Git and GitHub

`init`: Getting started

★☆☆☆ Can create a local Git repo

Let's take your first few steps in your Git (with GitHub) journey.

0. Take a peek at the full picture(?). Optionally, if you are the sort who prefers to have some sense of the full picture before you get into the nitty-gritty details, watch the video in the panel below:

Git Overview

1. The first step is to install SourceTree, which is Git + a GUI for Git. If you prefer to use Git via the command line (i.e., without a GUI), you can install Git instead.

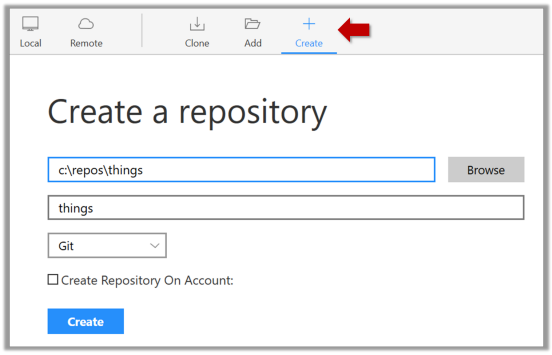
2. Next, initialize a repository. Let us assume you want to version control content in a specific directory. In that case, you need to initialize a Git repository in that directory. Here are the steps:

Create a directory for the repo (e.g., a directory named `things`).

SourceTree

Windows: Click `File` → `Clone/New...`. Click on `Create` button.
Mac: `New...` → `Create New Repository`.

Enter the location of the directory (Windows version shown below) and click `Create`.



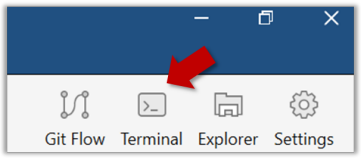
Go to the `things` folder and observe how a hidden folder `.git` has been created.

Windows: you might have to configure Windows Explorer to show hidden files.

CLI

Open a Git Bash Terminal.

If you installed SourceTree, you can click the `Terminal` button to open a GitBash terminal.



Navigate to the `things` directory.

Use the command `git init` which should initialize the repo.

```
$ git init
Initialized empty Git repository in c:/repos/things/.git/
```

You can use the command `ls -a` to view all files, which should show the `.git` directory that was created by the previous command.

```
$ ls -a
.  ..  .git
```

You can also use the `git status` command to check the status of the newly-created repo. It should respond with something like the following:

```
git status
```

↓

```
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

As you see above, this textbook explains how to use Git via SourceTree (a GUI client) as well as via the Git CLI. If you are new to Git, **we recommend you learn both the GUI method and the CLI method** -- The GUI method will help you visualize the result better while the CLI method is more universal (i.e., you will not be tied to any GUI) and more flexible/powerful.

If you are new to Git, **we caution you against using Git or GitHub features that come with the IDE** as it is better to learn Git independent of any other tool. Similarly, using clients provided by GitHub (e.g., *GitHub Desktop* GUI client) will make it harder for you to separate Git features from GitHub features.

`commit`: Saving changes to history

★☆☆☆ Can commit using Git

After initializing a repository, Git can help you with revision controlling files inside the *working directory*. However, it is not automatic. It is up to you to tell Git which of your changes (aka *revisions*) should be *committed* to its memory for later use. Saving changes into Git's memory in that way is often called *committing* and a change saved to the revision history is called a *commit*.

Working directory: the root directory revision-controlled by Git (e.g., the directory in which the repo was initialized).

Commit (noun): a change (aka a *revision*) saved in the Git revision history.
(verb): the act of creating a commit i.e., saving a change in the working directory into the Git revision history.

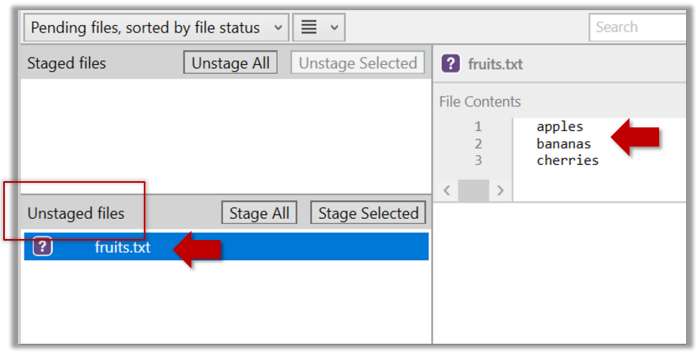
Here are the steps you can follow to learn how to work with Git commits:

1. Do some changes to the content inside the *working directory* e.g., create a file named `fruits.txt` in the `things` directory and add some dummy text to it.

2. Observe how the file is detected by Git.

SourceTree

The file is shown as 'unstaged'.



CLI

You can use the `git status` command to check the status of the working directory.

```
git status
```

↓

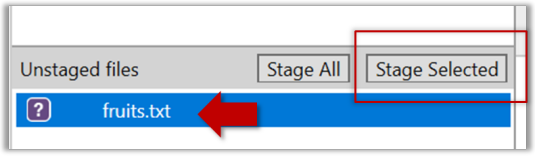
```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   a.txt
nothing added to commit but untracked files present (use "git add" to track)
```

3. Stage the changes to commit: Although Git has detected the file in the working directory, it will not do anything with the file unless you tell it to. Suppose you want to commit the current changes to the file. First, you should *stage* the file.

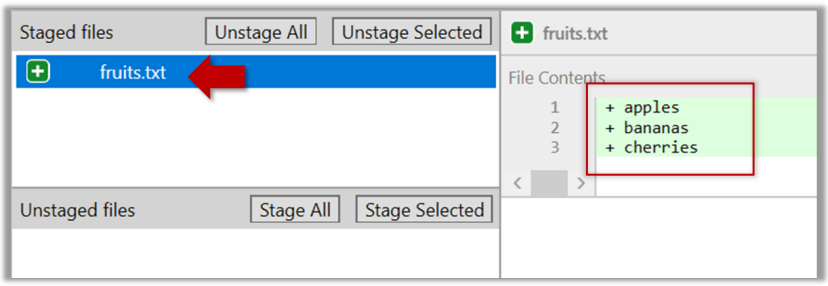
> **Stage** (verb): Instructing Git to prepare a file for committing.

SourceTree

Select the `fruits.txt` and click on the `Stage Selected` button.



`fruits.txt` should appear in the `Staged files` panel now.



CLI

You can use the `stage` or the `add` command (they are synonyms, `add` is the more popular choice) to stage files.

```
git add fruits.txt
git status
```

↓

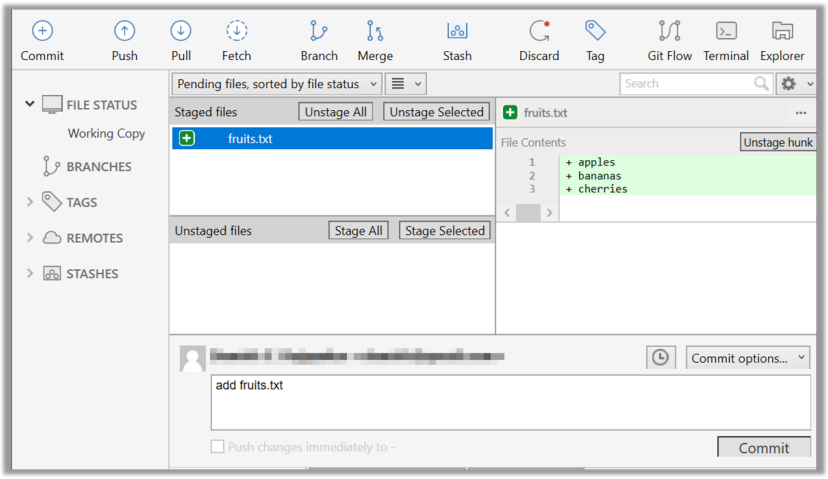
```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   fruits.txt
#
```

---

4. Commit the staged version of `fruits.txt`.

SourceTree

Click the `Commit` button, enter a commit message e.g. `add fruits.txt` into the text box, and click `Commit`.



CLI

Use the `commit` command to commit. The `-m` switch is used to specify the commit message.

```
git commit -m "Add fruits.txt"
```

You can use the `log` command to see the commit history.

