# Git Notes

Here I'm going to depict my Git learning as I get used to it, and thus have a record of the many processes in order to consult it when necessary to improve the understanding.

**# Starting elements and basic workflow**

First thing after the installation and account creation in any of the web repository remote server (like GitHub or GitLab) is to enable the communication between local computer (local host) and the remote server through SSH connection. To do that, is needed to generate a SSH key which is pretty simple:

- In Git bash, at home directory, write the command for "ssh-keygen" and it will generate a couple of files storing SSH keys each one. Both are "id_rsa" but one has .pub extension, this one contains the public key
- This public key is used by the remote server to identify local host and allow data exchange. Find the SSH section in the website of the selected remote server and add the public key.
- When done, it will be possible to download and upload source code files (or any other type in fact) from the terminal or bash using the "git clone" and "git push" commands respectively

Now, let's see how to upload files from the local host to remote server for first time.  If an attempt is made to file upload, it's going to fail. The wrong thing is some configuration is missing. The remote server can be able to identify the local host but this one doesn't knows about it. The configuration just consist of adding a remote server with

```
git remote add origin <REMOTE_SERVER>
```

The remote server is the repository created to store a determined project, that is, a set of files and directories and is indicated with its associated URL. The following is the example of my repo for some project:

```
git remote add origin git@github.com:EDR-9/LogicTest
```

What happen if a wrong or invalid remote server was entered? It seems that is not severe, just delete it. First, see the current origin is added with

```
git remote -v
```

Delete that origin with

```
git remote rm origin
```

When it's done, add the right origin (remote server identification) and there should not be problem with uploading or "pushing" the files to the remote server.

Once these configurations are ready, shift the attention to when we want to upload files. The basic knowledge about how Git internally works is already covered, so the goal is to understand how work flow is done. Suppose some project has been finished (at least in an initial form) and is ready to be uploaded. We need to add files to stage in order to start the tracking of the changes for each one. This is achieved with

```
git add <FILE> or git add <FILES> or git add .
```

Where FILE is the one to be tracked or FILES if several need to. The point "." is to indicate that ALL files in the project will be tracked.
After this, and ensuring that files are not going to change (for now), the commit is the next step because we're somehow "committing ourselves" that delivery is finished and ready to be uploaded.

To commit

```
git commit -m <COMMIT_MESSAGE>
```

As we see, a commit message is required just to briefly identify what we're doing in this delivery. When doing commit, Git assigns a tag or ID to the commit to keep a track of it. Now, the next step is file uploading is and is done through

```
git push origin master
```

This assumes that changes are coming from the master (main branch), but recall we can create any branch and make changes from there and then merge it to master. When changes are coming from a different branch the push is

```
git push origin <BRANCH>
```

We'll discuss about it later. One important thing is to resort to documentation and see all the available options a single command has.

**# Creating project branches**
The concept of branches is fundamental in Git, as we know, it consists of creating alternative paths within the project with the possibility of making changes in each one without altering the "original" copy at master (main branch); once some changes were made in a given branch, they can be merged to the master
To create a new branch and change the HEAD from the current one to this newest we simply write

```
git checkout -b <NEW_BRANCH>
```

To comeback to the previous branch (and change to any branch in fact) just write

```
git checkout <BRANCH>
```

The following is a discussion with ChatGPT asking it about the ways to make changes from a given branch and their implications

Now I have a question related to Git. If I create a branch and I make some changes there, is necessary to make push in that branch in order to merge to master or just merge directly?
Let's dive into your Git question about branches and merging.

### Do You Need to Push a Branch Before Merging?

No, **it’s not necessary to push your branch** before you merge it into `master` (or any other branch) locally. You can merge a branch directly into `master` **without pushing** it, as long as you're working on your local repository and haven't pushed any changes to the remote yet.

However, the scenario can vary depending on whether you are working **locally** or **remotely**, and whether you want to keep your remote repository updated.

### Two Main Scenarios for Merging:

#### **Scenario 1: Merging Locally (Without Pushing First)**

If you’re working locally and want to merge your branch into `master`, you don’t need to push the branch to the remote before doing the merge. Here’s how you can do it:

1. **Switch to the `master` branch:**
   ```bash
   git checkout master
   ```
2. **Merge the feature branch into `master`:**
   ```bash
   git merge my-feature-branch
   ```

~~~
3. **Resolve any merge conflicts** (if any appear).

4. If everything looks good and the merge is successful, you can either:
  - **Push `master` to the remote** to update the remote repository:
    ```bash
    git push origin master
    ~~~
  - Optionally, **delete the feature branch** (locally) if you no longer need it:
    ~~~bash
    git branch -d my-feature-branch     ```

So, in this scenario, **you do not need to push the feature branch first**. The important part is to push `master` (or whatever branch you merged into) after the merge.

#### **Scenario 2: Merging with Remote (Push First or Use Pull Requests)**

If you're working with a **remote repository** (e.g., on GitHub, GitLab, Bitbucket) and you want to ensure your work is safely stored remotely, **you need to push your branch to the remote repository before merging**. This is especially true if you're working with a team and using **pull requests** or **merge requests**.

Here's how it works:

1. **Push your feature branch to the remote:**
  ```bash
  git push origin my-feature-branch
  ~~~
2. **Create a Pull Request (PR)** or **Merge Request (MR)** on your Git hosting platform (GitHub, GitLab, etc.).
  This is a good practice in collaborative environments, as it allows others to review your code before merging into `master`.

3. **Merge the PR/MR on the platform** (GitHub, GitLab, etc.).

4. If you merged directly on the platform, you may want to **update your local `master`**:
  ```bash
  git checkout master
  git pull origin master
  ```
In this scenario, **pushing the branch first** is necessary because you need the remote version of the branch to be available for review or merging.

### Recap

- **Merging Locally**: If you're only working locally and you want to merge your feature branch into `master`, you **don't need to push** your feature branch first. You can merge directly, and then push `master` to the remote.

- **Merging Remotely (with PR/MR)**: If you're working with others or using a code hosting platform, you'll **need to push your branch first** so that others can review it or you can create a Pull Request (PR) or Merge Request (MR) before merging it into `master`.

#### Additional Notes:

- If your `master` branch has diverged from the remote (for example, someone else pushed changes to `master` after you created your feature branch), you may need to **pull the latest changes** from `master` before merging:
  ```bash
  git checkout master
  git pull origin master  # Make sure you have the latest changes from remote
  ```

- After merging, it's always a good idea to **push your changes** to keep the remote repository up-to-date.

Let me know if you'd like further clarification or examples!

Ok, I see, and in the first scenario there's no trace of the created branch right? Everything ends up being in master

Exactly! In the first scenario, where you merge your feature branch into `master` locally and **do not push the feature branch to the remote**, the feature branch **essentially disappears** after the merge. Here's why:

### How It Works in the First Scenario:

1. **Local Branch (Feature Branch) Merged into `master`:**
When you merge your feature branch (e.g., `my-feature-branch`) into `master`, Git will integrate the changes from your feature branch into `master`. The history and changes from the feature branch will be part of `master` going forward.

2. **Feature Branch Local Only (No Remote Changes):**
If you haven't pushed the feature branch to the remote repository (GitHub, GitLab, etc.), then there is no trace of that feature branch on the remote. It's just a local branch that was merged into `master` locally.

3. **Result in `master`:**
After the merge, the changes from the feature branch are now part of `master`. When you push `master` to the remote repository, only the `master` branch will be updated, and the feature branch will not exist on the remote unless you explicitly push it.

4. **Deleting the Feature Branch Locally:**
After the merge, you can choose to delete the feature branch locally (using `git branch -d my-feature-branch`), especially if you're done with it. This keeps your local repository clean. The changes you made in the feature branch are now in `master`, so the feature branch is no longer necessary.

### Summary

- **Merging Locally**: When you merge your feature branch into `master` locally, and you **don't push** the feature branch, the only thing that remains in the repository is the merged changes in `master`.

- **No Remote Trace of Feature Branch**: Since the feature branch was never pushed to the remote repository, there's no trace of it there. The feature branch only existed locally and was merged into `master`.

- **Master Branch Updated**: The `master` branch will now have the feature branch's changes as part of its history.

- **Clean-Up**: You can delete the feature branch locally (`git branch -d my-feature-branch`), but that's optional—it's just for cleanliness.

So, yes, after the merge, everything ends up in `master`, and the feature branch is not needed unless you explicitly push it to the remote or keep it locally for other purposes. The feature branch won't appear anywhere else unless you push it.

Let me know if you'd like to dive deeper into any part of this!

As we can see, in order to make a branch relevant and available at the remote server is required to do the "push" of that branch before merging it with master. Furthermore, there's a remarkable topic about the repo updating with the "pull" command, the difference is in "git pull" and "git pull origin naster":

You mentioned that if new changes are made after created my branch, then pull the latest changes from master with git pull origin master. So, is not enough with just git pull?

Good question! Let me explain the difference between `git pull` and `git pull origin master` in more detail.

### The Difference Between `git pull` and `git pull origin master`

- **`git pull`**:
This command updates your current branch by fetching changes from the **remote tracking branch** that corresponds to your local branch. For example, if you are on `master` and you run `git pull`, Git will fetch changes from the remote `master` branch (usually `origin/master`) and merge those changes into your local `master` branch.

- **`git pull origin master`**:
This command explicitly tells Git to **fetch changes** from the remote repository (in this case, `origin`) and then **merge** them into your **local `master` branch**. It's specifying exactly which branch to pull from (`origin/master`), rather than relying on the default tracking branch.

So, here's how they apply in different scenarios:

### 1. **When You're On `master` and Just Want to Update It:**
If you're already on the `master` branch and you want to pull the latest changes from the remote `master` branch (e.g., from `origin`), then **just running** `git pull` will be sufficient:

```bash
git pull
```

This will automatically fetch the changes from `origin/master` (assuming `origin` is the default remote) and merge them into your local `master` branch.

### 2. **When You’re On a Different Branch (e.g., `my-feature-branch`):**
  If you’re on a branch other than `master` (like `my-feature-branch`) and you want to get the latest changes from `master` into your branch, **you’ll need to specify `master`** because `git pull` will default to fetching and merging changes from your current branch’s remote counterpart.

  Here’s what you’d do in this case:

  1. **Switch to `master`:**
    ```bash
    git checkout master
    ```
  2. **Pull the latest changes into `master`**:
    ```bash
    git pull origin master
    ```
    This will fetch and merge the latest changes from `origin/master` into your local `master` branch.

  Alternatively, if you **don’t want to switch branches**, you can also pull the latest changes from `master` directly into your feature branch by running:
  ```bash
  git pull origin master
  ```
  While you’re on your feature branch. This will fetch changes from `origin/master` and merge them into your current branch (`my-feature-branch` in this case). This is especially useful if you want to keep your feature branch up to date with `master` while working on it.

### 3. **Why Specify `origin master`?**
  When you specify `git pull origin master`, you are explicitly pulling changes from the `master` branch of the `origin` remote.

  This can be useful for clarity and avoiding confusion, especially in the following cases:
  - **Multiple remotes**: If you have more than one remote (e.g., `origin` and `upstream`), you may want to specify which remote’s branch you want to pull from.
  - **Branch Tracking**: If you're on a branch that doesn’t have an upstream branch set (or it's set to a different branch), using `git pull` without arguments will fail or might pull from a branch you didn’t intend.

### Example Workflow for Merging Latest Changes from `master` into Your Feature Branch

Let’s say you’re working on a feature branch (`my-feature-branch`) and you want to ensure your branch is up to date with the latest changes from `master` before you merge:

1. **Checkout `master`**:
  ```bash
  git checkout master
  ```
2. **Pull the latest changes from `origin/master`**:
  ```bash
  git pull origin master
  ```
  This fetches and merges the changes into your local `master`.

3. **Switch back to your feature branch**:
  ```bash
  git checkout my-feature-branch
  ```
4. **Merge the latest `master` into your feature branch**:
  ```bash
  git merge master
  ```
  This will bring the changes you just pulled from `master` into your feature branch.

Alternatively, you could skip the `checkout master` step and pull directly into your feature branch using:
```bash
git pull origin master
```
This will fetch the latest changes from `origin/master` and merge them directly into your current branch (in this case, `my-feature-branch`).

### Summary:

- **`git pull`**: Fetches and merges changes from the **upstream branch** of your **current branch**. If you're on `master`, `git pull` fetches and merges from `origin/master`.

- **`git pull origin master`**: Explicitly fetches and merges changes from the **remote `origin` repository's `master` branch** into your current branch, regardless of which branch you're on.

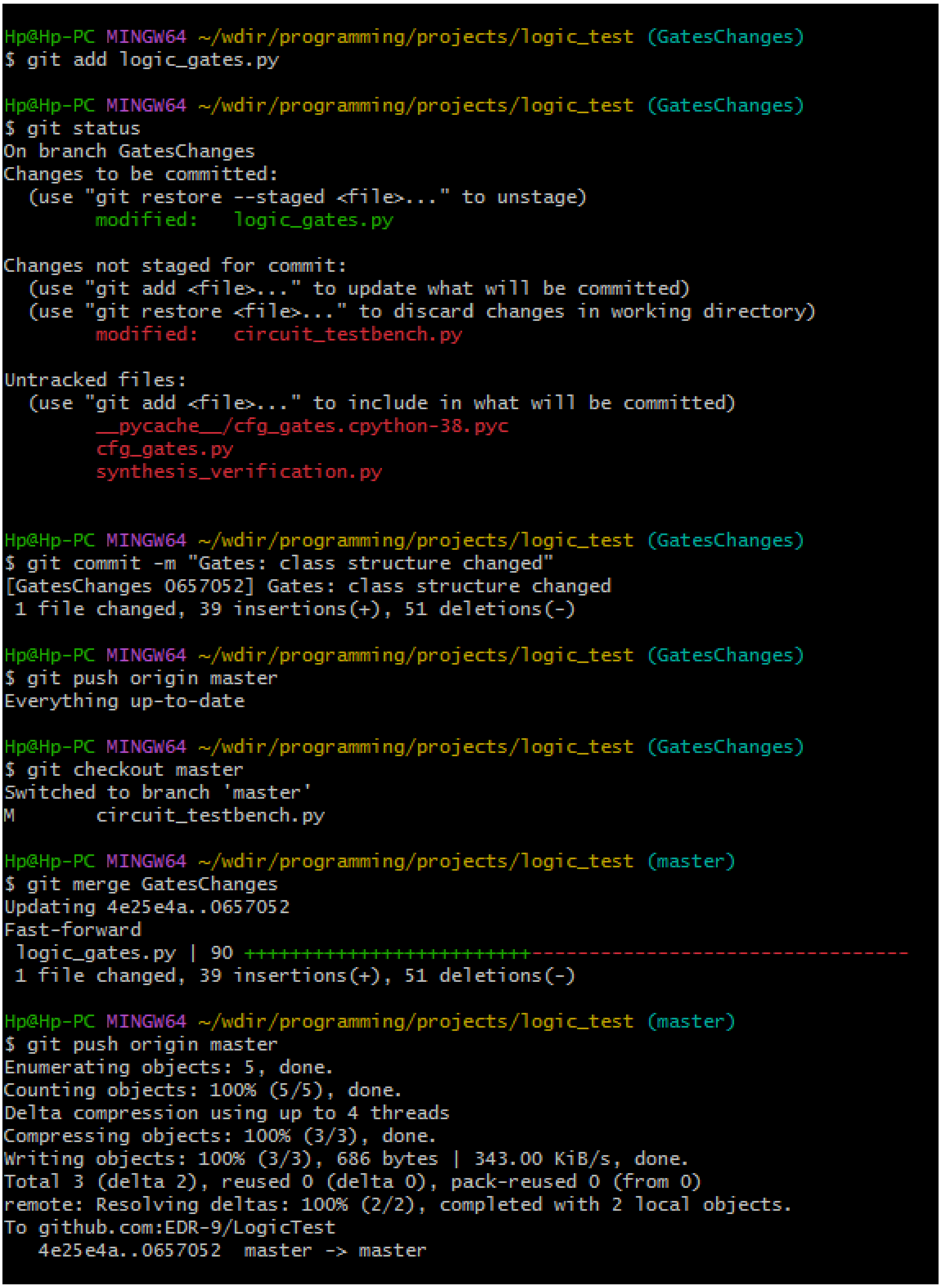
So, if you’re already on `master`, **`git pull` is enough** to get the latest updates. If you're on another branch and want to update from `master`, use **`git pull origin master`** or **`git merge master`** after switching to the branch you want to update.

Let me know if anything is unclear or if you'd like further details!

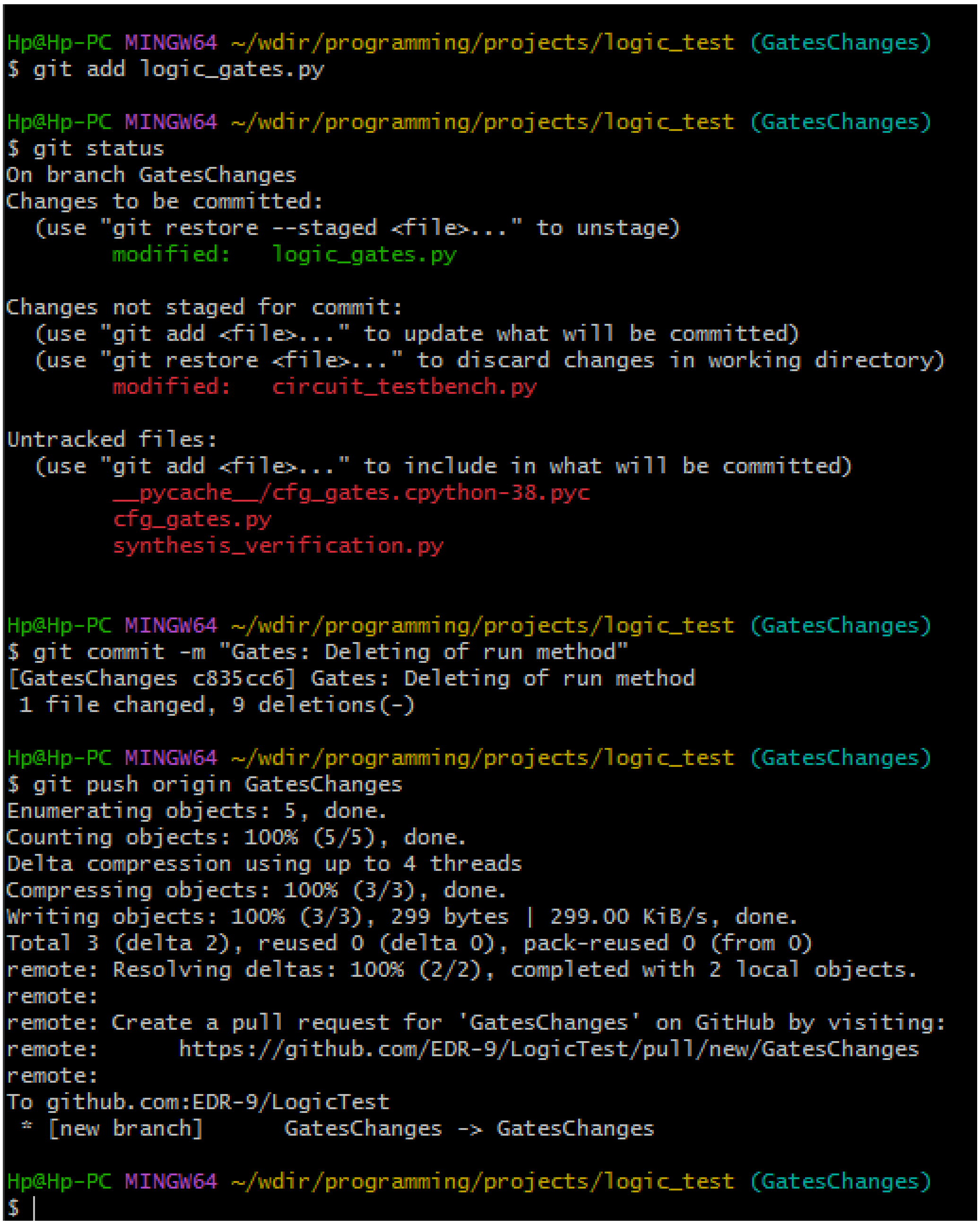
SsWhat is exactly a pull request? Isn't pull related to downloading? Also, if "pull" is related to downloading, just like "clone", what is the difference between them?

When a pull request is done, is merge no longer necessary?

Next, illustration of the branching and merching workflows:



```
Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (GatesChanges)
$ git add logic_gates.py

Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (GatesChanges)
$ git status
On branch GatesChanges
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   logic_gates.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   circuit_testbench.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        __pycache__/cfg_gates.cpython-38.pyc
        cfg_gates.py
        synthesis_verification.py


Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (GatesChanges)
$ git commit -m "Gates: class structure changed"
[GatesChanges 0657052] Gates: class structure changed
 1 file changed, 39 insertions(+), 51 deletions(-)

Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (GatesChanges)
$ git push origin master
Everything up-to-date

Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (GatesChanges)
$ git checkout master
Switched to branch 'master'
M       circuit_testbench.py

Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (master)
$ git merge GatesChanges
Updating 4e25e4a..0657052
Fast-forward
 logic_gates.py | 90 +++++++++++++++++++++++++---------------------------------
 1 file changed, 39 insertions(+), 51 deletions(-)

Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (master)
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 686 bytes | 343.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:EDR-9/LogicTest
   4e25e4a..0657052  master -> master
```



```
Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (GatesChanges)
$ git add logic_gates.py

Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (GatesChanges)
$ git status
On branch GatesChanges
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   logic_gates.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   circuit_testbench.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        __pycache__/cfg_gates.cpython-38.pyc
        cfg_gates.py
        synthesis_verification.py


Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (GatesChanges)
$ git commit -m "Gates: Deleting of run method"
[GatesChanges c835cc6] Gates: Deleting of run method
 1 file changed, 9 deletions(-)

Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (GatesChanges)
$ git push origin GatesChanges
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 299 bytes | 299.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: Create a pull request for 'GatesChanges' on GitHub by visiting:
remote:      https://github.com/EDR-9/LogicTest/pull/new/GatesChanges
remote:
To github.com:EDR-9/LogicTest
 * [new branch]      GatesChanges -> GatesChanges

Hp@Hp-PC MINGW64 ~/wdir/programming/projects/logic_test (GatesChanges)
$ |
```

## More discussion with ChatGPT about Git nuances

I'm working with Git and I have some questions: what is exactly a pull request? Isn't pull related to downloading? Also, if "pull" is related to downloading, just like "clone", what is the difference between them?. Another thing, when a pull request is done, is merge no longer necessary?

Great questions! Let's break them down one by one to clarify the concepts around Git and pull requests.

### 1. **What is a Pull Request?**

A **pull request (PR)** is a way to propose changes to a repository, typically when you're working in a collaborative environment. It allows you to submit your changes (often in a branch) to be reviewed and merged into another branch (usually the `main` or `master` branch).

- **PR Process:** You make changes in a separate branch, push those changes to a remote repository, and then create a pull request. The pull request provides a discussion space for reviewing the changes, testing, and collaboration before the changes are merged into the main branch.
- **Why "Pull"?** The term comes from the fact that you're asking the repository maintainers to **pull** your changes into their repository. It's like you're saying, "Here's a set of changes—please pull them into your codebase."

### 2. **Isn't "Pull" Related to Downloading?**

Yes, **pull** is indeed related to downloading in Git, but not in the exact same way you might think. Here's the distinction:

- **`git pull`**: When you run `git pull`, Git fetches the latest changes from a remote repository (i.e., it downloads any new commits) and merges them into your local branch. So, in a way, it **downloads** and integrates new changes into your local branch.

- **`git clone`**: On the other hand, `git clone` is used when you first create a copy of a repository. It downloads the entire repository (all branches, history, etc.) from the remote server to your local machine. You would use `git clone` to get started with a repository, but **you wouldn't typically "clone" after that**—you'd pull updates or push your changes.

### 3. **Difference Between `git pull` and `git clone`**

- **`git clone`**: Copies the entire repository to your local machine for the first time, including all branches and commit history.

- **`git pull`**: Downloads new changes from a remote repository and merges them into your current branch. It’s for **updating** your local repository after you’ve already cloned it or initialized it.

### 4. **Is Merge No Longer Necessary After a Pull Request Is Done?**

Not exactly. Here’s how it works:

- When you create a pull request, you're asking someone to **review** and merge your changes. The merge doesn't happen automatically when the pull request is created; someone (or yourself) has to **merge** the pull request after reviewing.

- **After the PR is approved and merged**, your changes are merged into the target branch (like `main` or `master`). At this point, no further merging is needed for that specific PR.

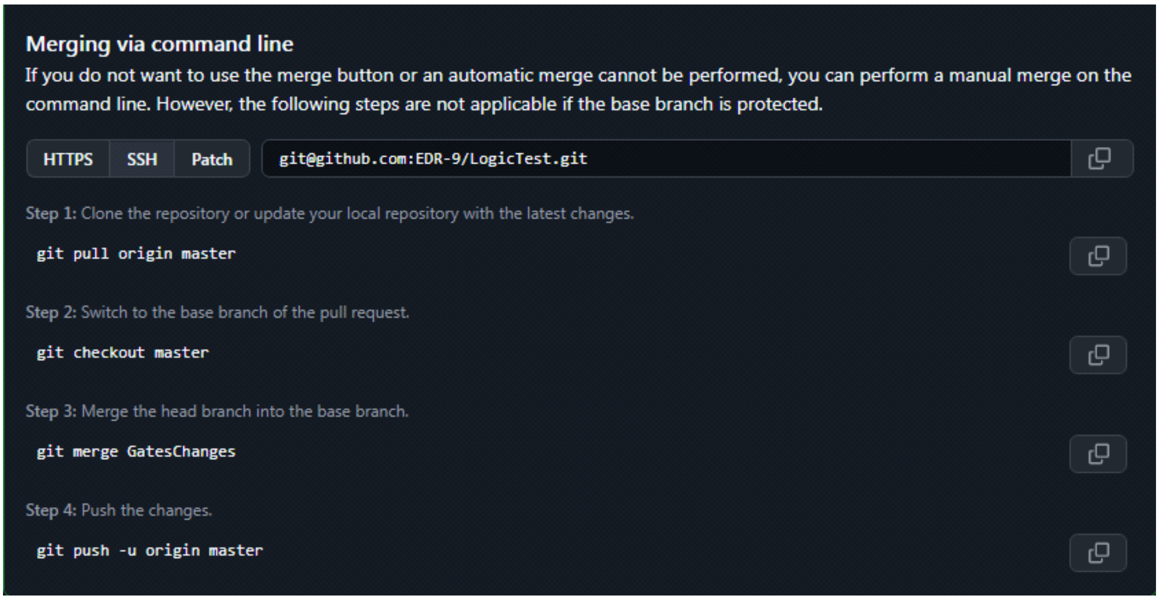
However, you still need to merge changes if you:

- Have ongoing work in other branches that needs to be merged into the main branch later.
- Need to pull in newer changes from the remote repository if other people have made updates while you were working on your PR.

### Summary:
- **Pull Request (PR)** is a proposal to merge changes into a project. It’s a reviewable process for code collaboration.
- **`git pull`** fetches new changes and merges them into your current local branch, updating your code.
- **`git clone`** copies a repository to your local machine initially, including all history.
- **Merging after a PR** is typically done manually once the PR is approved. After merging a PR, no more merging is necessary unless you're dealing with other branches or need to sync with updates.

I hope that clears things up! Let me know if you need any further clarification.

The help offered at GitHub when merging via command line after a pull request is really useful:



**Merging via command line**

If you do not want to use the merge button or an automatic merge cannot be performed, you can perform a manual merge on the command line. However, the following steps are not applicable if the base branch is protected.

