offers all of the functionality of Git while also adding its own features. While Git is a command-line tool, GitHub provides a web-based graphical interface. It provides access control and many features that assist with collaboration, such as wikis and basic task management tools for all projects.

Each project hosted on GitHub will have its own repository. Anyone can sign up for an account on GitHub and create their own repositories. They can then invite other GitHub users to collaborate on their project.

GitHub is not just a project-hosting service, it is also a large social networking site for developers and programmers. Each user on GitHub has their own profile, showing their past work and contributions they have made to other projects. GitHub allows users to follow each other, subscribe to updates from projects, or like them by giving them a star rating.

GETTING A GIT REPOSITORY

There are two ways to get a Git project. You can either initialise a new repository or clone an existing repository.

Initialising a Repository

To create a new repository, you have to initialise it using the **init** command. To do this, open your terminal (or command prompt if you are using Windows) and go to your project's directory. To change your current directory, use the **cd** (change directory) command followed by the pathname of the directory you wish to access.

After you have navigated to your project's directory, enter the following command:

```
git init
```

This creates a new, hidden subdirectory called **.git** in your project directory. This is where Git stores necessary repository files, such as its database and configuration information, so that you can track your project.

Cloning a Repository

If you would like to get a copy of an existing Git repository, such as a project you would like to contribute to, you need to use the Git **clone** command. Running a Git clone command pulls a complete copy of the remote repository to your local system. To use this command, enter **git clone [repository_url]** into the terminal or command prompt. For example, if you would like to clone the Wikimedia Commons Android App repository, you would enter the following:

```
git clone https://github.com/commons-app/apps-android-commons.git
```

This creates a new directory called "apps-android-commons", initialises a **.git** directory within it and pulls all the data from the remote repository. If you go to this new directory, you will find all of the project files, ready to be used.

ADDING A NEW FILE TO THE REPOSITORY

Now that your repository has been cloned or initialised, you can add new files to your project using the **git add** command.

Assume that you have set up a project at `/Users/user/your_repository` and that you have created a new file called **newFile.py**. To add newFile.py to the repository

staging area, you would need to enter the following into your terminal or command prompt:

```
cd /Users/user/your_repository  
git add newFile.py
```

It can be tedious to add files individually, so if you want to stage all of your project files, you can do so with this command:

```
git add .
```

The full stop in this case means 'everything'.

CHECKING THE STATUS OF YOUR FILES

Files can either exist in a **tracked** state or in an **untracked** state in your working directory. Tracked files are files that were in the last snapshot, while untracked files are any files in your working directory that were not in your last snapshot and are not currently in the staging area. We use the **git status** command to determine which files are in which state.

Using the **git add** command begins tracking a new file. If you run the **git status** command after you have added **newFile.py**, you should see the following code, showing that **newFile.py** is now tracked:

```
git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
Changes to be committed:  
  (use "git reset HEAD <file>..." to unstage)  
  
    new file:   newFile.py
```

You can tell that **newFile.py** is staged because it is under the "Changes to be committed" heading.

COMMITTING YOUR CHANGES

You should now be ready to commit your staged snapshot to the project history using the **commit** command. If you have edited any files and have not run **git add**

on them, they will not go into the commit. To commit your changes, enter the following:

```
git commit -m "added new file newFile.py"
```

The message after the -m flag inside the quotation marks is known as a commit message. Every commit needs a meaningful commit message. This makes it easier for other people who might be working on the project (or even for yourself later on) to identify what modifications you have made. Your commit message should be short and descriptive, and you should write one for every commit you make.

VIEWING THE CHANGE HISTORY

Git saves every commit that is ever made in the course of your project. To see your repository or change history over time, you need to use the **git log** command. Running the **git log** command shows you a list of changes in reverse chronological order, meaning that the most recent commit will be shown first. The **git log** command displays the commit hash (which is a long string of letters and numbers that serves as a unique ID for that particular commit), the author's name and email, the date written and the commit message.

Below is an example of what you might see if you run **git log**:

```
git log
commit a9ca2c9f4e1e0061075aa47cbb97201a43b0f66f
Author: HyperionDev Student <hyperiondevstudent@gmail.com>
Date: Mon Sep 8 6:49:17 2017 +0200
```

Initial commit.