```
git log
```

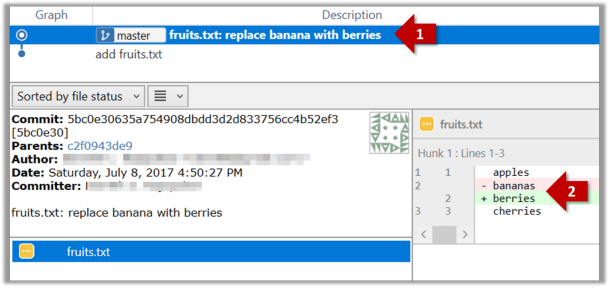
↓

```
commit 8fd30a6910efb28bb258cd01be93e481caeab846
Author: … < … @... >
Date:   Wed Jul 5 16:06:28 2017 +0800

  Add fruits.txt
```

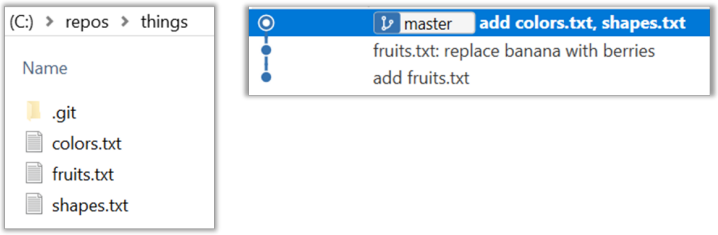
Note the existence of something called the `master` branch. Git allows you to have multiple branches (i.e. it is a way to evolve the content in parallel) and Git auto-creates a branch named `master` on which the commits go on by default.

5. Do a few more commits.

1. Make some changes to `fruits.txt` (e.g. add some text and delete some text). Stage the changes, and commit the changes using the same steps you followed before. You should end up with something like this.



2. Next, add two more files `colors.txt` and `shapes.txt` to the same working directory. Add a third commit to record the current state of the working directory.



6. See the revision graph: Note how commits form a path-like structure aka the *revision tree/graph*. In the revision graph, each commit is shown as linked to its 'parent' commit (i.e., the commit before it).

SourceTree

To see the revision graph, click on the `History` item on the menu on the right edge of SourceTree.

CLI

The `gitk` command opens a rudimentary graphical view of the revision graph.

Resources

- **Try Git** is an online simulation/tutorial of Git basics. You can try its first few steps to solidify what you have learned in this LO.

Omitting files from revision control

★★☆☆ Can set Git to ignore files

Often, there are files inside the Git working folder that you don't want to revision-control e.g., temporary log files. Follow the steps below to learn how to configure Git to ignore such files.

1. Add a file into your repo's working folder that you supposedly don't want to revision-control e.g., a file named `temp.txt`. Observe how Git has detected the new file.

2. Tell Git to ignore that file:

SourceTree

The file should be currently listed under `Unstaged files`. Right-click it and choose `Ignore...`. Choose `Ignore exact filename(s)` and click `OK`.

Observe that a file named `.gitignore` has been created in the working directory root and has the following line in it.

```
1 temp.txt
```

CLI

Create a file named `.gitignore` in the working directory root and add the following line in it.

```
1 temp.txt
```

The `.gitignore` file

The `.gitignore` file tells Git which files to ignore when tracking revision history. That file itself can be either revision controlled or ignored.

- To version control it (the more common choice – which allows you to track how the `.gitignore` file changes over time), simply commit it as you would commit any other file.
- To ignore it, follow the same steps you followed above when you set Git to ignore the `temp.txt` file.
- It supports file patterns e.g., adding `temp/*.tmp` to the `.gitignore` file prevents Git from tracking any `.tmp` files in the `temp` directory.

More information about the `.gitignore` file: git-scm.com/docs/gitignore

Files recommended to be omitted from version control

- **Binary files** *generated* when building your project e.g., `*.class`, `*.jar`, `*.exe` (reasons: 1. no need to version control these files as they can be generated again from the source code 2. Revision control systems are optimized for tracking text-based files, not binary files.
- **Temporary files** e.g., log files generated while testing the product
- **Local files** i.e., files specific to your own computer e.g., local settings of your IDE
- **Sensitive content** i.e., files containing sensitive/personal information e.g., credential files, personal identification data (especially, if there is a possibility of those files getting leaked via the revision control system).

`tag`: Naming commits

★★★☆ Can tag commits using Git

Each Git commit is uniquely identified by a hash e.g., `d670460b4b4aece5915caf5c68d12f560a9fe3e4`. As you can imagine, using such an identifier is not very convenient for our day-to-day use. As a solution, Git allows adding a more human-readable *tag* to a commit e.g., `v1.0-beta`.

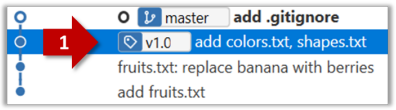
Here's how you can tag a commit in a local repo (e.g. in the `samplerepo-things` repo):

SourceTree

Right-click on the commit (in the graphical revision graph) you want to tag and choose `Tag...`.

Specify the tag name e.g. `v1.0` and click `Add Tag`.

The added tag will appear in the revision graph view.



CLI

To add a tag to the current commit as `v1.0`,

```
git tag -a v1.0
```

To view tags

```
git tag
```

To learn how to add a tag to a past commit, go to the 'Git Basics – Tagging' page of the git-scm book and refer the 'Tagging Later' section.

> ! Remember to push tags to the repo. A normal push does not include tags.
>
> ```
> # push a specific tag
> git push origin v1.0b
>
> # push all tags
> git push origin --tags
> ```

After adding a tag to a commit, you can use the tag to refer to that commit, as an alternative to using the hash.

> ! **Tags are different from commit messages**, in purpose and in form. A commit message is a description of the commit that is *part of* the commit itself. A tags is a short name for a commit, which exists as a separate entity that *points to* a commit.

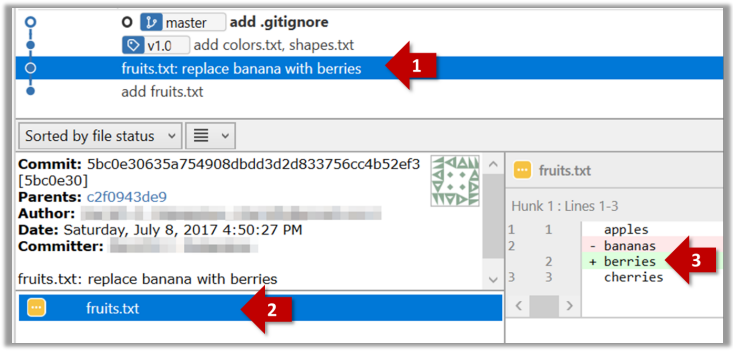
`diff`: Comparing revisions

★★☆☆ Can compare git revisions

Git can show you what changed in each commit.

SourceTree

To see which files changed in a commit, click on the commit. To see what changed in a specific file in that commit, click on the file name.



CLI

```
git show < part-of-commit-hash >
```

Example:

```
git show 251b4cf
```

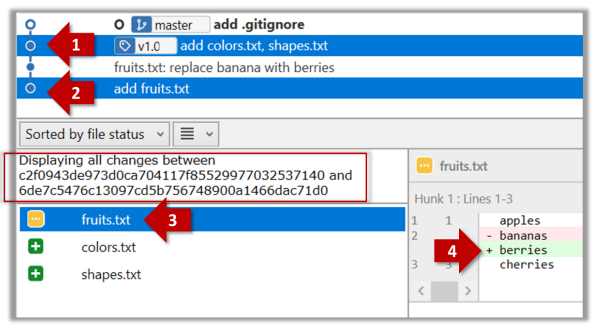
↓

```
commit 5bc0e30635a754908dbdd3d2d833756cc4b52ef3
Author: … < … >
Date:   Sat Jul 8 16:50:27 2017 +0800

