HyperionDev

Git Workflows

Task

Visit our website

Introduction

In a past task, you saw how to manage different versions of the same codebase. In this lesson, we will look at how Git helps ensure multiple developers can work simultaneously on the same project without getting in each other's way. One day you will be a developer working on a team project, so it is important to know how it is done.

Cloning a repository

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 programmer, you will be tasked with closing one or more issues on a specific repository. We will get onto exactly what 'closing an issue' means, but for the time being let's first focus on getting the repository cloned so that you can start developing.

Cloning a repository pulls a complete copy of the remote repository to your local system. First, navigate to the folder where you would like the local copy of the repository to be stored. For example:

```
cd learning-to-clone
```

Otherwise, create a folder and then navigate to it. For example:

```
mkdir learning-to-clone
cd learning-to-clone
```

To clone a repository, enter 'git clone [repository_url]' into the terminal. This would look something like:

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

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.
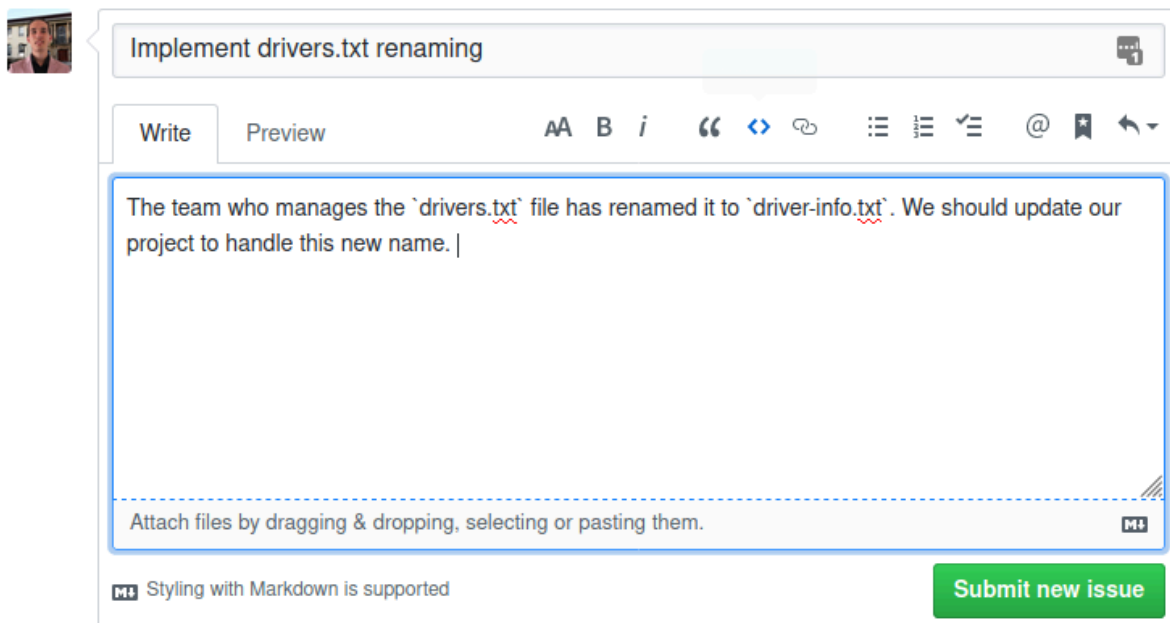
Having successfully cloned a repository, let's shift gears and explore how to contribute to the project by addressing issues raised by collaborators.

Addressing 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. Developers seldom do any work unless there's an issue attached to it. This ensures everyone knows who's doing what and helps line managers understand what's being done to the codebase and the progress.

Let's pretend you are working on a team project. As the developer, you would first clone the repository. Then, you would usually spend a while resolving dependencies and fixing any other problems on your computer to get the project running smoothly. Once you have the project up and running you are ready to address any issues listed on GitHub. To find the issues, you would go to the repository on GitHub and click on "Issues" (on the top bar, next to Code).

Your project manager has just returned from a meeting where another team has asked to make a change to the project you are working on. To let your team know what needs to be done, the project manager makes an issue on the repository. They click on "New Issue" (top right) and fill in the following details:

When finished, they click "Submit" to officially create the issue. You will see the issue has been given an ID (**#1**).

To the right of the issue description, they click on the gear icon next to "Assignees". They type in your username and click on it to assign you to the job.