There are a large number and variety of options to the **git log** command that enable you to customise or filter what you would like to see. One extremely useful option is **--pretty** which changes the format of the log output. The **oneline** option is one of the prebuilt options available for you to use in conjunction with **--pretty**. This option displays the commit hash and commit message on a single line. This is particularly useful if you have many commits.

Below is an example of what you might see if you run **git log --pretty=oneline**:

```
git log --pretty=oneline
a9ca2c9f4e1e0061075aa47cbb97201a43b0f66f Initial commit.
```

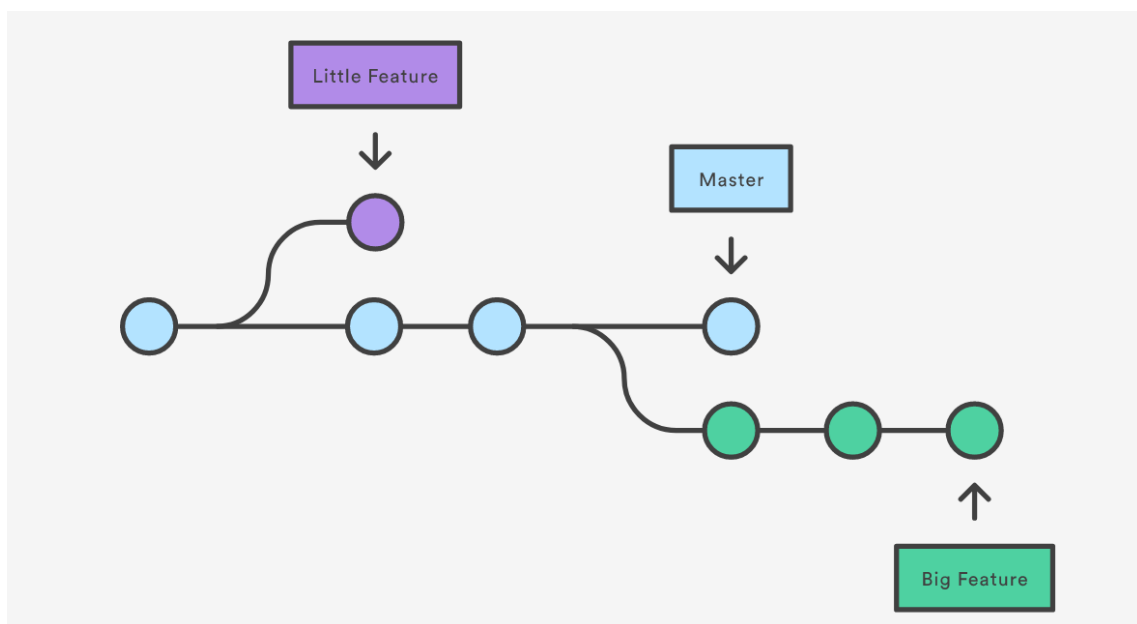
For the full set of options, you can run **git help log** from your terminal or command prompt. Alternatively, take a look at the [reference documentation](#).

BRANCHES

It is common for several developers to share and work on the same source code. Since different developers will have to be able to work on different parts of the code at the same time, it is important to be able to maintain different versions of the same codebase. This is where **branching** comes in.

One of the fundamental aspects of working with Git is branching. A branch represents an independent line of development. It allows each developer to branch out from the original codebase and isolate their work from others. By branching, you diverge from the main line of development and continue to work without messing up or disrupting the main line.

Branches are essential when working on new features or bug fixes. You create a **new branch** whenever you add a new feature or fix a bug to encapsulate your changes. This ensures that unstable code is not committed to the main codebase and also enables you to clean up your feature's history before merging it into the main branch.



"A repository with two isolated lines of development" by [Atlassian](#) under CC BY 2.5 Australia; from [Atlassian](#)

The image above visually represents the concept of branching. It shows a repository with two branches; one for a small feature and one for a larger feature. As you can see, each branch is an isolated line of development which can be worked on in parallel and keeps the main branch, known as the master branch, free from dubious code.