    fruits.txt: replace banana with berries

diff --git a/fruits.txt b/fruits.txt
index 15b57f7..17f4528 100644
--- a/fruits.txt
+++ b/fruits.txt
@@ -1,3 +1,3 @@
 apples
-bananas
+berries
 cherries
```

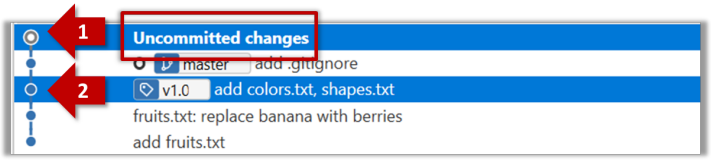
Git can also show you the difference between two points in the history of the repo.

SourceTree

Select the two points you want to compare using `Ctrl` + `Click`. The differences between the two selected versions will show up in the bottom half of SourceTree, as shown in the screenshot below.



The same method can be used to compare the current state of the working directory (which might have uncommitted changes) to a point in the history.



CLI

The `diff` command can be used to view the differences between two points of the history.

- `git diff`: shows the changes (uncommitted) since the last commit.
- `git diff 0023cdd..fcd6199`: shows the changes between the points indicated by commit hashes.
  Note that when using a commit hash in a Git command, you can use only the first few characters (e.g., first 7-10 chars) as that's usually enough for Git to locate the commit.
- `git diff v1.0..HEAD`: shows changes that happened from the commit tagged as `v1.0` to the most recent commit.

Resources

- Git-Tower Tutorial: Inspecting Changes with Diffs
- How to view the next page of a `git` command result

`checkout`: Retrieving a specific revision

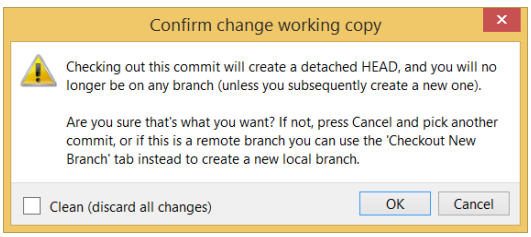
★★☆☆ Can load a specific version of a Git repo

Git can load a specific version of the history to the working directory. Note that if you have uncommitted changes in the working directory, you need to stash them first to prevent them from being overwritten.

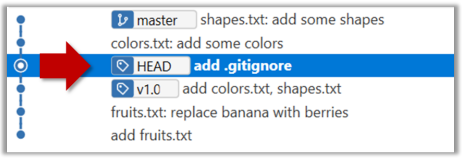
SourceTree

Double-click the commit you want to load to the working directory, or right-click on that commit and choose `Checkout...`.

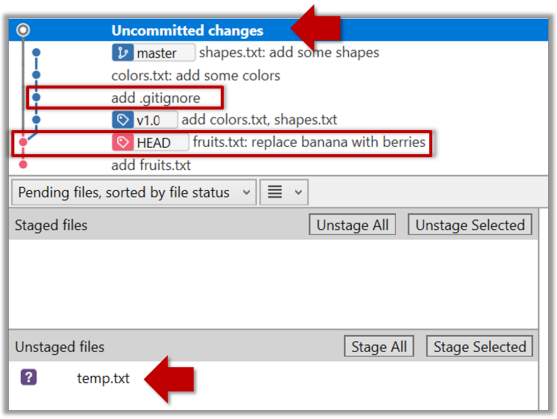
Click `OK` to the warning about 'detached HEAD' (similar to below).



The specified version is now loaded to the working folder, as indicated by the `HEAD` label. `HEAD` is a reference to the currently checked out commit.



If you checkout a commit that comes before the commit in which you added the `.gitignore` file, Git will now show ignored files as 'unstaged modifications' because at that stage Git hasn't been told to ignore those files.



To go back to the latest commit, double-click it.

CLI

Use the `checkout <commit-identifier>` command to change the working directory to the state it was in at a specific past commit.

- `git checkout v1.0`: loads the state as at commit tagged `v1.0`
- `git checkout 0023cdd`: loads the state as at commit with the hash `0023cdd`
- `git checkout HEAD~2`: loads the state that is 2 commits behind the most recent commit

For now, you can ignore the warning about 'detached HEAD'.

`stash`: Shelving changes temporarily

★★★★ Can use Git to stash files

You can use Git's *stash* feature to temporarily shelve (or stash) changes you've made to your working copy so that you can work on something else, and then come back and re-apply the stashed changes later on. -- adapted from Atlassian

Source Tree

Follow this article from SourceTree creators. Note that the GUI shown in the article is slightly outdated but you should be able to map it to the current GUI.

CLI

Follow this article from Atlassian.

`clone`: Copying a repo

★☆☆☆ Can clone a remote repo

Given below is an example scenario you can try yourself to learn Git cloning.

Suppose you want to clone the sample repo samplerepo-things to your computer.

> ! Note that the URL of the GitHub project is different from the URL you need to clone a repo in that GitHub project. e.g.
>
> GitHub project URL: `https://github.com/se-edu/samplerepo-things`
> Git repo URL: `https://github.com/se-edu/samplerepo-things.git` (note the `.git` at the end)

SourceTree

`File` → `Clone / New…` and provide the URL of the repo and the destination directory.

CLI

You can use the `clone` command to clone a repo.

Follow the instructions given here.

`pull`, `fetch`: Downloading data from other repos

★☆☆☆ Can pull changes from a repo

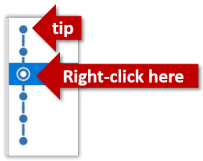
Here's a scenario you can try in order to learn how to pull commits from another repo to yours.

1. Clone a repo (e.g., the repo used in [Git & GitHub → Clone]) to be used for this activity.

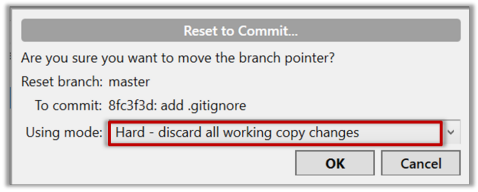
2. Delete the last few commits to simulate cloning the repo a few commits ago.

SourceTree

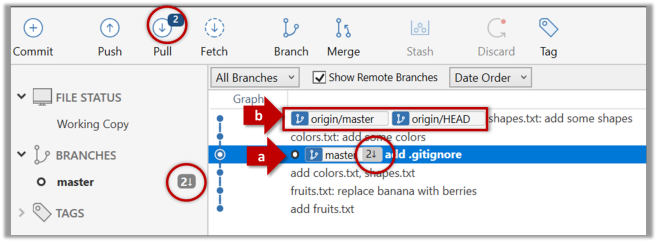
Right-click the target commit (i.e. the commit that is 2 commits behind the tip) and choose `Reset current branch to this commit`.



Choose the `Hard - ...` option and click `OK`.



This is what you will see.



Note the following (cross refer the screenshot above):

Arrow marked as `a`: The local repo is now at this commit, marked by the `master` label.
Arrow marked as `b`: The `origin/master` label shows what is the latest commit in the `master` branch in the remote repo.

CLI

Use the `reset` command to delete commits at the tip of the revision history.

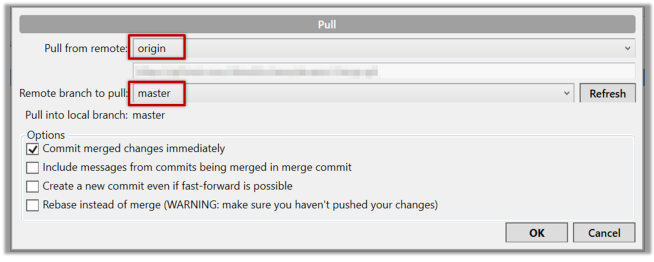
```
git reset --hard HEAD~2
```

Now, your local repo state is exactly how it would be if you had cloned the repo 2 commits ago, as if somebody has added two more commits to the remote repo since you cloned it.

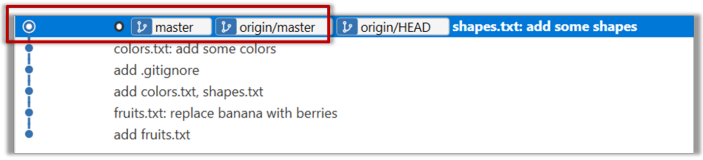
3. Pull from the other repo: To get those missing commits to your local repo (i.e. to sync your local repo with upstream repo) you can do a pull.

SourceTree

Click the `Pull` button in the main menu, choose `origin` and `master` in the next dialog, and click `OK`.



Now you should see something like this where `master` and `origin/master` are both pointing the same commit.



CLI

`git pull origin`

You can also do a `fetch` instead of a `pull` in which case the new commits will be downloaded to your repo but the working directory will remain at the current commit. To move the current state to the latest commit that was downloaded, you need to do a `merge`. A `pull` is a shortcut that does both those steps in one go.

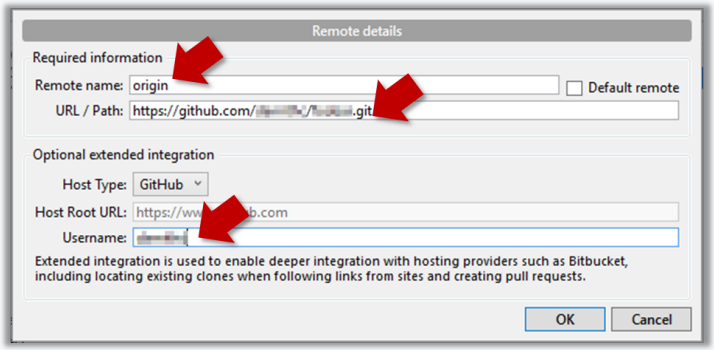
Working with multiple remotes

When you clone a repo, Git automatically adds a *remote* repo named `origin` to your repo configuration. As you know, you can pull commits from that repo. As you know, a Git repo can work with remote repos other than the one it was cloned from.

To communicate with another remote repo, you can first add it as a *remote* of your repo. Here is an example scenario you can follow to learn how to pull from another repo:

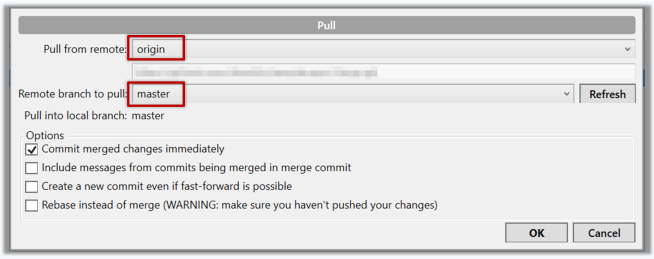
SourceTree

1. Open the local repo in SourceTree. Suggested: Use your local clone of the `samplerepo-things` repo.
2. Choose `Repository` → `Repository Settings` menu option.
3. Add a new *remote* to the repo with the following values.



   - `Remote name`: the name you want to assign to the remote repo e.g., `upstream1`
   - `URL/path`: the URL of your repo (ending in `.git`) that. Suggested: `https://github.com/se-edu/samplerepo-things-2.git` (`samplerepo-things-2` is another repo that has a shared history with `samplerepo-things`)
   - `Username`: your GitHub username

4. Now, you can pull from the added repo as you did before but choose the remote name of the repo you want to pull from (instead of `origin`):



If the `Remote branch to pull` dropdown is empty, click the `Refresh` button on its right.

5. If the pull from the `samplerepo-things-2` was successful, you should have received one more commit into your local repo.

CLI

1. Navigate to the folder containing the local repo.

2. Set the new remote repo as a *remote* of the local repo.
   command: `git remote add {remote_name} {remote_repo_url}`
   e.g., `git remote add upstream1 https://github.com/johndoe/foobar.git`

3. Now you can pull from the new remote.
   e.g., `git pull upstream1 master`

Fork: Creating a remote copy

★☆☆☆ Can fork a repo

Given below is a scenario you can try in order to learn how to fork a repo:.

0. Create a GitHub account if you don't have one yet.

1. Go to the GitHub repo you want to fork e.g., samplerepo-things

2. Click on the Fork **button** on the top-right corner. In the next step,

- choose to fork to your own account or to another GitHub organization that you are an admin of.
- Un-tick the `[ ] Copy the master branch only` option.

GitHub does not allow you to fork the same repo more than once to the same destination. If you want to re-fork, you need to delete the previous fork.

`push`: Uploading data to other repos

★☆☆☆ Can push to a remote repo

Given below is a scenario you can try in order to learn how to push commits to a remote repo hosted on GitHub:

1. Fork an existing GitHub repo (e.g., samplerepo-things) to your GitHub account.

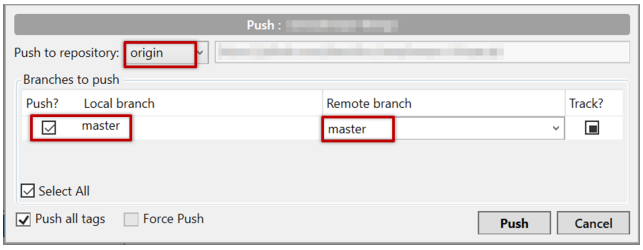
2. Clone the fork (not the original) to your computer.

3. Commit some changes in your local repo.

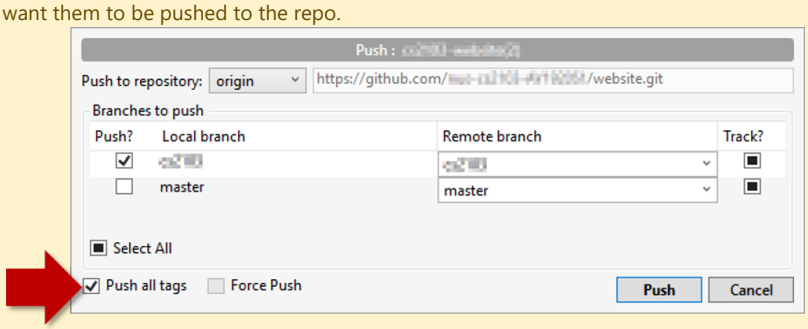
4. Push the new commits to your fork on GitHub

SourceTree

Click the `Push` button on the main menu, ensure the settings are as follows in the next dialog, and click the `Push` button on the dialog.



! Tags are not included in a normal push. Remember to tick `Push all tags` when pushing to the remote repo if you want them to be pushed to the repo.



CLI

Use the command `git push origin master`. Enter your Github username and password when prompted.

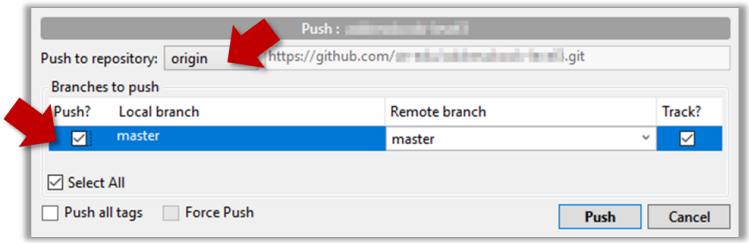
! Tags are not included in a normal push. To push a tag, use this command: `git push origin <tag_name>` e.g. `git push origin v1.0`

You can push to repos other than the one you cloned from, as long as the target repo and your repo have a shared history.

1. Add the GitHub repo URL as a remote, if you haven't done so already.
2. Push to the target repo.

SourceTree

Push your repo to the new remote the usual way, but select the name of target remote instead of `origin` and remember to select the `Track` checkbox.



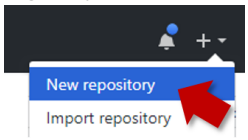
CLI

Push to the new remote the usual way e.g., `git push upstream1 master` (assuming you gave the name `upstream1` to the remote).

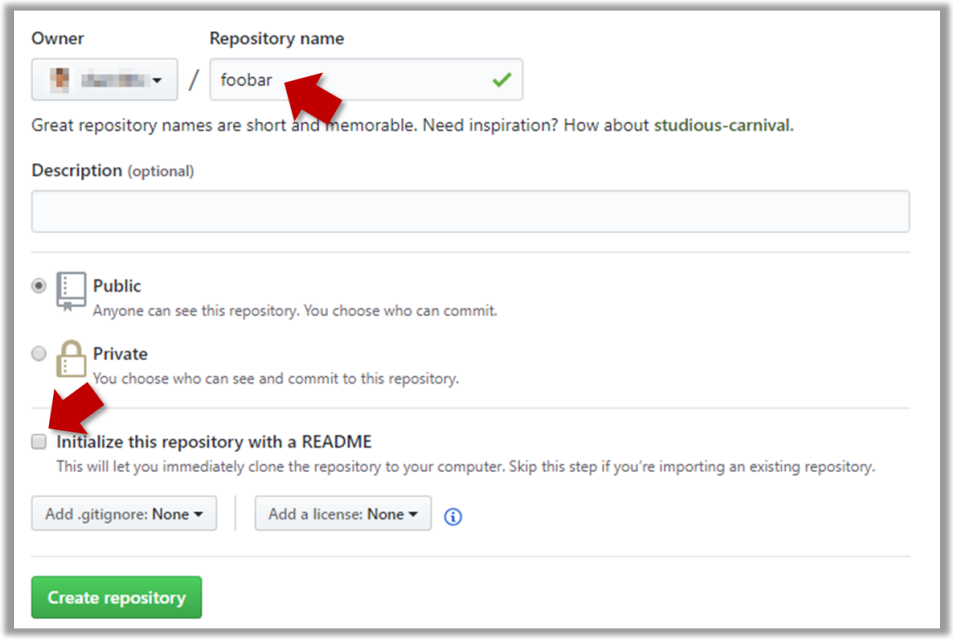
You can even push an entire local repository to GitHub, to form an entirely new remote repository. For example, you created a local repo and worked with it for a while but now you want to upload it onto GitHub (as a backup or to share it with others). The steps are given below.

1. Create an empty remote repo on GitHub.

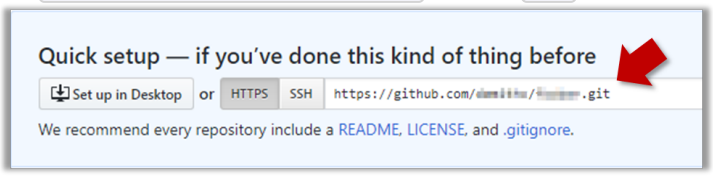
1. Login to your GitHub account and choose to create a new Repo.



2. In the next screen, provide a name for your repo but keep the `Initialize this repo ...` tick box unchecked.



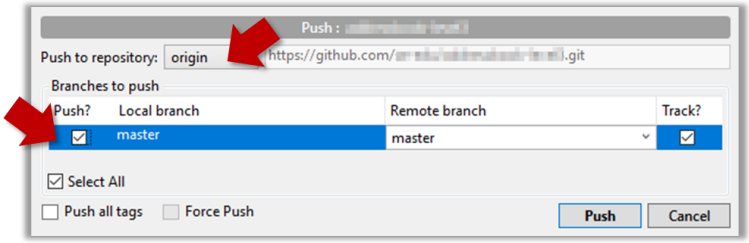
3. Note the URL of the repo. It will be of the form `https://github.com/{your_user_name}/{repo_name}.git`.
e.g., `https://github.com/johndoe/foobar.git` (note the `.git` at the end)



2. Add the GitHub repo URL as a remote of the local repo. You can give it the name `origin` (or any other name).

3. Push the repo to the remote.

SourceTree



CLI

Push each branch to the new remote the usual way but use the `-u` flag to inform Git that you wish to track the branch.
e.g., `git push -u origin master`

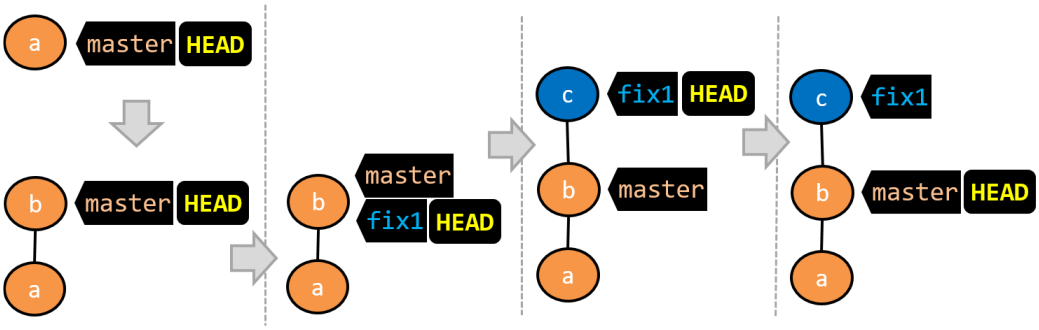
`branch`: Doing multiple parallel changes

★★☆☆ Can use Git branching

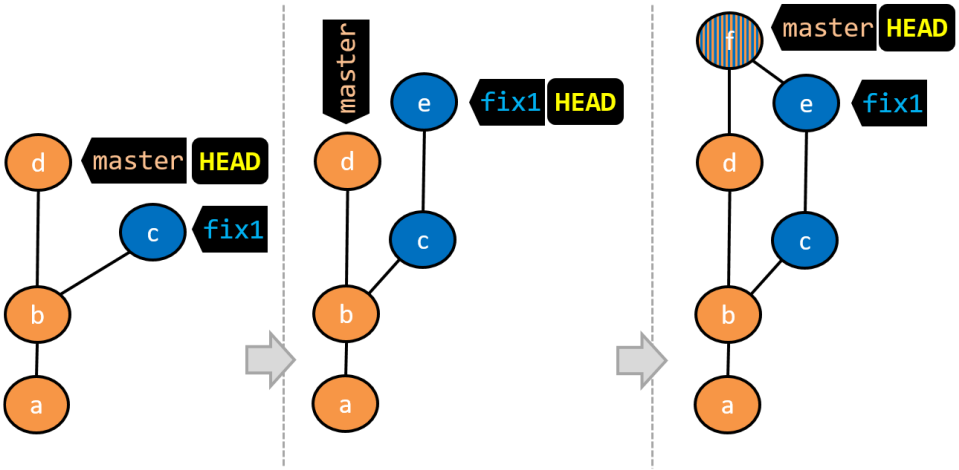
Git supports branching, which allows you to do multiple parallel changes to the content of a repository.

A Git branch is simply a *named label* pointing to a commit. The `HEAD` label indicates which branch you are on. Git creates a branch named `master` by default. When you add a commit, it goes into the branch you are currently on, and the branch label (together with the `HEAD` label) moves to the new commit.

Given below is an illustration of how branch labels move as branches evolve.



1. There is only one branch (i.e., `master`) and there is only one commit on it.
2. A new commit has been added. The `master` and the `HEAD` labels have moved to the new commit.
3. A new branch `fix1` has been added. The repo has switched to the new branch too (hence, the `HEAD` label is attached to the `fix1` branch).
4. A new commit (`c`) has been added. The current branch label `fix1` moves to the new commit, together with the `HEAD` label.
5. The repo has switched back to the `master` branch.

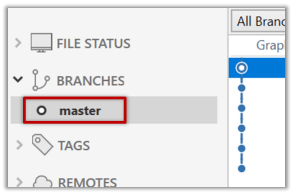


6. A new commit (`d`) has been added. The `master` label has moved to that commit.
7. The repo has switched back to the `fix1` branch and added a new commit (`e`) to it.
8. The repo has switched to the `master` branch and the `fix1` branch has been merged into the `master` branch, creating a *merge commit* `f`. The repo is currently on the `master` branch.

Follow the steps below to learn how to work with branches. You can use any repo you have on your computer (e.g. a clone of the samplerepo-things) for this.

0. Observe that you are normally in the branch called `master`.

SourceTree



CLI