To work on this update you will need to know about branching in Git.

Branching

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. It is common for several developers to share and work on the same source code. Since different developers will often 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. You do not want to accidentally overwrite your co-worker's code!

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 that of others. By branching, you diverge from the main line of development and continue to work without introducing errors or disrupting the main line.

Branches are crucial for developing new features or fixing bugs. Creating a branch allows you to isolate changes, keep unstable code out of the main codebase, and refine your work history before merging it back into the main branch.
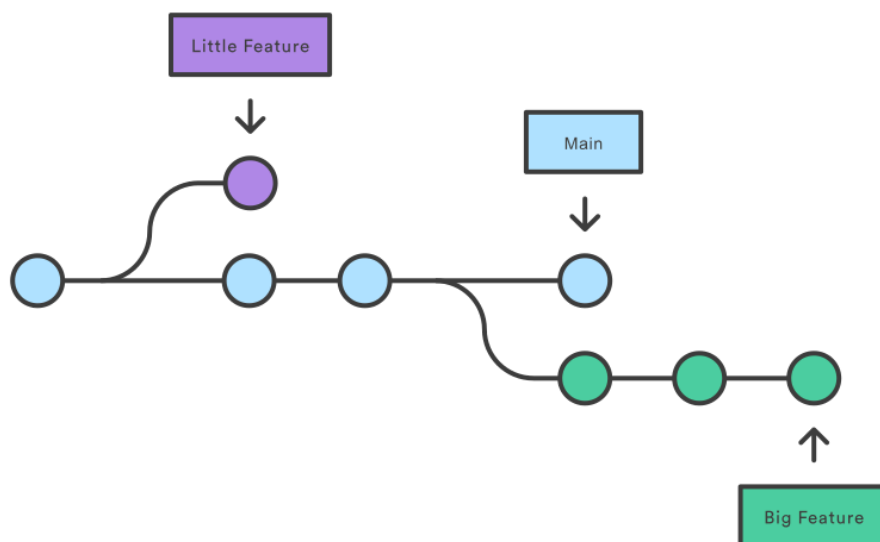
A repository with two isolated lines of development (Atlassian, n.d.)

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 free from dubious code.

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

Take note

You may encounter situations where the main branch is called the master branch. However, new repositories will have a main branch as the default. Explore the **renaming repository** on GitHub if you would like to rename your default branch.

The **HEAD** in Git represents the most recent commit in the current branch, essentially the branch's tip. By default, for a new repository, the HEAD points to the main branch. Switching the HEAD changes your current branch, effectively moving you to a different branch or commit. You can check where the HEAD is pointing using the `git status` command, which shows this information in the first line of output:

```
> git status
On branch main
```

Creating and switching branches

To create a new branch, use the `git branch` command, followed by the name of your branch. You can name the branch something descriptive indicating what you will 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. For example:

```
> git branch issue-1
```

Note: Branching requires **at least one commit** in the repository's history. If you try to create a branch in an empty repository without any commits, you will encounter an error.

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 issue-1
```

Using this command moves the `HEAD` to the `issue-1` branch. You can check this by entering the `git status` command.

```
> git status
On branch issue-1
```

To create a branch and switch to it at the same time, you can run the `git checkout` command with a `-b` switch instead of running the `git branch` and `git checkout` commands separately. For example:

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

Now you can 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.

Saving changes temporarily

When you make a commit, your changes are saved permanently. However, if you need to save local changes temporarily – such as when you're working on a feature but need to quickly fix a bug – you can switch back to the main branch without including your feature changes. Git lets you do this easily.

Before you switch to the main 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. This is because it is better to have a clean working slate when switching branches. To work around this issue we use the `git stash` command.

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. You can check this by running git status:

```
> git status
On branch issue-1
nothing to commit, working tree clean
```

Later, when you want to work on your feature again, you can restore your changes from the clipboard to 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>`. This is useful if you want to copy the same stashed changes to different branches.

Now, let's get back to what happens after you have implemented the changes requested in issue #1 and pushed your branch to the remote repository.
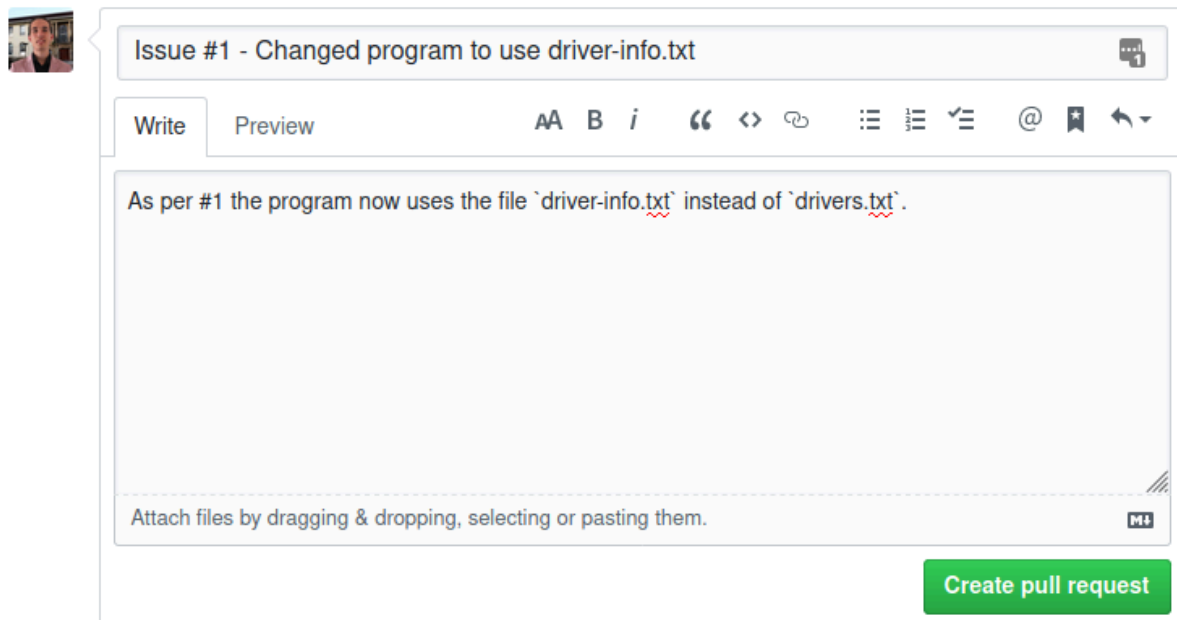
Making a pull request

To merge your branch with the main branch, you will need to make a pull request (**PR**). If accepted, this will make your code changes part of the main codebase. Depending on how the repository 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. You can do this by going to your repo on GitHub and clicking on "branches" (next to commits, just above your code). Find your branch in the list, and to its right, click on "New pull request".
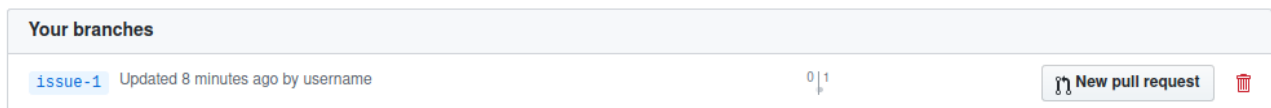
Write a good description of your PR, explaining everything that has been changed in it. It is good practice to mention the issue that originally motivated the changes.
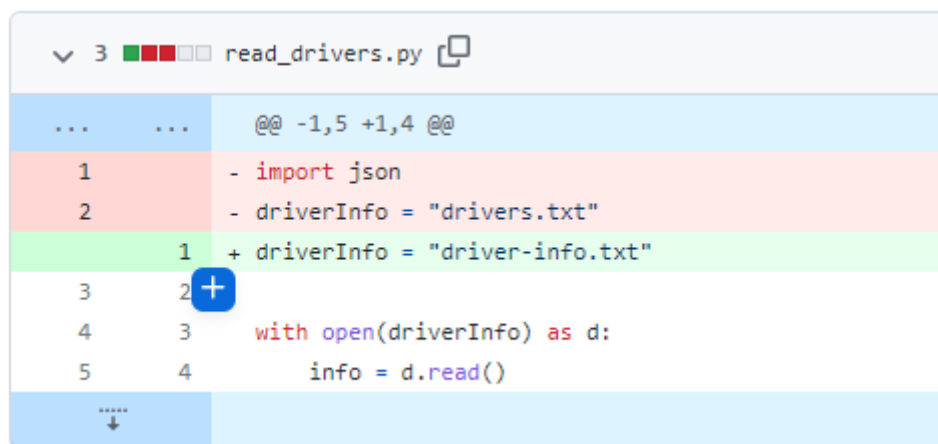

Click on 'Create pull request'. The PR will be created with an ID (**#2**).

The project manager then clicks on the gear icon next to "Reviewers" to assign someone in your team to review the PR.

Code reviewing

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 mistakes are spotted.

Imagine now you are a code reviewer and click on "Commits" in the PR on GitHub. Here you will 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 ■■■□□ read_drivers.py ⎘

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

As a reviewer, you might remark that the variable name `driverInfo` violates PEP8 style guidelines (should be snake_cased not camelCased) and that comments need to be added to document your code.

If you find a mistake, 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 cannot review their own code, so don't worry about doing this formally right now.

Merging

After all this work, the branch is finally ready to be merged. Either the code reviewer or the project manager merges the pull request. They would head to your PR on GitHub, click on "Conversation" and scroll to the bottom where there is a green "Merge pull request" button. After clicking on "Merge pull request", GitHub automatically merges your branch into the main branch.

Note that if your team uses a CI/CD pipeline approach, additional automated quality checks such as unit tests may be performed before allowing a merge to happen. Setting this up is quite involved and typically not the developer's responsibility. You can ignore it for now.

Extra resource

Explore the **power of continuous integration and continuous deployment (CI/CD)**, a streamlined approach to software development. By automating the building, testing, and deployment processes, CI/CD ensures faster delivery of high-quality applications. This continuous cycle minimises human error, accelerates time-to-market, and empowers teams to innovate at speed.

Now it is time to close the original issue. You would typically make some closing remarks to show that the problem has been resolved. It is a good idea to mention the PR that officially closes the issue.

Clicking on "Close and comment" officially marks the issue as closed.

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

HyperionDev

9