Git creates a master branch automatically when you make your first commit in a repository. Until you decide to create a new branch and switch over to it, all following commits will go under the master branch. You are, therefore, always working on a branch.

The HEAD is used by Git to represent the current position of a branch. By default, the HEAD will point to the master branch for a new repository. Changing where the HEAD is pointing will update your current branch. You can check where the HEAD is currently pointing with the **git status** command which will display this information in the first line of output.

Creating a Branch

To create a new branch, use the **git branch** command, followed by the name of your branch. For example:

```
git branch my-first-branch
```

Switching Branches

Using the **git branch** command does not switch you to the new branch; it only creates the new branch. To switch to the new branch that you created, use the **git checkout** command.

```
git checkout my-first-branch
```

Using this command moves the HEAD to the my-first-branch branch.

Alternatively, you can run the **git checkout** command with a **-b** switch to create a branch and switch to it at the same time. For example:

```
git checkout -b my-second-branch
```

This is short for:

```
git branch my-second-branch
```

```
git checkout my-second-branch
```

Saving Changes Temporarily

When you make a commit, you save your changes permanently in the repository. However, you might find that you would like to save your local changes temporarily. For example, imagine you are working on a new feature when you are suddenly required to make an important bug fix right away. Obviously, the changes you made so far for your feature don't belong to the bug fix you are going to make.

Fortunately, with Git, you don't have to deploy your bug fix with the new feature changes you have made. All you have to do is switch back to the master branch.

Before you switch to the master branch, however, you should first make sure that your working directory or staging area has no uncommitted changes in it otherwise Git will not let you switch branches. Therefore, it is better to have a clean working slate when switching branches. To work around this issue we use the **git stash** command.

Using git stash

The **git stash** command takes all the changes in your working copy and saves them on a clipboard. This leaves you with a clean working copy. Later, when you want to work on your feature again, you can restore your changes from the clipboard in your working copy.

To restore your saved stash you can either:

- Get the newest stash and clear it from your stash clipboard by using **git stash pop**.
- Get the specified stash but keep it saved on the clipboard by using **git stash apply <stashname>**.

Merging

When you are done working on your new feature or bug fix in an isolated branch, it is important to merge it back into the master branch. The **git merge** command allows you to take an independent line of development created by **git branch** and **integrate** it into a single branch.

To perform a merge, you need to:

- Check out the branch that you would like to use to receive the changes.
- Run the git merge command with the name of the branch you would like to merge.

```
git checkout master
git merge my-first-branch
```

The above example merges the branch, my-first-branch into the master branch. Please keep in mind that Git has updated the default branch name from 'master' to 'main'. Read more about this [here](#).



A note from the HyperionDev Team

Check out this [HyperionDev blog post](#) about Git and GitHub. It provides a simple guide to GitHub for beginners.

CLONING A REPO

You know how to initialise a Git repository from scratch, but in practice, this is very rarely how you start developing. Typically when starting as a software engineer, you'll be tasked with closing one or more issues on a specific repository. We'll get onto exactly what "closing an issue" means, but for the time being let's first focus on getting the repo cloned so that you can start developing.

For this task, you'll be using your last Python repository. If you haven't put it on GitHub yet, go ahead and do that now.

For this task we'll be pretending your repo is a team project, and you'll be playing the role of developer, code reviewer, and even project manager. To start off, pretend you're a developer seeing the project for the first time and clone the repo:

```
git clone https://github.com/[username]/[repo].git
```