```
git status
```

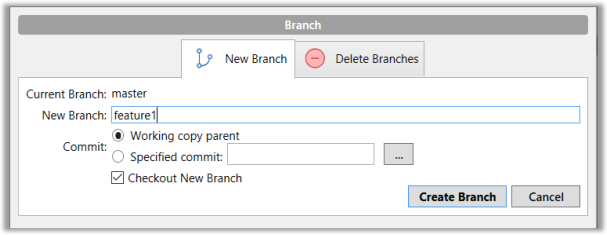
↓

```
on branch master
```

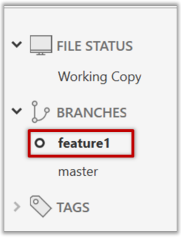
1. Start a branch named `feature1` and switch to the new branch.

SourceTree

Click on the `Branch` button on the main menu. In the next dialog, enter the branch name and click `Create Branch`.



Note how the `feature1` is indicated as the current branch.



CLI

You can use the `branch` command to create a new branch and the `checkout` command to switch to a specific branch.

```
git branch feature1
git checkout feature1
```

One-step shortcut to create a branch and switch to it at the same time:

```
git checkout -b feature1
```

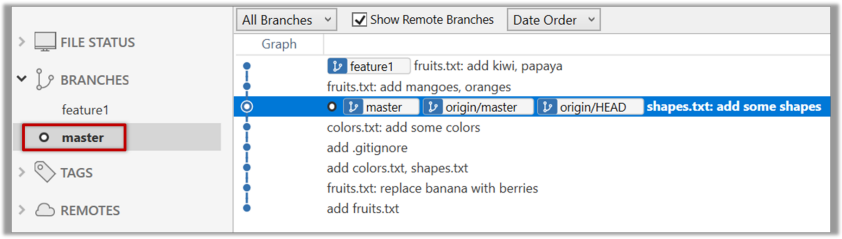
---

2. Create some commits in the new branch. Just commit as per normal. Commits you add while on a certain branch will become part of that branch.
Note how the `master` label and the `HEAD` label moves to the new commit (The `HEAD` label of the local repo is represented as ● in SourceTree).

3. Switch to the `master` branch. Note how the changes you did in the `feature1` branch are no longer in the working directory.

SourceTree

Double-click the `master` branch.

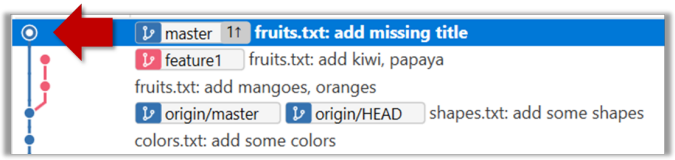


CLI

```
git checkout master
```

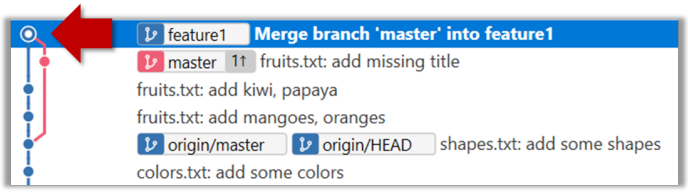
---

4. Add a commit to the master branch. Let's imagine it's a bug fix.
To keep things simple for the time being, this commit should not involve the same content that you changed in the `feature1` branch. To be on the safe side, this commit can change an entirely different file.



5. Switch back to the `feature1` branch (similar to step 3).

6. Merge the `master` branch to the `feature1` branch, giving an end-result like the following. Also note how Git has created a *merge commit*.



SourceTree

Right-click on the `master` branch and choose `merge master into the current branch`. Click `OK` in the next dialog.

CLI

```
1 git merge master
```

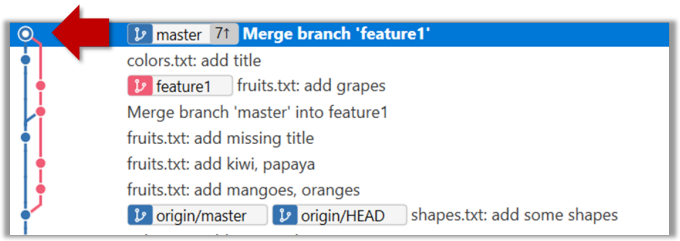
The objective of that merge was to *sync* the `feature1` branch with the `master` branch. Observe how the changes you did in the `master` branch (i.e. the imaginary bug fix) is now available even when you are in the `feature1` branch.

> Instead of merging `master` to `feature1`, an alternative is to *rebase* the `feature1` branch. However, rebasing is an advanced feature that requires modifying past commits. If you modify past commits that have been pushed to a remote repository, you'll have to *force-push* the modified commit to the remote repo in order to update the commits in it.

7. Add another commit to the `feature1` branch.

8. Switch to the `master` branch and add one more commit.

9. Merge `feature1` to the master branch, giving and end-result like this:



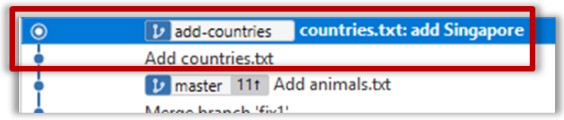
SourceTree

Right-click on the `feature1` branch and choose `Merge...`.

CLI