```
git@github.com:EDR-9/LogicTest.git
```

Step 1: Clone the repository or update your local repository with the latest changes.

```
git pull origin master
```

Step 2: Switch to the base branch of the pull request.

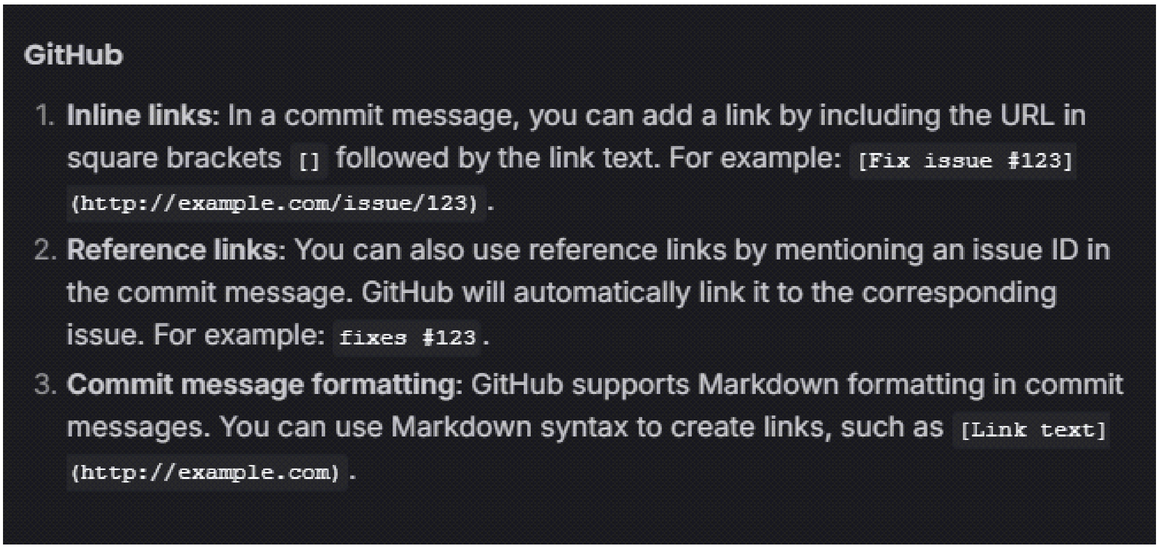
```
git checkout master
```

Step 3: Merge the head branch into the base branch.

```
git merge GatesChanges
```

Step 4: Push the changes.

```
git push -u origin master
```

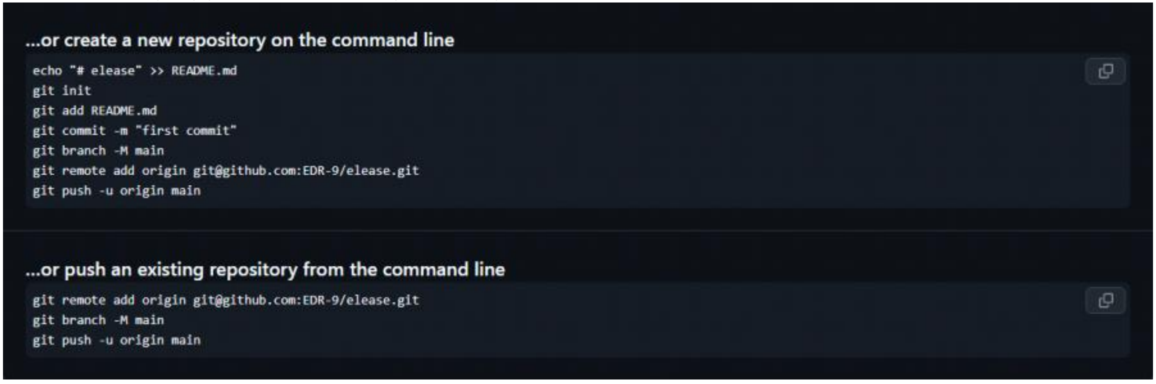


**GitHub**

1. **Inline links:** In a commit message, you can add a link by including the URL in square brackets `[]` followed by the link text. For example: `[Fix issue #123](http://example.com/issue/123)`.
2. **Reference links:** You can also use reference links by mentioning an issue ID in the commit message. GitHub will automatically link it to the corresponding issue. For example: `fixes #123`.
3. **Commit message formatting:** GitHub supports Markdown formatting in commit messages. You can use Markdown syntax to create links, such as `[Link text](http://example.com)`.

https://stackoverflow.com/questions/3832227/github-commit-message-links

Also, this help appears with every new repo creation at GitHub website



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