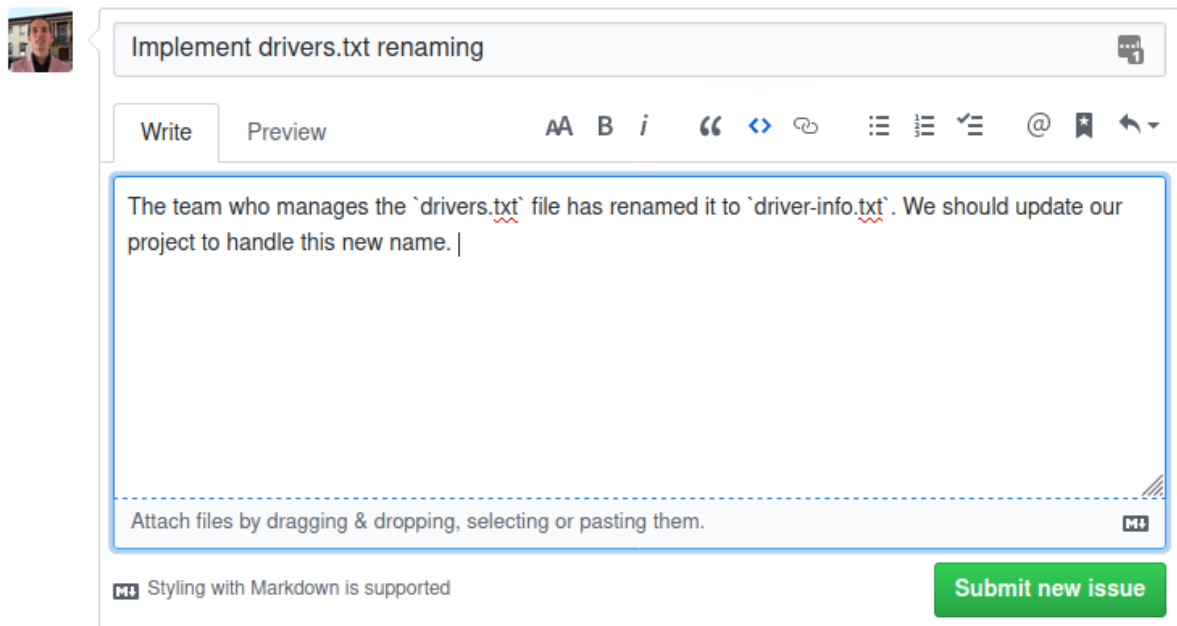
If you were at your job, at this point you'd usually spend a while compiling the code, resolving dependencies and fixing issues on your computer to get the project running. This should, however, not be a problem for you since you recently wrote that very code.

ISSUES

On GitHub (and many other platforms) an issue is a plain-English description of some improvement a codebase needs. This could be a bug fix, a performance improvement, or a new feature. Software engineers almost never do any work unless there's an issue attached to it. This makes sure everyone knows who's doing what and helps line managers understand what's being done to the codebase and what the progress is.

On the GitHub website, go to your repo and click on “Issues” (on the top bar, next to Code). You should see a plain greeting message since your repo doesn’t have any issues yet (yay!).

Pretend you’re a project manager and you just came back from a meeting where another team asked you to make a change to your project. In order to let your team know what needs to be done, you make an issue on the repo. Click on “New Issue” (top right) and fill in the following details:



Implement `drivers.txt` renaming

Write Preview AA B i “ <> 🔗 ☰ ☷ ✓ @ 📎 ↶

The team who manages the `drivers.txt` file has renamed it to `driver-info.txt`. We should update our project to handle this new name. |

Attach files by dragging & dropping, selecting or pasting them. 📎

📎 Styling with Markdown is supported

Submit new issue

When finished, click “Submit” to officially create the issue. You’ll see the issue has been given an ID (#1).

To the right of your issue description, click on the gear icon next to “Assignees”. Type in your username and click on it to assign yourself to the job. If you were a real project manager, you’d be assigning one or more of your team members.

DOING WORK ON YOUR OWN BRANCH

Now that you (as the developer) have been assigned to the issue, you should implement it. But before doing any coding, be sure to go to the directory and create a new branch. This ensures you don’t accidentally work on a co-worker’s branch and their code gets overwritten later on. You can name the branch something descriptive indicating what you’ll be doing on it (like **driver-file-fix**), or to ensure uniqueness you can name it according to the issue ID (**issue-1**). Each company will have its own policy in this respect. As you may recall, the code below creates the branch **issue-1** and switches to it at the same time.

```
git checkout -b issue-1
```

Now implement the change. As described in the issue, you need to make the program use **driver-info.txt** instead of **drivers.txt**. You would also rename the actual text file and run your program to ensure there are no problems with the new change.

Now commit your changes, giving a good description of what you changed, and then push your branch to the remote repo.