```
git checkout main
git pull origin main
```

⚠ Take note

Git can be useful whether you are coding an app on your own, or if you are a company with thousands of employees. But after today's task, you should understand how it is used differently in the latter case. When programming on your own, you do not have to create issues or merge in pull requests. This is far too slow a process if you're just one person. Especially for this course, it is perfectly acceptable to complete a task without making one commit.

For capstone projects, it is recommended, however, that you make use of Git and push your work to GitHub. If you want to impress future employers it is a good idea to make use of multiple branches, issues and PRs. But this is at your discretion.

Simple merging

If you are working alone on a project you can skip the pull request and code review steps to merge your changes back into the main 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 main
git merge issue-1
```

The above example merges the branch called `issue-1` into the `main` branch.

📖 Extra resource

While specific policies and procedures may vary across organisations, understanding **industry best practices** can provide a solid foundation for a successful start.

Issue and pull request workflow

Following this workflow will help you establish a structured process for managing new features or bugs in your projects using GitHub issues and pull requests. It ensures your projects remain organised, keeps the `main` branch clean, and showcases professional development practices.

1. **Set up your local repository:**

    ○ Clone the repository using the `git clone` command to clone your GitHub repository to your local machine:

    ```
    git clone repository-url
    ```

    ○ Navigate to the repository folder:

    ```
    cd repository-folder
    ```

    ○ Set up the remote link (if not already configured), and ensure your local repository is connected to the GitHub repository by verifying the remote link:

    ```
    git remote -v
    ```

    ○ If no remote link is found, add it using:

    ```
    git remote add origin repository-url
    ```

2. **Open an issue**:

    Create an issue in your GitHub repository by navigating to the Issues tab, clicking "New issue," and providing a clear title and description (e.g., "Add More Comments to Code").

3. **Create a local branch for the issue**:

    Before working on the new task, create a new branch to isolate your changes. Start by updating your local repository with changes from the `main` branch:

    ```
    git pull origin main
    ```

    Then create a new branch with a descriptive name:

```
git checkout -b feature/add-comments
```

4. **Make the changes locally**:

Work on the task in the new branch (e.g., adding comments). Once done, stage and commit the changes:

```
git add .
git commit -m "Add comments to improve code readability (closes #5)"
```

Referencing the issue number in the commit message (e.g. `closes #5`) links the commit to the issue automatically on GitHub.

5. **Push the branch to GitHub**:

After committing the changes, push the created branch to the remote repository to help make the branch and your changes available on GitHub:

```
git push origin feature/add-comments
```

Pushing the branch in such a manner would create a remote copy of your branch on GitHub, which helps to prepare it for a pull request.