```
1 git merge feature1
```

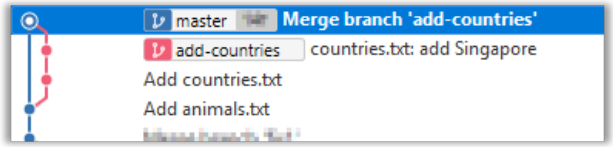
10. Create a new branch called `add-countries`, switch to it, and add some commits to it (similar to steps 1-2 above). You should have something like this now:



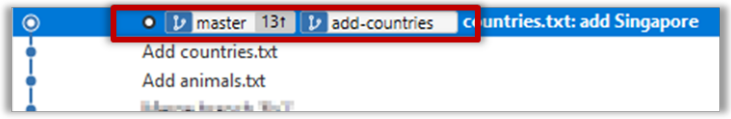
✕ **Avoid this common rookie mistake!**

Always remember to switch back to the `master` branch before creating a new branch. If not, your new branch will be created on top of the current branch.

11. Go back to the `master` branch and merge the `add-countries` branch onto the `master` branch (similar to steps 8-9 above). While you might expect to see something like the following,



... you are likely to see something like this instead:

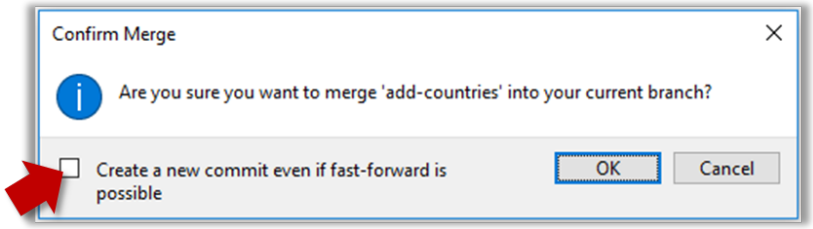


That is because **Git does a *fast forward* merge if possible**. Seeing that the `master` branch has not changed since you started the `add-countries` branch, Git has decided it is simpler to just put the commits of the `add-countries` branch in front of the `master` branch, without going into the trouble of creating an extra merge commit.

It is possible to force Git to create a merge commit even if fast forwarding is possible.

SourceTree

Tick the box shown below when you merge a branch:



CLI

Use the `--no-ff` switch (short for *no fast forward*):

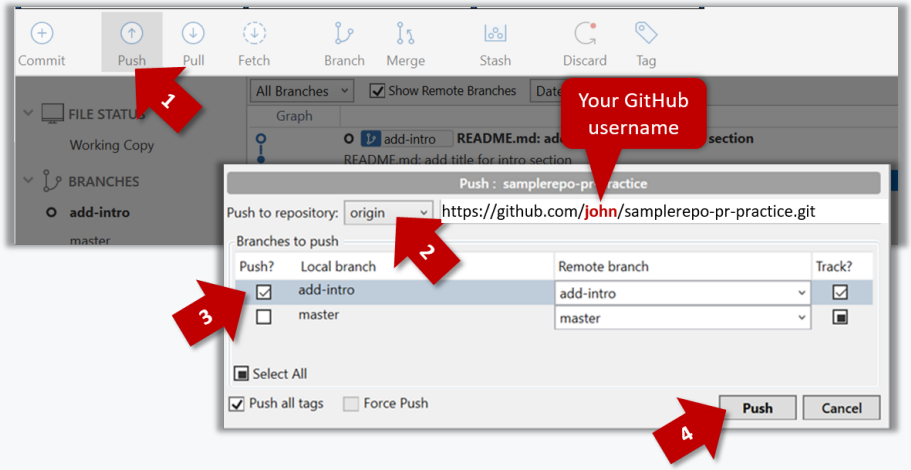
```
git merge --no-ff add-countries
```

Pushing a branch to a remote repo

Here's how to push a branch to a remote repo:

SourceTree

Here's how to push a branch named `add-intro` to your own fork of a repo named `samplerepo-pr-practice`:



CLI

Normally: `git push {remote repository} {branch}`. Examples:

- `git push origin master` pushes the `master` branch to the repo named `origin` (i.e., the repo you cloned from)
- `git push upstream-repo add-intro` pushes the `add-intro` branch to the repo named `upstream-repo`

If pushing a branch you created locally to the remote for the *first time*, add the `-u` flag to get the local branch to track the new upstream branch:
e.g., `git push -u origin add-intro`

See git-scm.com/docs/git-push for details of the `push` command.

Dealing with merge conflicts

★★☆☆ Can use Git to resolve merge conflicts

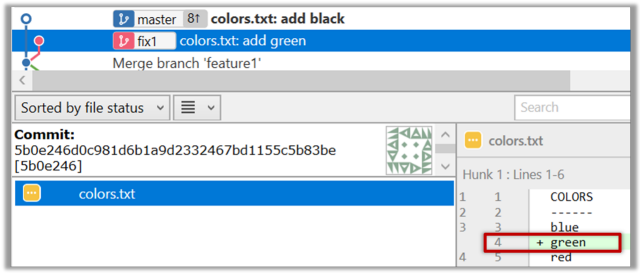
Merge conflicts happen when you try to combine two incompatible versions (e.g., merging a branch to another but each branch changed the same part of the code in a different way).

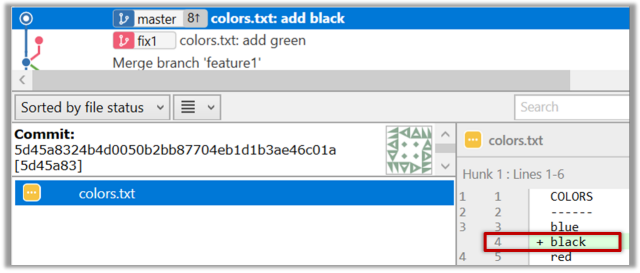
Here are the steps to simulate a merge conflict and use it to learn how to resolve merge conflicts.

0. Create an empty repo or clone an existing repo, to be used for this activity.

1. Start a branch named `fix1` in the repo. Create a commit that adds a line with some text to one of the files.

2. Switch back to `master` branch. Create a commit with a conflicting change i.e. it adds a line with some different text in the exact location the previous line was added.





3. Try to merge the `fix1` branch onto the `master` branch. Git will pause mid-way during the merge and report a merge conflict. If you open the conflicted file, you will see something like this:

```
COLORS
------
blue
<<<<<< HEAD
black
=======
green
>>>>>> fix1
red
white
```

4. Observe how the conflicted part is marked between a line starting with `<<<<<<` and a line starting with `>>>>>>`, separated by another line starting with `=======`.

Highlighted below is the conflicting part that is coming from the `master` branch:

```
blue
<<<<<< HEAD
black
=======
green
>>>>>> fix1
red
```

This is the conflicting part that is coming from the `fix1` branch:

```
blue
<<<<<< HEAD
black
=======
green
>>>>>> fix1
red
```

5. Resolve the conflict by editing the file. Let us assume you want to keep both lines in the merged version. You can modify the file to be like this:

```
COLORS
------
blue
black
green
red
white
```

6. Stage the changes, and commit.

Creating PRs

★☆☆☆ Can create PRs on GitHub

Suppose you want to propose some changes to a GitHub repo (e.g., samplerepo-pr-practice) as a pull request (PR). Here is a scenario you can try in order to learn how to create PRs:

1. Fork the repo onto your GitHub account.

2. Clone it onto your computer.

3. Commit your changes e.g., add a new file with some contents and commit it.

- **Option A - Commit changes to the `master` branch**
- **Option B - Commit to a new branch** e.g., create a branch named `add-intro` (remember to switch to the `master` branch before creating a new branch) and add your commit to it.

4. Push the branch you updated (i.e., `master` branch or the new branch) to your fork, as explained here.

5. Initiate the PR creation:

1. Go to your fork.
2. Click on the `Pull requests` tab followed by the New pull request **button**. This will bring you to the 'Comparing changes' page.
3. Set the appropriate target repo and the branch that should receive your PR, using the `base repository` and `base` dropdowns. e.g.,
   base repository: **se-edu/samplerepo-pr-practice** base: **master**

   > ℹ Normally, the default value shown in the dropdown is what you want but in case your fork has multiple upstream repos, the default may not be what you want.

4. Indicate which repo:branch contains your proposed code, using the `head repository` and `compare` dropdowns. e.g.,
   head repository: **myrepo/samplerepo-pr-practice** compare: **master**

6. Verify the proposed code: Verify that the diff view in the page shows the exact change you intend to propose. If it doesn't, update the branch as necessary.

7. Submit the PR:

1. Click the Create pull request button.
2. Fill in the PR name and description e.g.,
   Name: `Add an introduction to the README.md`
   Description:

   ```
   Add some paragraph to the README.md to explain ...
   Also add a heading ...
   ```

3. If you want to indicate that the PR you are about to create is 'still work in progress, not yet ready', click on the dropdown arrow in the Create pull request button and choose `Create draft pull request` option.
4. Click the Create pull request button to create the PR.
5. Go to the receiving repo to verify that your PR appears there in the `Pull requests` tab.

The next step of the PR life cycle is the PR *review*. The members of the repo that received your PR can now review your proposed changes.

- If they like the changes, they can *merge* the changes to their repo, which also closes the PR automatically.
- If they don't like it at all, they can simply close the PR too i.e., they reject your proposed change.
- In most cases, they will add comments to the PR to suggest further changes. When that happens, GitHub will notify you.

You can update the PR along the way too. Suppose PR reviewers suggested a certain improvement to your proposed code. To update your PR as per the suggestion, you can simply modify the code in your local repo, commit the updated code to the same `master` branch, and push to your fork as you did earlier. The PR will auto-update accordingly.

Sending PRs using the `master` branch is less common than sending PRs using separate branches. For example, suppose you wanted to propose two bug fixes that are not related to each other. In that case, it is more appropriate to send two separate PRs so that each fix can be reviewed, refined, and merged independently. But if you send PRs using the `master` branch only, both fixes (and any other change you do in the `master` branch) will appear in the PRs you create from it.

To create another PR while the current PR is still under review, create a new branch (remember to switch back to the `master` branch first), add your new proposed change in that branch, and create a new PR following the steps given above.

It is possible to create PRs within the same repo e.g., you can create a PR from branch `feature-x` to the `master` branch, within the same repo. Doing so will allow the code to be reviewed by other developers (using PR review mechanism) before it is merged.

Resources

- GitHub's own documentation on creating a PR

Reviewing PRs

★★☆☆ Can review PRs on GitHub

The *PR review* stage is a dialog between the PR author and members of the repo that received the PR, in order to refine and eventually merge the PR.

Given below are some steps you can follow when reviewing a PR.

1. Locate the PR:

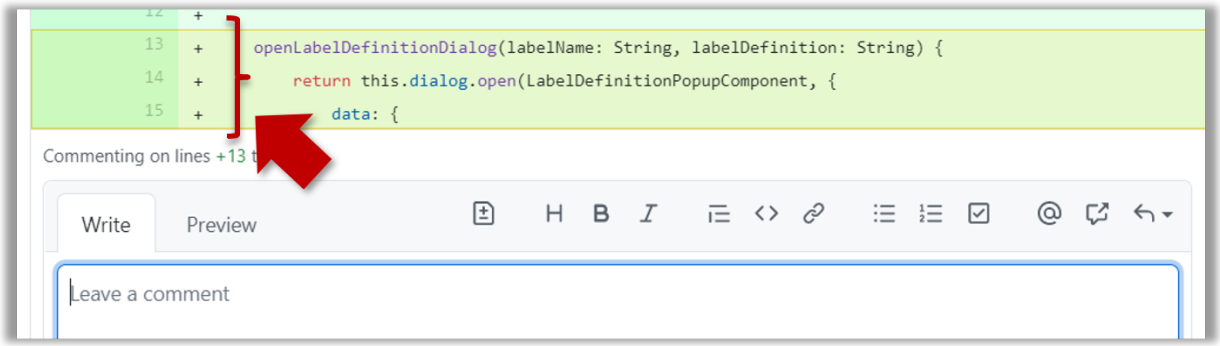
1. Go to the GitHub page of the repo.
2. Click on the `Pull requests` tab.
3. Click on the PR you want to review.

2. Read the PR description. It might contain information relevant to reviewing the PR.

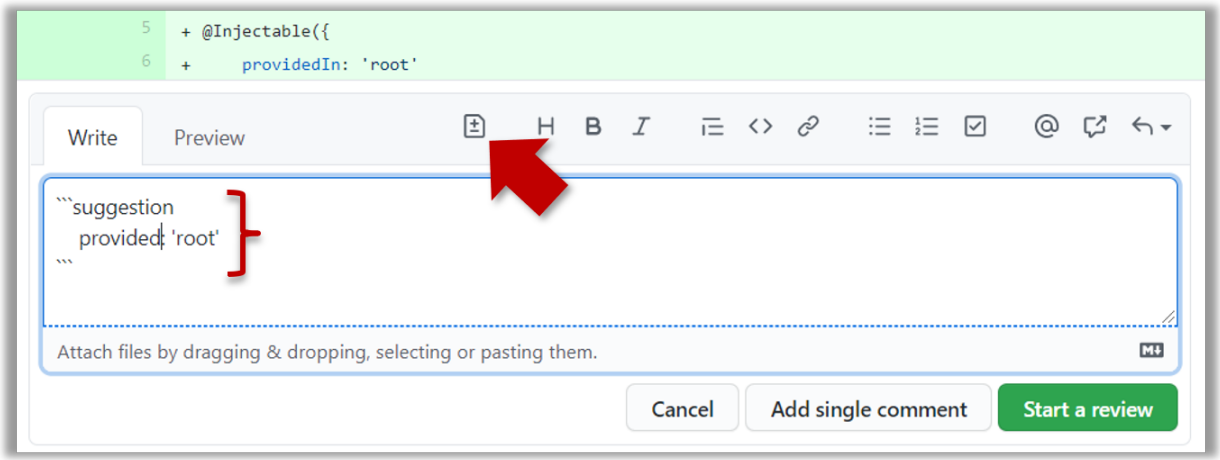
3. Click on the `Files changed` tab to see the *diff* view.

4. Add review comments:

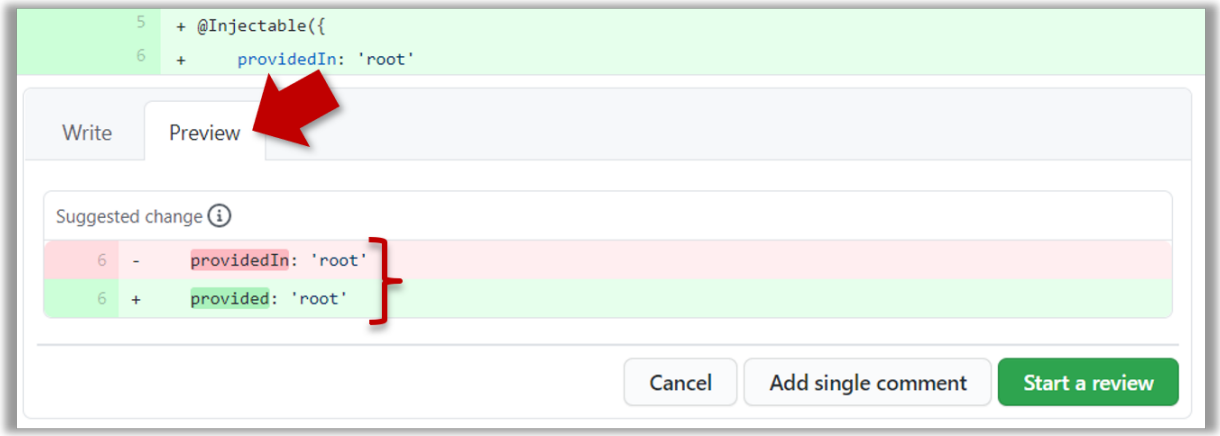
1. Hover over the line you want to comment on and click on the ➕ icon that appears on the left margin. That should create a text box for you to enter your comment.
   - 💡 To give a comment related to multiple lines, click-and-drag the ➕ icon. The result will look like this:



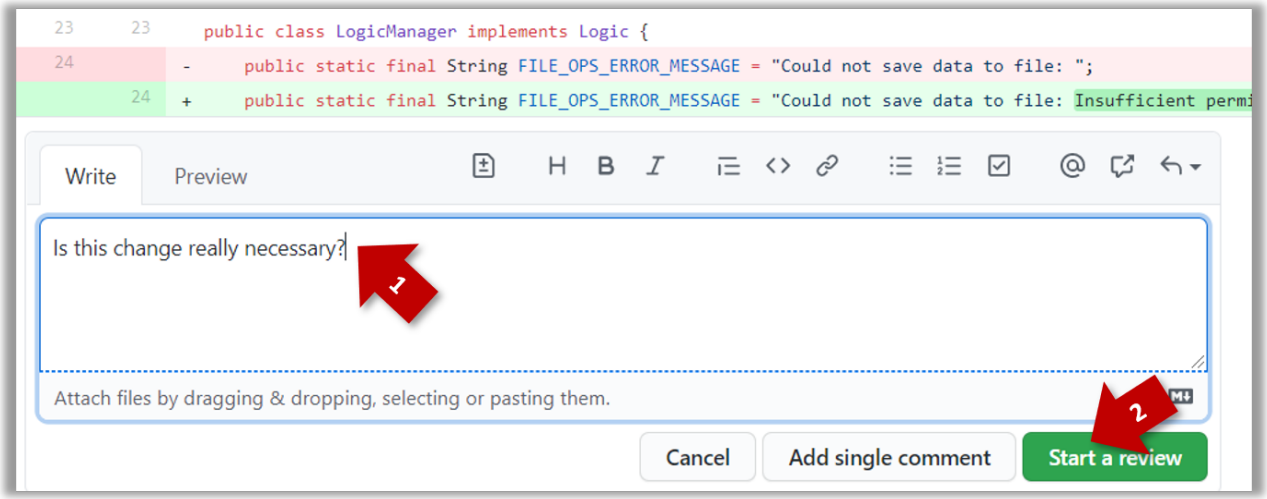
2. Enter your comment.
   - 📎 This page @SE-EDU/guides has some best practices PR reviewers can follow.
   - 💡 To suggest an in-line code change, click on this icon:



The comment will look like this to the viewers:

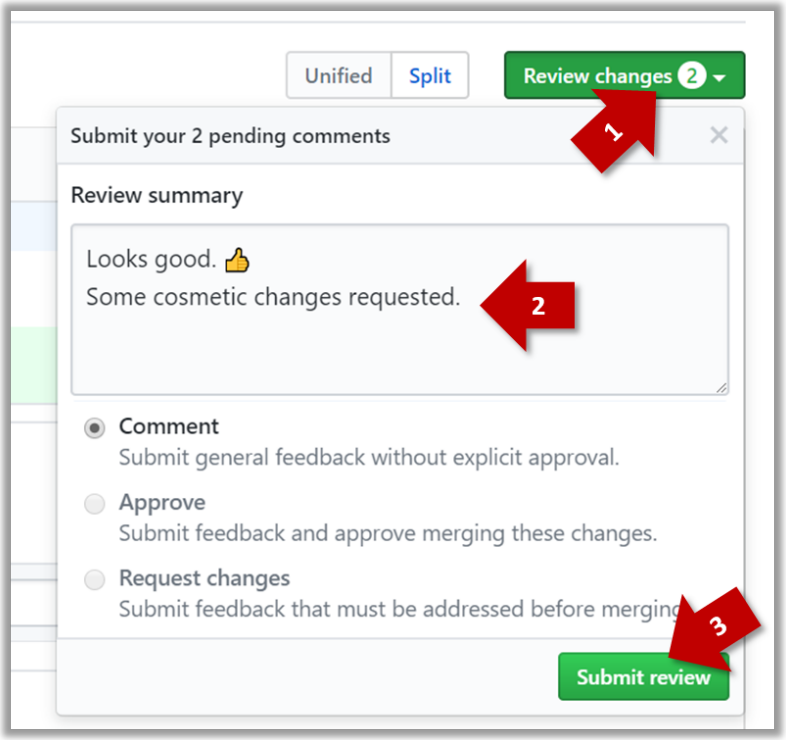


3. After typing in the comment, click on the `Start a review` button (not the `Add single comment` button. This way, your comment is saved but not visible to others yet. It will be visible to others only when you have finished the entire review.



4. Repeat the above steps to add more comments.

5. Submit the review:



1. When there are no more comments to add, click on the `Review changes` button (on the top right of the diff page).
2. Type in an overall comment about the PR, if any. e.g.,