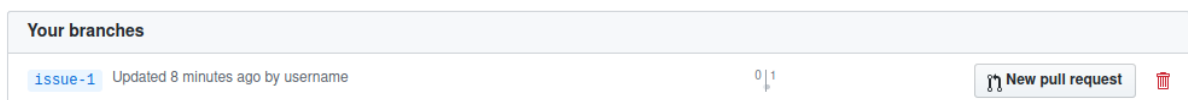
```
git commit -m "Changed program to use driver-info.txt"
git push origin issue-1
```

You can make multiple commits and pushes on your branch as needed until you are satisfied the change has successfully been implemented.

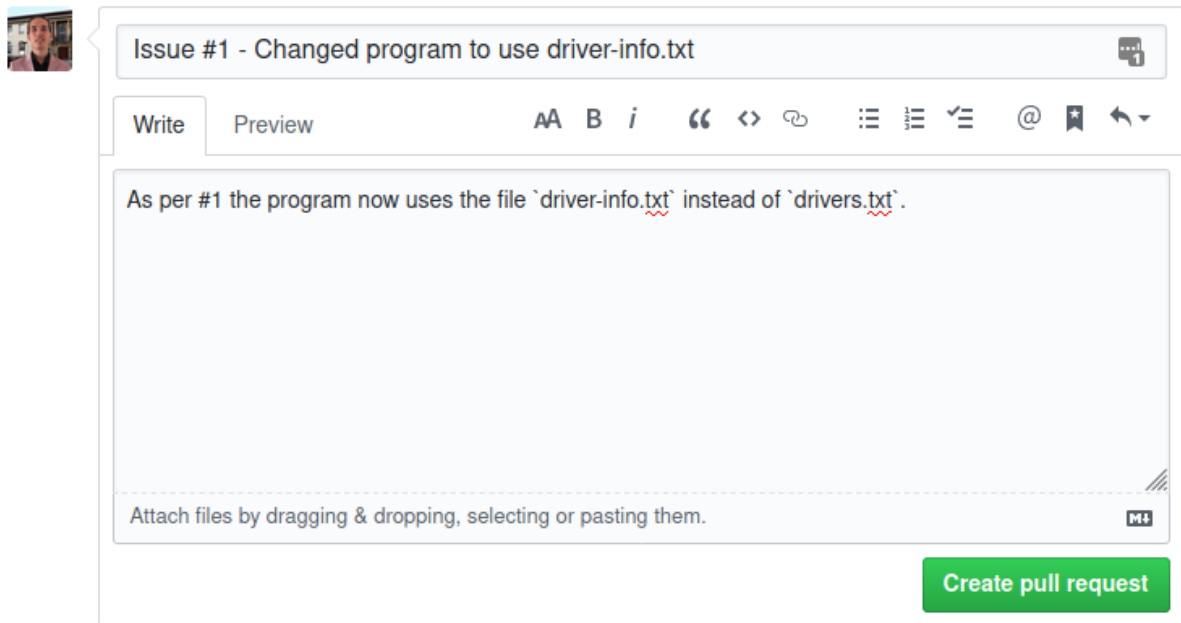
MAKING A PULL REQUEST

A pull request (**PR**) is a request asking that your branch be merged with the master branch. If accepted, this will make your code changes part of the main codebase. Depending on how the repo is set up, this might even deploy the new version of the program to your users.

It is the responsibility of the developer who made the changes (in the new branch) to create the PR. Go to your repo on GitHub, and click on “branches” (next to commits, just above your code). Find your branch in the list, and to its right, click on “New pull request”.



Write a good description of your PR, explaining everything that has been changed in it. It's good practice to mention the issue that originally motivated the changes.



Issue #1 - Changed program to use driver-info.txt

Write Preview AA B i “ < > 🔗 ☰ ☷ ☰ @ 📌 ↶

As per #1 the program now uses the file `driver-info.txt` instead of `drivers.txt`.

