# Version Control, Git, and GitHub
Software Engineering for Scientist

Version control is a system that manages changes to files for a project. The basic functionality of a version control system includes:
- keeping track of changes
- synchronizing code between developers and users
- allowing developers to test changes without losing the original
- reverting back to an old version
- tagging specific versions

Git is one of many version control systems. **GitHub** is one of a few companies that host software repositories using Git. You can have version control without Git, you can use Git without GitHub, and you can use GitHub without *really* using Git or version control.

Example workflow

Install Git https://mac.github.com https://windows.github.com
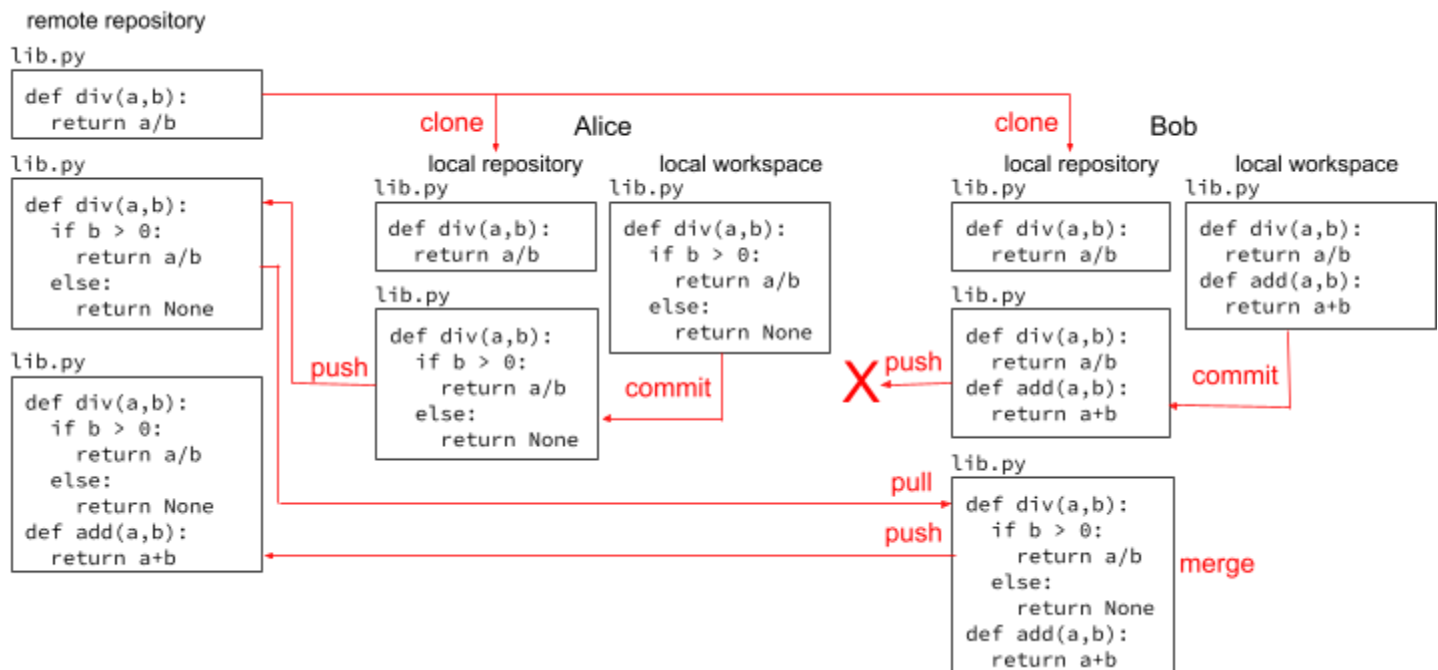
Make a GitHub account https://github.com/

Create a repository
To start a new project in GitHub, you need to create a repository. Public repositories will be listed under your GitHub account, and there is a unique URL associated with each repo (github + your user name + repo name). GitHub repos are the home pages for
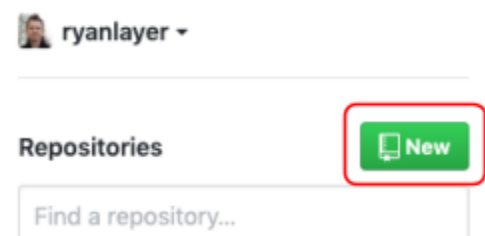
many open-source software projects, and it is not uncommon to see a repo URL in the abstract of high-impact journals.

Next, pick (1) a repo name, (2) a short description, and (3) choose if you want the repo to be public or private. If you make it private, then only users that you have granted access to will be able to see the project. At this point, you can also add (4) a mostly-blank README file and (5) a software license. These are optional, and you can add, remove, and change these files in the future. You can even change the repo name and who owns the repo. Once you fill out this form, hit "Create repository."