```
Overall, I found your code easy to read for the most part except a few places
where the nesting was too deep. I noted a few minor coding standard violations
too. Some of the classes are getting quite long. Consider splitting into
smaller classes if that makes sense.
```

`LGTM` is often used in such overall comments, to indicate `Looks good to merge`.
`nit` is another such term, used to indicate minor flaws e.g., `LGTM, almost. Just a few nits to fix.`.

3. Choose `Approve`, `Comment`, or `Request changes` option as appropriate and click on the Submit review button.

Merging PRs

★★☆☆ Can review and merge PRs on GitHub

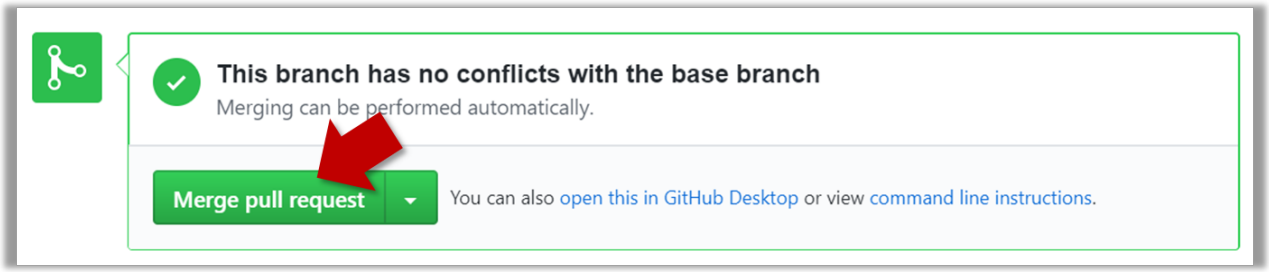
Let's look at the steps involved in merging a PR, assuming the PR has been reviewed, refined, and approved for merging already.

Preparation: If you would like to try merging a PR yourself, you can create a dummy PR in the following manner.

1. Fork any repo (e.g., samplerepo-pr-practice).
2. Clone in to your computer.
3. Create a new branch e.g., (`feature1`) and add some commits to it.
4. Push the new branch to the fork.
5. Create a PR from that branch to the `master` branch in your fork. Yes, it is possible to create a PR within the same repo.

1. Locate the PR to be merged in your repo's GitHub page.

2. Click on the `Conversation` tab and scroll to the bottom. You'll see a panel containing the PR status summary.



3. If the PR is not merge-able in the current state, the Merge pull request will not be green. Here are the possible reasons and remedies:

- **Problem: The PR code is out-of-date**, indicated by the message **This branch is out-of-date with the base branch**. That means the repo's `master` branch has been updated since the PR code was last updated.
  - If the PR author has allowed you to update the PR and you have sufficient permissions, GitHub will allow you to update the PR simply by clicking the Update branch on the right side of the 'out-of-date' error message. If that option is not available, post a message in the PR requesting the PR author to update the PR.
- **Problem: There are merge conflicts**, indicated by the message **This branch has conflicts that must be resolved**. That means the repo's `master` branch has been updated since the PR code was last updated, in a way that the PR code conflicts with the current `master` branch. Those conflicts must be resolved before the PR can be merged.
  - If the conflicts are simple, GitHub might allow you to resolve them using the Web interface.
  - If that option is not available, post a message in the PR requesting the PR author to update the PR.

3. Merge the PR by clicking on the Merge pull request button, followed by the `Confirm merge` button. You should see a `Pull request successfully merged and closed` message after the PR is merged.

- You can choose between three merging options by clicking on the down-arrow in the Merge pull request button. If you are new to Git and GitHub, the `Create merge commit` options are recommended.

Next, sync your local repos (and forks). Merging a PR simply merges the code in the upstream remote repository in which it was merged. The PR author (and other members of the repo) needs to pull the merged code from the upstream repo to their local repos and push the new code to their respective forks to sync the fork with the upstream repo.

Forking Workflow

★★★☆ 🏆 Can follow Forking Workflow

You can follow the steps in the simulation of a forking workflow given below to learn how to follow such a workflow.

ℹ️ This activity is best done as a team.

Step 1. One member: set up the team org and the team repo.

1. **Create a GitHub organization** for your team. The org name is up to you. We'll refer to this organization as *team org* from now on.
2. **Add a team** called `developers` to your team org.
3. **Add team members** to the `developers` team.
4. **Fork** se-edu/samplerepo-workflow-practice to your team org. We'll refer to this as the *team repo*.
5. **Add the forked repo to the `developers` team.** Give write access.

Step 2. Each team member: create PRs via own fork.

1. **Fork that repo** from your team org to your own GitHub account.
2. **Create a branch** named `add-{your name}-info` (e.g. `add-johnTan-info`) in the local repo.
3. **Add a file** `yourName.md` into the `members` directory (e.g., `members/jonhTan.md`) containing some info about you into that branch.
4. **Push that branch to your fork**.
5. **Create a PR** from that branch to the `master` branch of the team repo.

Step 3. For each PR: review, update, and merge.

1. **[A team member (not the PR author)] Review the PR** by adding comments (can be just dummy comments).
2. **[PR author] Update the PR** by pushing more commits to it, to simulate updating the PR based on review comments.
3. **[Another team member] Approve and merge** the PR using the GitHub interface.
4. **[All members] Sync your local repo (and your fork)** with upstream repo. In this case, your *upstream repo* is the repo in your team org.
    - The basic mechanism for this has two steps (which you can do using Git CLI or any Git GUI):
      (1) First, pull from the upstream repo -- this will update your clone with the latest code from the upstream repo.
      (2) Then, push the updated branches to your fork. This will also update any PRs from your fork to the upstream repo.
    - Some alternatives mechanisms to achieve the same can be found in this GitHub help page.
      If you are new to Git, we recommend that you use the above two-step mechanism instead, so that you get a better view of what's actually happening behind the scene.

Step 4. Create conflicting PRs.

1. **[One member]: Update README:** In the `master` branch, remove John Doe and Jane Doe from the `README.md`, commit, and push to the main repo.
2. **[Each team member] Create a PR** to add yourself under the `Team Members` section in the `README.md`. Use a new branch for the PR e.g., `add-johnTan-name`.

Step 5. Merge conflicting PRs one at a time. Before merging a PR, you'll have to resolve conflicts.

1. [Optional] A member can inform the PR author (by posting a comment) that there is a conflict in the PR.
2. **[PR author] Resolve the conflict locally**:
    1. Pull the `master` branch from the repo in your team org.
    2. Merge the pulled `master` branch to your PR branch.

    3. Resolve the merge conflict that crops up during the merge.
    4. Push the updated PR branch to your fork.
3. **[Another member or the PR author]: Merge the de-conflicted PR**: When GitHub does not indicate a conflict anymore, you can go ahead and merge the PR.