Attach files by dragging & dropping, selecting or pasting them. M+

Create pull request

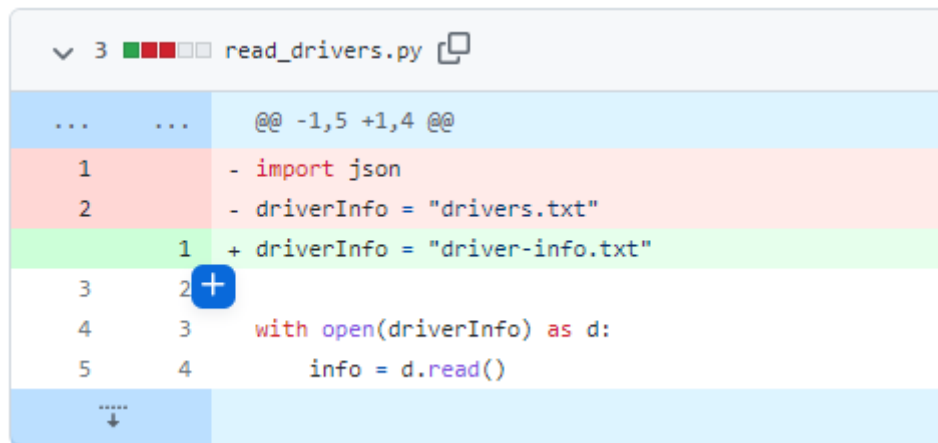
Click on “Create pull request”. The PR will be created with an ID (#2).

Imagine now you are the project manager and click on the gear icon next to “Reviewers” to assign someone to review the PR. For this task, assign yourself, but, of course, in practice, your project manager will be assigning a co-worker instead.

CODE REVIEW

Because getting a PR accepted can have so many implications for the project, it's important that your code is reviewed by another developer. This is often how easily-overlooked mistakes are spotted.

Imagine now you are a code reviewer and click on “Commits” in the PR on GitHub. Here you'll see a list of commits made to the branch the PR is for. For an all-encompassing view of changes made to the project, click on “Files changed” on the same top bar. This will show all changes made to the project in a human-readable **diff** view. Here's an example:



```
... 3 @@ -1,5 +1,4 @@
1 - import json
2 - driverInfo = "drivers.txt"
3 + driverInfo = "driver-info.txt"
4   with open(driverInfo) as d:
5     info = d.read()
```

As a reviewer, you might remark that the method name violates project standards (should be snake_cased not camelCased), and that you need to add comments. If you find a mistake in your PR, hover over the offending line and click on the blue “+” button to its left. This will let you write a comment on the code. Once submitted, this will be visible to the original developer.

You as the developer can now make the requested fixes and commit and push them to have the PR updated automatically.

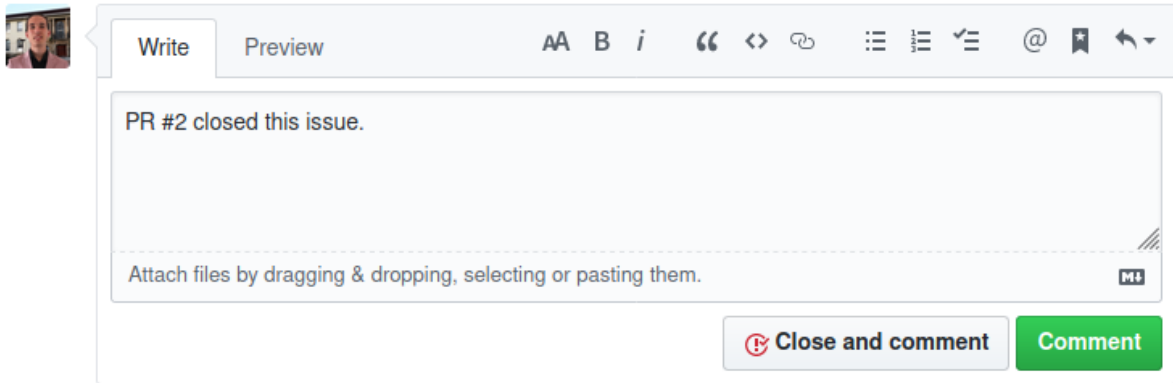
Once pushed, as the reviewer you should go back to the “Files changed” tab and click on review changes (top right) to accept the changes. GitHub restricts review usage so that PR authors can’t review their own code, so don’t worry about doing this formally right now.