The resulting repository can be accessed at `github.com/<YOUR ACCOUNT NAME>/<REPO NAME>` (e.g., https://github.com/ryanlayer/software-project), which includes information about the repo and its history, management tools, a simple file browser, and a rendering of the markdown file `README.md` (https://guides.github.com/features/mastering-markdown/).

Clone the repository

Cloning a repo copies all of the contents of the remote repo, including the full revision history, to a local workspace. The local copy is a fully functional version control system. Click on "Clone or download" to get the URL for this repo then run `$ git clone <URL>`

This command will create your local workspace in a new directory named after the repo. Changes to files in this directory have no effect on the remote repository until changes are "pushed."

Recommend workflow branch and merge

To understand Git you must understand branching.

Let's consider the version control you almost certainly used when writing a term paper, where you start writing in a file called `term.doc`. That file is a branch. As long as you keep saving changes to `term.doc`, the branch is progressing. At some point, you have an idea on how to make the paper better but you are not sure if it will work so you make a copy of `term.doc` and name it `term.new_idea.doc`. This is a new branch, and you can modify `term.new_idea.doc` without fear of losing anything in `term.doc`. Ideally, you would merge the changes that you like back into `term.doc`, which would result in the following branch diagram:

In practice, this is a pain so you just abandon `term.doc` and by the end, your final draft is named `term.new_idea.test.old.new.new.final.old.doc`.

Git's main advantage over other version control systems is that branching and merging are easy enough that all changes, from new features to bug fixes, can (should) be done in a branch. Once the feature is complete or the bug is fixed and tested, changes are merged back into the main branch, which in Git is called `main`.

Let's look more closely at how Alice should have made her changes:
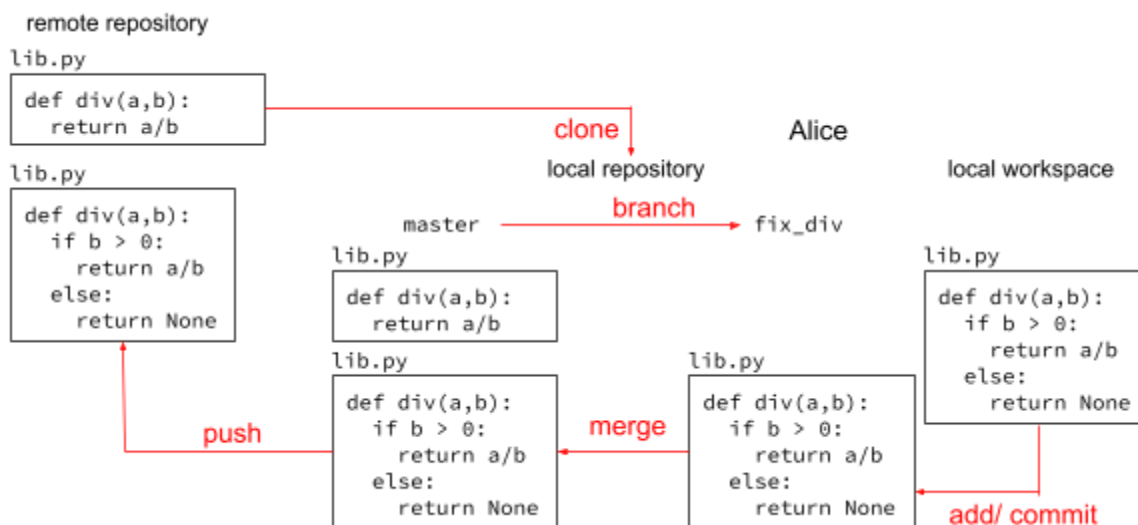
Create a new branch

```
$ git checkout -b fix_div main
Switched to a new branch 'fix_div'
```

After making changes add file and commit the changes to `fix_div` branch

```
$ git add lib.py
```

3

```
$ git commit -m "handle zero denominator"
[fix_div 7b6bb77] handle zero denominator
 1 file changed, 2 insertions(+)
 create mode 100644 lib.py
```

Merge those changes with the local `main` branch and remove the local `fix_div` branch

```
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
$ git merge --no-ff fix_div
Merge made by the 'recursive' strategy.
 lib.py | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 lib.py
$ git branch -d fix_div
```

Then update the remote `main` branch

```
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 437 bytes | 437.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/alice/math_libs.git
   b5c330d..d5d05c5  main -> main
```

Push / pull conflicts

A push can fail if the remote repo has been changed since your last pull.

```
$ git push origin main
To https://github.com/ryanlayer/software_project.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/alice/math_libs.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

In many cases this error is fixed by running `git pull` and allowing git to handle the merge automatically, but not always.