6. **Open a pull request**:

Once you have pushed your branch to GitHub you will have two options to create a pull request:

○   **Compare and pull request prompt**: GitHub may display a banner at the top of the repository page, presenting you with the option to compare and create a pull request for the newly pushed branch. If this appears, click "**Compare & pull request**" to proceed.

○   **Pull requests tab**: Go to the Pull Requests tab and click "**New pull request**." Select your source branch (feature/add-comments) and target branch (main). Ensure the arrow points from your feature branch to main.

In both cases, be sure to add a descriptive title and description to your pull request, referencing the issue number (e.g. "Fixes #5"). Once ready, click "**Create pull request**" to submit the pull request for review.

7. **Review, merge, and close the issue**:

○   After submitting the pull request, review the changes in the "**Files Changed**" tab to ensure only the intended modifications are included and no unintended changes have been made.

HyperionDev 12

- If working alone, use this step to double-check your work. Add comments for any further refinements. Once satisfied, click "**Merge pull request**" to integrate the changes, then delete the feature branch to keep the repository clean.

- Once merged, GitHub will automatically close the associated issue if the commit or pull request references the issue number with keywords like "closes #5" or "fixes #5". If not, manually close the issue in the Issues tab. For more details, refer to **GitHub's documentation** on linking pull requests to issues.

Instructions

Feel free to refer to the **Git cheatsheet** in the folder for this task as needed for this or any future tasks in which you use Git.

Practical task

Imagine you are working on an app for gardening enthusiasts around the world. So far the app provides gardening tips and advice based on the month and the season. You will implement improvements to the code provided.

1. Create a public repository on GitHub called garden-app

2. Upload the garden_advice.py or garden_advice.js file to the repository.

   - The uploaded file (.py or .js) includes TODO comments suggesting potential improvements, such as creating functions, adding documentation, or replacing hardcoded values.

3. Clone the garden-app repository to your local machine.

4. Review the uploaded file locally and identify improvements based on the TODO comments. These comments can guide you in creating GitHub issues for tasks like refactoring code or enhancing comments.

5. Create two issues on GitHub based on the suggested improvements or your own ideas.

6. Follow the Issue and pull request workflow:

- For each issue:
  - Create a branch locally for the issue.
  - Implement the changes in the new branch.
  - Commit and push your changes to GitHub.
  - Open a pull request from your branch to the main branch.
  - Review and merge the pull request.
  - Close the issue.

7. Refer to the detailed Issue and pull request workflow section for precise steps on creating branches, committing changes, and managing pull requests.
8. Create a text file named repo.txt containing the link to your GitHub repository, then save the file.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Challenge

Use this opportunity to extend yourself by completing an optional challenge activity.

Contributing to open source is a great way to help out software initiatives. It shows the developer community that you care and it shows future employers that you write quality code that's on-standard for open source projects.
Find a project you would like to contribute to on GitHub. The explore page usually has some ideas if you're browsing: **https://github.com/explore**.

1. Look at the open issues for your chosen repository.

2. Read through their descriptions and pick the one you think you can tackle. Some issues (especially on big projects) will be tagged with the label "Good first issue", which is a great place to start.

3. Fork the repository. This will duplicate the repo to your account. This allows you to make changes (and eventually a PR) without altering the original project. It's standard procedure in open source. Learn more about **forking a repository**.

4. Clone your newly forked repo and create a new branch for the issue you are tackling.

5. Most open-source projects have a "contributing.md" file that specifies how you should go about making changes to the project. Be sure to read through it before starting your coding.

6. Make the required changes and test your code to make sure it works. This could take a while. Remember to commit and push along the way.

7. Create a pull request from your branch to the original repo. Be descriptive and mention the issue you are resolving.

If all goes well, your PR will be accepted and you will have contributed to an open-source project. It is likely that a more familiar contributor to the project will review your code and make suggestions for changes. If this happens, be sure to implement the suggestions and commit and push them again to update your PR.

It may happen that your PR gets rejected if, for instance, the project owners do not approve of the way you tried to resolve the issue. If this happens, do not be discouraged. Pick a different issue, or a different project and try again.
Please feel free to book a mentor call if you'd like a mentor to review your challenge code.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

Atlassian. (n.d.). Git Branch. **https://www.atlassian.com/git/tutorials/using-branches**