FINALLY MERGING

After all this work, the branch is finally ready to be merged. Either the code reviewer or the project manager will merge the pull request. Head to your PR on GitHub, click on “Conversation” and scroll to the bottom where you should see a big green “Merge pull request” button. Note that continuous integration may pose further limits (beyond code review) before allowing a merge to happen. This may include successful building and passing of tests for your code. Setting this up is quite involved and typically not the software engineer’s responsibility. You can ignore it for now.

Click “Merge pull request” and change the description if you want to. GitHub will now automatically merge your branch into master.

Now head back to the original issue, scroll to the bottom and make some closing remarks to show that the problem has been resolved. It’s a good idea to mention the PR that officially closes the issue.



Click on “Close and comment” to officially mark the issue as closed.

To have the changes reflected on your computer, switch to the master branch and pull the update:

```
git checkout master
git pull origin master
```

Instructions

Give the accompanying Git cheatsheet document a read if you'd like to recap some useful commands and others that may come in handy as you become more proficient with Git.

Prepare by first following these steps:

- Install Git by following the instructions provided earlier in the task.
- Once you have successfully downloaded and installed Git, enter **git --version** into your terminal or command prompt to display your current version of Git.

Let's create your free GitHub account now!

Follow these steps:

- To create a free GitHub account, visit: <https://github.com/join>.

Compulsory Task 1

Follow these steps:

- Create an empty folder called `git_task_project`.
- Open your terminal or command prompt and then change directory (`cd`) to your newly created folder.
- Enter the `git init` command to Initialise your new repository.
- Enter the `git status` command and make a note of what you see. You should have a clean working directory.
- Create a new file in the `git_task_project` folder called **helloWorld.py** and write a program that prints out the message "Hello World!"
- Run the `git status` command again. You should now see that your **helloWorld.py** file is untracked.
- Enter the `git add` command followed by `helloWorld.py` to start tracking your new file.
- Once again, run the `git status` command. You should now see that your **helloWorld.py** file is tracked and staged to be committed
- Now that it is tracked, let us change the file **helloWorld.py**. Change the message printed out by the program to "Git is awesome!"
- Run `git status` again. You should see that **helloWorld.py** appears under a section called "Changes not staged for commit". This means that the file is tracked but has been modified and not yet staged.
- Take a screenshot of this change and title it: **git_modified.pdf** or **git_modified.jpeg**. Include it as part of your submission.
- To stage your file, simply run `git add` again.
- If you run `git status` again you should see that it is once again staged for your next commit.
- You can now commit your changes by running the `git commit -m` command. Remember to enter a suitable commit message after the `-m` switch.
- Running the `git status` command should show a clean working directory once again.
- Now run the `git log` command. You should see your commit listed.
- Take a screenshot of the output and add it to the task folder. Name the submitted image file: **git_log.pdf** or **git_log.jpeg**.

Compulsory Task 2

Follow these steps:

- Open your terminal or command prompt and change directory (**cd**) to the folder `git_task_project` created above.
- Create a new branch called `issue-1` using the **git branch** command.
- Switch to your new `issue-1` branch by using the **git checkout** command.
- Once you are on the `issue-1` branch, change the `helloWorld.py` file. Modify your program to accept input from the user and then print out the inputted data.
- Add and commit your changes.
- Check out the master branch and use the **git merge** command to merge branches.
- Take a screenshot of the output after running the **git merge** command and upload it to the task folder.
- Name the submitted image file: **git_merge.pdf** or **git_merge.jpeg**.

Compulsory Task 3

- Pick any one of your GitHub repos.
- Create 2 issues for things you think could be improved. Ideas for improvements include making new methods, adding constants, renaming variables and functions, or adding comments.
- For each issue:
 - Create a branch with a meaningful name.
 - Implement the changes required by the issue.
 - Commit and push your work.
 - Create a PR for your changes.
 - Merge in the PR and close the issue.
- In a text file called **repo.txt**, paste the link to your repo. Add the file to this task's folder.

Completed the task(s)?

Ask an expert code reviewer to review your work!

[Review work](#)



Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

[Click here](#) to share your thoughts anonymously.