```
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/alice/math_libs
   03d8120..1a96897  main       -> origin/main
Auto-merging math_lib.py
```

```
CONFLICT (content): Merge conflict in math_lib.py
Automatic merge failed; fix conflicts and then commit the result.
```

When the automatic merge fails, git will flag the conflicting lines. Between `<<<<<<< HEAD` and `=======` are the lines from the local branch, and between `=======` and `>>>>>>> 1a968` are the lines from the remote branch. Resolving conflicts involves manually fixing these lines.

math_lib.py

```python
def div(a, b):
<<<<<<< HEAD
    return a/b

def add(a, b):
    return a+b
=======
    if b == 0:
        return None
    else:
        return a/b
>>>>>>> 1a968974635128c2009b7355181abb48a4af2e7c
```

In this case the result may be:

math_lib.py

```python
def div(a, b):
    if b == 0:
        return None
    else:
        return a/b

def add(a, b):
    return a+b
```

When all of the conflicts are resolved, the modified files should be added and committed.

```
$ git add math_lib.py
$ git commit -m "resolved merge conflicts"
[main d00da46] resolved merge conflicts
$ git push origin main
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 618 bytes | 618.00 KiB/s, done.
Total 6 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/alice/math_libs.git
```

```
   1a96897..d00da46  main -> main
```

One issue with this model is that the remote `main` branch is changed without passing any tests or any review of the changes. A more conservative process involves **pull requests**. In this case, Alice would push her `fix_div` branch to the remote repo, and then initiate a pull request. That pull request would then be reviewed and merged if no issues are found. If there is an issue, then Alice would fix her local `fix_div` branch and push the updated branch to the remote repo. These changes will update the existing pull request. While this process has a few more steps, it reduces the likelihood that bad code makes its way into the remote repo, especially for repositories configured with continuous integration tests which will run on all pull requests. Pull requests are very important when multiple developers are working on the same project and when accepting updates from the community.

To issue a pull request, making changes, add the file, commit the changes to `fix_div` branch, and push the branch to the remote repo

```
$ git add lib.py
$ git commit -m "handle zero denominator"
[fix_div 7b6bb77] handle zero denominator
 1 file changed, 2 insertions(+)
 create mode 100644 lib.py
$ git push origin fix_div
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 575 bytes | 575.00 KiB/s, done.
Total 5 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
remote:
remote: Create a pull request for 'fix_div' on GitHub by visiting:
remote:      https://github.com/ryanlayer/alice/pull/new/fix_div
remote:
To https://github.com/alice/math_libs.git
 * [new branch]      fix_div -> fix_div
```

That new branch will now show up on the repo's GitHub page. Select `fix_div` to switch to the branch, and select Compare & pull request to start the pull request process. The next screen provides a text box to describe what changes the pull request includes and any other relevant information. Once that form is filled out select Create pull request.
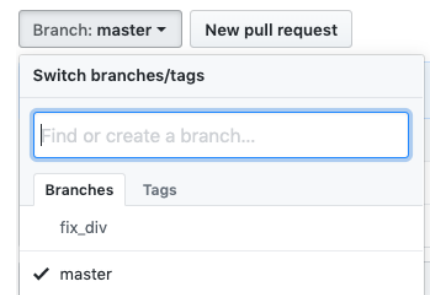
A full list of pull requests can be viewed in the Pull requests tab

The default check looks for any code-level conflicts. The results of any other tests that have been added to the repo will also show up here. You can also look at which files and lines of code were changed. If everything looks good select Merge pull request, and the changes will be made to the remote `main` branch.
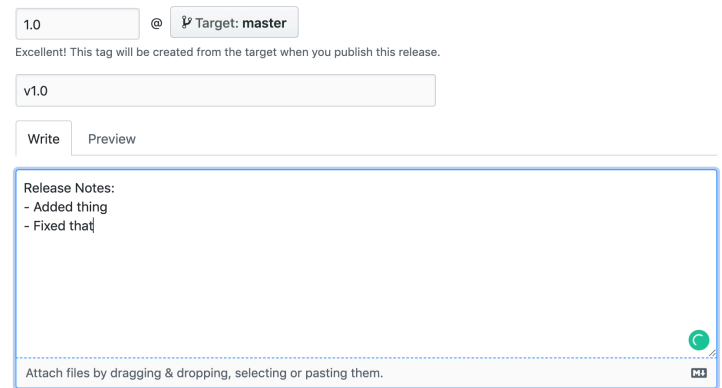
At this point, Alice's local `main` is now out of date, and she must switch to her `main` branch pull the latest changes before creating new branches and adding new features.

```
$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)
$ git pull
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
From https://github.com/ryanlayer/software-project
   d5d05c5..66ebd5d  main       -> origin/main
Updating 09a387e..66ebd5d
Fast-forward
 lib.py | 5 ++++-
 1 file changed, 4 insertions(+), 1 deletion(-)
```

Tagging releases

As you add a significant feature or fix a major issue, it is a good idea to tag a release, which is similar to versions. Tagging marks a specific point in the repo's history so that users can know when they should upgrade their software and so that other developers using your project can better maintain their dependencies. Also, tagging is easy. From the repo GitHub page, select the Releases tab, then Create new release / Draft new release, fill out the form, and Publish. Tags are listed under the Release tab, and users and either download the repository through two autogenerated files (`.zip` or `tar.gz`).