```
echo "# elease" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:EDR-9/elease.git
git push -u origin main
```

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

```
git remote add origin git@github.com:EDR-9/elease.git
git branch -M main
git push -u origin main
```

The meaning of the first and second part is basically the same: when you have a local repo initialized (that is, a project with Git tracking) and make the basic workflow of stage desired files, commit, set to master branch (or any other), add the remote sever for first time and push the project to remote GitHub server

Workflow example on how to proceed to solve common problems:

1. I want to upload some changes to a big project, I created a branch but it doesn't appear in the braches shown at GitHub. Are new branches taken into account once after a pull request from them is accepted?

   It sounds like the branch hasn't been pushed to the remote repository yet. Here's a quick guide to get your branch visible on GitHub:

   i. **Create your branch (which you've already done):**
   ```bash
   git checkout -b your-branch-name
   ```
   ii. **Add and commit your changes:**
   ```bash
   git add .
   git commit -m "Your commit message"
   ```
   iii. **Push the branch to the remote repository:**
   ```bash
   git push origin your-branch-name
   ```

iv. Once the branch is pushed, it should appear on GitHub. From there, you can create a pull request (PR).
v. After the pull request is merged, the branch will typically still appear on GitHub unless it’s deleted manually.

If you follow these steps, your branch should show up in the list of branches on GitHub. Let me know if you run into any issues or if there's anything more specific you're dealing with!

2. Ok, before doing that, do I need to add all of the project files to the staging area (git add .) or just the ones I changed?

   You only need to add the files that have been changed. Using `git add .` will stage all changes in your working directory, including any new, modified, or deleted files. However, you can selectively add the specific files you’ve modified with:

   ```bash
   git add path/to/your/file
   ```
   or

   ```bash
   git add filename1 filename2
   ```
   This can help keep your commits clean and focused.

   Do you have a preference for using one method over the other, or is there anything else you need clarification on?

3. Ok, then, can I skip the git add . as you wrote in past comment?

   Absolutely! You can definitely skip `git add .` if you prefer to add only the specific files you’ve changed. Just use `git add` followed by the path or filename of each individual file you want to stage. Here's a quick example:

   ```bash
   git add file1.txt file2.js path/to/file3.py
   ```
   This way, you only stage the changes that are relevant to your commit. This can help keep your commits more precise and manageable.

   Is there anything else you'd like to know about Git or any other topic?

4. How to undo a "git add ." command ?

   If you've accidentally staged files using `git add .` and want to unstage them, you can use the `git reset` command. Here's how:

   To unstage all files:
   ```bash
   git reset
   ```
   To unstage specific files, you can use:
   ```bash
   git reset path/to/file
   ```
   This will remove the files from the staging area but keep the changes in your working directory, so you won’t lose any of your modifications.

Give it a go, and let me know if you run into any issues or need more assistance! □

5. What is the difference with "git restore --staged <file> ..." to unstage?

Great question! Both `git reset` and `git restore` can be used to unstage files, but they have some nuanced differences.

### `git reset`:
- **Command**: `git reset <file>`
- **Purpose**: It unstages changes by resetting the index, but it keeps the changes in your working directory.
- **Usage**: It's often used to unstage files after an accidental `git add .` or to undo the staging of specific files.
- **Example**:
 ```bash
 git reset file1.txt
 ```

### `git restore --staged`:
- **Command**: `git restore --staged <file>`
- **Purpose**: It unstages changes by moving the changes from the index to the working directory. Essentially, it reverts the index to the last committed state for the specified files.
- **Usage**: It's a more recent command introduced to make Git commands more intuitive and aligns with the idea of "restoring" the state of files.
- **Example**:
 ```bash
 git restore --staged file1.txt
 ```

In summary, both commands achieve similar outcomes, but `git restore --staged` is newer and more aligned with the terminology of "restoring" a file to its previous state in the index. Use whichever feels more intuitive for your workflow.

6. How to undo a commit?

Undoing a commit can be done in several ways, depending on what you want to achieve. Here are a few common methods:

### **1. Undo the most recent commit (keep changes):**
This command will unstage the most recent commit and keep the changes in your working directory.
```bash
git reset --soft HEAD~1
```

### **2. Undo the most recent commit (discard changes):**
This command will unstage and discard the changes in the most recent commit.
```bash
git reset --hard HEAD~1
```

### **3. Undo a commit and keep changes (for any commit):**
If you want to undo a specific commit but keep the changes, you can use:
```bash
git revert <commit_hash>
```

This will create a new commit that undoes the changes from the specified commit.

### **4. Remove a specific commit (discard changes):**
If you want to remove a specific commit and discard its changes, you can use an interactive

rebase:
```bash
git rebase -i <commit_hash>^
```
This will open an editor where you can remove the line corresponding to the commit you want to undo.

### **5. Undo a commit that has been pushed:**
If the commit has already been pushed to a remote repository, you can force push the changes:
```bash
git reset --hard HEAD~1
git push --force
```
Be cautious with force pushing as it can overwrite history and affect other collaborators.

Which method suits your needs, or is there a specific scenario you're dealing with?

7. If I had an error on CI testing in the pull request, do I have to close pull request, commit again and pull request again or how to proceed?

No need to close the pull request and start over. Here's how you can proceed:

1. **Fix the Error**: Make the necessary changes in your code to fix the CI testing error.
2. **Commit the Changes**: Add and commit the changes in your local branch:
  ```bash
  git add .
  git commit -m "Fix CI testing error"
  ```
3. **Push to the Same Branch**: Push the changes to the same branch associated with your pull request:
  ```bash
  git push origin your-branch-name
  ```

Your pull request will automatically update with the new commit, and the CI tests will re-run to check the updated code.

This way, you don't have to close and reopen the pull request. Your CI system will handle the new commit and re-evaluate it.

Anything else you need help with?

8. Ok, I did that but now there're 2 commits to be merged when actually only one is the right, how to fix that?

No worries! You can squash the commits into a single one, making your pull request cleaner. Here's how you can do it:

### **1. Interactive Rebase:**
Use interactive rebase to squash your commits into one. Start the rebase process with:
```bash
git rebase -i HEAD~2
```
The general command is git rebase -i HEAD~n (where n is the number of recent commits you want to combine).
This command will open an editor showing the last two commits. It will look something like this:

```
pick 1234567 Commit message for the right commit
pick 89abcde Commit message for the incorrect commit
```
### **2. Change "pick" to "squash":**
Change the second "pick" to "squash" (or just "s") for the commit you want to combine:
```
pick 1234567 Commit message for the right commit
squash 89abcde Commit message for the incorrect commit
```
### **3. Save and Close the Editor:**
Save and close the editor. Another editor will open allowing you to merge the commit messages. You can keep the message of the correct commit or edit as needed.

### **4. Force Push the Squashed Commit:**
Finally, you need to force push the changes to update the pull request:
```bash
git push --force
```
This will replace the two commits with a single squashed commit in your pull request.

Let me know if you need any further assistance!

9. I have a local repo that doesn't have the latest changes showed in the GitHub repo. How can I update my local repo?

To update your local repository with the latest changes from the remote GitHub repository, you can use a few `git` commands. Here's a step-by-step guide to help you get the latest changes:

### 1. **Ensure You Are On the Correct Branch**:
First, make sure you're on the correct branch that you want to update (e.g., `main`, `master`, or any other branch).

Run the following command to check your current branch:
```bash
git branch
```

If you need to switch to a different branch, you can use:
```bash
git checkout <branch-name>
```
For example:
```bash
git checkout main
```

### 2. **Fetch the Latest Changes from the Remote Repository**:
To update your local repository with the latest changes from the GitHub repository, you'll need to **fetch** the changes from the remote (usually called `origin`):

```bash
git fetch origin
```
This will get the latest commits and changes from the remote but won’t merge them into your local branch yet.

### 3. **Merge the Changes into Your Local Branch**:
Now, you can **merge** the fetched changes into your current branch. This ensures your local repository is up to date with the latest changes from the remote.

If you want to update the branch you're currently on (e.g., `main`), run:
```bash
git merge origin/main
```
Replace `main` with whatever branch you are working on if it's different.

### 4. **Alternatively, Use Pull to Fetch and Merge in One Command**:
You can also combine the **fetch** and **merge** steps using the `git pull` command, which will automatically fetch the changes from the remote and merge them into your current branch:
```bash
git pull origin main
```
This does both the fetch and the merge in one go.

### 5. **Resolve Any Merge Conflicts (if any)**:
If there are any conflicts between your local changes and the changes in the remote repository, Git will notify you, and you'll need to resolve them manually.

After resolving conflicts, mark them as resolved:
```bash
git add <file-with-conflict>
```
Then commit the resolved changes:
```bash
git commit
```

### 6. **Push Your Changes (If You Have Local Commits to Push)**:
If you've made local changes and want to push them back to the remote repository, use:
```bash
git push origin main
```
### Summary of Commands:
- `git fetch origin` — Fetch latest changes from the remote.
- `git merge origin/<branch>` — Merge the fetched changes into your local branch.
- `git pull origin <branch>` — Fetch and merge changes in one command.
- `git add <file>` — Stage files after resolving conflicts.
- `git commit` — Commit resolved conflicts or changes.
- `git push origin <branch>` — Push your local changes to the remote.

Let me know if anything's unclear or if you run into any issues!

10. Great! I have some questions:
- Is there a need to use fetch and merge commands separately or pull command is completely equivalent?
- The latest changes just apply to the current branch I'm on?

I'm glad the explanation was helpful! Let's dive into your questions:

### 1. **Is there a need to use `fetch` and `merge` commands separately, or is `pull` completely equivalent?**

The **`git pull`** command is essentially a shortcut that combines **`git fetch`** and **`git merge`** in one step. Here's the breakdown:

- **`git fetch`**: Fetches the latest changes from the remote repository, but it doesn't modify your working directory or merge anything into your local branches. It simply updates your local copy of the remote tracking branches (like `origin/main`, `origin/feature-xyz`, etc.).

- **`git merge`**: After fetching the changes, you use `git merge` to merge the changes from the remote tracking branch (e.g., `origin/main`) into your local branch (e.g., `main`).

- **`git pull`**: A shortcut that does both: first, it fetches the latest changes (like `git fetch`), then it merges the changes into your current branch (like `git merge`).

So, **`git pull` is equivalent to using `fetch` followed by `merge`**, but it's more convenient because it does both steps in one command. If you want more control over the fetch and merge process, you might prefer using `git fetch` and `git merge` separately. But if you're okay with the automatic merge, `git pull` is perfectly fine.

### 2. **Do the latest changes only apply to the current branch you're on?**

Yes, the latest changes **only apply to the branch you're currently on**. Here's how it works:

- When you use **`git fetch`**, it updates all remote tracking branches (e.g., `origin/main`, `origin/feature-xyz`) in your local repository. But it doesn't change the state of your local branches yet.

- When you use **`git pull`** (or `git merge`), it will only pull the changes for the **branch you're on**. So if you're on `main`, it will pull changes from `origin/main` into your local `main` branch. If you're on a different branch (e.g., `feature-xyz`), `git pull` will only pull the changes for that specific branch (from `origin/feature-xyz`).

#### How to pull changes for another branch:
If you want to update a different branch (not the current branch you're on), you can explicitly pull changes for that branch by doing something like:

```bash
git checkout feature-xyz  # Switch to the branch you want to update
git pull origin feature-xyz  # Pull changes for 'feature-xyz'
```

Alternatively, you can fetch the latest changes for all branches without switching branches using:
```bash
git fetch origin
```
Then, you can merge or check the status of the specific branch you want to update.

### Recap:
- **`git pull`** combines `git fetch` and `git merge`. You don't *need* to run them separately unless you want more control over each step.
- The latest changes apply **only to the branch you're currently on**, so you'll need to switch branches if you want to pull updates for a different branch.

Feel free to ask if you need any further clarification!
You say that iit's possible to fecth latestc changes for all branches, the same can be for updating all branches like "git pull origin"?

You're absolutely right to ask about updating all branches, and I can clarify this!

### Can I use `git pull origin` to update **all** branches at once?

No, **`git pull origin`** will only update the **current branch** you're on. It will **not** update all branches in your repository. So, you’d need to run **`git pull`** for each branch you want to update.

### Fetching Changes for All Branches
When you run **`git fetch origin`**, it fetches updates for **all remote tracking branches**. This includes changes for all branches that exist on the remote (e.g., `origin/main`, `origin/feature-xyz`, `origin/dev`, etc.). However, it does **not merge** them into your local branches.

The fetch command only updates the remote tracking branches in your local repository, but **does not modify your working directory or local branches**.

So, after doing **`git fetch origin`**, you can:
- Switch to a specific branch and run `git pull` to update it, or
- Manually merge the changes from the remote tracking branches into your local branches.

### Updating All Local Branches
If you want to update **all** of your **local branches** with their respective remote counterparts (after you've fetched the latest changes), you'll need to do it manually for each branch. Here’s how you can do that:

#### 1. **Fetch All Changes from the Remote**:
```bash
git fetch origin
```

#### 2. **Update Each Branch**:
After fetching, you need to **switch to each local branch** and pull the changes:

```bash
git checkout main
git pull origin main

git checkout feature-xyz
git pull origin feature-xyz

git checkout dev
git pull origin dev
```

### Automating the Process
If you want to automate the process of pulling updates for all branches, you could use a loop in the terminal. For example, in **Bash** (Linux/macOS) or Git Bash, you could run the following:

```bash
for branch in $(git branch -r | grep -v '\->'); do
   git checkout ${branch#origin/}
   git pull origin ${branch#origin/}
done
```

This loop does the following:
1. Lists all remote branches (`git branch -r`).
2. Iterates over them, checking out each branch (`git checkout ${branch#origin/}`) and pulling the changes (`git pull origin <branch>`).

### Alternative: Using Git Reflog or Checkout
You could also use the **`git reflog`** to get a list of all the branches that have been checked out recently, and then run `git pull` on those.

### Summary:
- **`git pull origin`** only updates the current branch.
- **`git fetch origin`** updates remote tracking branches, but doesn't modify your local branches.
- To update all branches, you must switch to each one and use `git pull`.
- You can automate this process using a loop in your terminal.

Let me know if you'd like more help with automation or any other Git-related topics!

Some reading
[https://www.warp.dev/terminus/undo-git-add](https://www.warp.dev/terminus/undo-git-add)

**********************************************************************************
****************

Accessing elements of an **array of structures** in C is quite similar to accessing elements in a regular array. However, since each element is a structure, you will need to use the **dot operator (`.`)** to access the individual members of the structure.

### Declaring an Array of Structures

First, let’s define a structure and then declare an array of that structure.

#### Example:

```c
#include <stdio.h>

#define SIZE 3  // Number of elements in the array of structures

// Define a structure for a "Person"
struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    // Declare an array of structures
    struct Person people[SIZE] = {
        {"Alice", 30, 5.6},
        {"Bob", 25, 5.9},
        {"Charlie", 35, 5.8}
    };

    // Access elements of the array of structures
    printf("Person 1: %s, Age: %d, Height: %.1f\n", people[0].name, people[0].age,
people[0].height);
    printf("Person 2: %s, Age: %d, Height: %.1f\n", people[1].name, people[1].age,
people[1].height);
    printf("Person 3: %s, Age: %d, Height: %.1f\n", people[2].name, people[2].age,
people[2].height);

    return 0;
}
```

### Explanation:

1. **Structure Declaration**:
  - A structure `Person` is defined with three fields: `name`, `age`, and `height`.

2. **Array of Structures**:
  - The array `people` is an array of `SIZE` `Person` structures (in this case, 3 elements).

3. **Accessing Elements**:
  - To access individual elements of the array (`people[i]`), you use the **dot operator (`.`)** to access specific members of the structure.
  - For example:
    - `people[0].name` accesses the `name` of the first person (`Alice`).
    - `people[1].age` accesses the `age` of the second person (`Bob`).

### Output:

```
Person 1: Alice, Age: 30, Height: 5.6
Person 2: Bob, Age: 25, Height: 5.9
Person 3: Charlie, Age: 35, Height: 5.8
```

### Accessing Members Using Pointers

If you have a pointer to an element of the array (or if you have a pointer to an entire structure), you use the **arrow operator (`->`)** instead of the dot operator.

#### Example with Pointers:

```c
#include <stdio.h>

#define SIZE 3  // Number of elements in the array of structures

// Define a structure for a "Person"
struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    // Declare an array of structures
    struct Person people[SIZE] = {
        {"Alice", 30, 5.6},
        {"Bob", 25, 5.9},
        {"Charlie", 35, 5.8}
    };

    // Declare a pointer to a structure
    struct Person *ptr = &people[1];  // Point to the second person (Bob)

    // Access elements using the pointer (arrow operator ->)
    printf("Using pointer (Person 2): %s, Age: %d, Height: %.1f\n", ptr->name, ptr->
age, ptr->height);

    // You can also loop over the array of structures using pointers
    for (int i = 0; i < SIZE; i++) {
        printf("Person %d: %s, Age: %d, Height: %.1f\n", i + 1, people[i].name,
people[i].age, people[i].height);
    }

    return 0;
}
```

### Explanation:

1. **Pointer to a Structure**:
   - A pointer `ptr` is initialized to point to the second `Person` structure in the `people` array (`people[1]`).
   - You can access the members of the structure using the **arrow operator (`->`)** instead of the dot operator. So `ptr->name`, `ptr->age`, and `ptr->height` will access the name, age, and height of the person `ptr` is pointing to.

2. **Looping Over the Array of Structures**:
   - You can use a loop to iterate over the array and print each structure's fields.

### Output:

```

Using pointer (Person 2): Bob, Age: 25, Height: 5.9
Person 1: Alice, Age: 30, Height: 5.6
Person 2: Bob, Age: 25, Height: 5.9
Person 3: Charlie, Age: 35, Height: 5.8
```

---

### Key Points:
1. **Array of Structures**:
   - An array of structures is simply a collection of structures where each element of the array is a structure.
   - You can access individual structures using array indexing (`people[i]`) and access members using the dot operator (`.`).

2. **Pointer to Structure**:
   - If you have a pointer to a structure, you use the **arrow operator (`->`)** to access the structure's members.

3. **Dot (`.`) vs Arrow (`->`)**:
   - **Dot operator (`.`)**: Used when you have an actual structure variable (e.g., `people[i].name`).
   - **Arrow operator (`->`)**: Used when you have a pointer to a structure (e.g., `ptr->name`).

---

Let me know if you need any more details or examples!

Resource to learn more
https://learngitbranching.js.org/?locale=es_ES

# Resources to consult
https://rogerdudler.github.io/git-guide/index.es.html
https://www.atlassian.com/es/git/tutorials/atlassian-git-cheatsheet
https://git-scm.com/docs
https://stackoverflow.com/questions/11690709/can-a-project-have-multiple-origins