VERSION CONTROL TUTORIAL

`Git`, *GitHub*, branches, merge conflicts, remote repositories, pull requests

Roman Gerasimov

1 Windows setup

`Git` is a command line application available for all major operating systems including *Windows*, *Mac OS* and *Linux*. The latter two may be commonly referred to as *Unix-like* operating systems. They share similar file structures (to be discussed below) and provide default POSIX-compliant (or nearly POSIX-compliant) terminals, which means that there is a certain standard format in which terminal commands are keyed in. On the other hand, *Windows* operating systems use a different (non-POSIX) command line by default and employ a different file structure. While it is possible to use `Git` on *Windows* natively, doing so is fairly uncommon and will not be covered in this tutorial. Instead, the reader using *Windows* is advised to install a third-party application that provides a Unix-like environment. We recommend installing Ubuntu (as in, the *Windows 10* application; not the operating system of the same name, although it would work too) which is available free of charge in Microsoft Store. An alternative commonly used application that serves a similar purpose is CygWin, although it is arguably harder to configure (ask your mentor for help if necessary!).

To use Ubuntu, the *Linux* subsystem must be enabled in the system settings first. Go to Settings » Apps » Programs and Features (hyperlink on the right), then click *Turn Windows features on or off* on the left, check Windows Subsystem for Linux and click OK. The configuration may take a few minutes and request a system restart which is recommended.

After the restart, the Ubuntu app should be available through search. On first start, the app may take a few minutes to install itself[1]. Once launched, a terminal window will show up which should be indistinguishable to that of a proper Unix-like system for the purposes of this tutorial. The terminal may ask you to enter a username and a password before allowing access to the command line.

The environment may not have Python installed by default, which can be checked by running:

[1]If installation fails, try resetting the app in its settings and re-launching it again. If that does not resolve the problem, seek help from your mentor

```
python --version
```

If the command above produces an error instead of the Python version, you can install minimal Python with the following commands:

```
sudo apt-get update
sudo apt-get install python
```

The commands may request the password chosen when setting up Ubuntu (this is not your main *Windows* password). When the installation completes, double check that Python is working by requesting its version again. Note that we are installing Python in the Ubuntu environment purely for convenience in this tutorial: for all *real* application you should use the version you already have installed in your native *Windows* environment.

Access to files
The regular *Windows* volumes (such as `C:\`) can be accessed from Ubuntu at `/mnt/` (more on file structure and paths later)

---

2 Installation

On most *Linux* distributions, `Git` will be installed by default, which also applies to the environment provided by the Ubuntu app on *Windows*. On *Mac OS*, `Git` may not be installed by default but will be installed at first request. To check that your `Git` is functioning properly, try running the following command:

```
git --version
```

A properly installed `Git` will display its version number (e.g. `2.25.1`).

After confirming that your `Git` functions as expected, run the following commands:

```
git config --global user.name "<username>"
git config --global user.email "<email>"
git config --global core.editor "nano"
```

where you must replace `<username>` and `<email>` with your username and Email address that should ideally match the ones used in your *GitHub* account. The last of the three commands is optional as it changes the default text editor to an application called `nano`. The default text editor used by `Git` – `vim` – is considerably harder to use for an inexperienced user.

3 Command line overview

In this document, we will mostly use `Git` through its command line interface. Only basic command line expertise is expected of the reader which is briefly reviewed here. As described in the previous section, this document deals entirely with `POSIX`-compliant environments.

You may already know that the data stored on the computer are arranged into *files*, each identified by its file name such as `readme.txt`. Typically, file names consist of two parts separated by period (`.`) known as the *stem* (in this case, `readme`) and the *extension* (`txt`). The stem is arbitrary and is typically chosen to describe the purpose of the file. The extension identifies the type of the file, such as `txt` for plain text, `py` for Python code, `jpg` for photographs, `mp3` for music etc. Note that this structure is entirely conventional: you may create and use files without extensions, with multiple extensions or with non-standard extensions.

Since a typical workspace contains millions of individual files, they are arranged into directories[2], which may in turn contain sub-directories up to an indefinite number of layers. Conventionally, directory names do not have extensions. The outermost directory is called *root* directory and is denoted with a single forward slash (`/`). If the aforementioned file – `readme.txt` – happens to be stored in the root directory, it may be referred to simply as `/readme.txt`. If, instead, the file is placed in a sub-directory called `text_files` which is in turn placed in the root directory, the correct reference is `/text_files/readme.txt`. In case of multiple levels, the names of sub-directories can be stacked indefinitely separated by forward slashes. For example:

```
/home/roman/text_files/readme.txt
```

The above *path* refers to a file called `readme.txt` that is stored in a directory called `text_files`, which may in turn be found in the directory called `roman`, which is in turn placed in the directory called `home` which is in the root directory. Paths like this that start with the root directory and list all sub-directories up to the target file are called *absolute*. With a large number of levels they may get too unwieldy to work with, which is why *relative* paths are often used instead.

A relative path is specified in relation to some directory chosen by the user known as the *working* directory. The working directory can be referred to with the placeholder name `.` (period). For instance, in the example above the relative path to `readme.txt` for the case of `/home/roman/text_files` being the working directory is:

```
./readme.txt
```

If, on the other hand, the working directory is `/home`, the relative path would be:

```
./roman/text_files/readme.txt
```

For simplicity, the characters "`./`" at the beginning of the relative path can be omitted:

[2]While files and directories appear distinct to the user, on a fundamental level directories are, in fact, ordinary files that store locations of other files

```
roman/text_files/readme.txt
```

Note that the example above does **not** begin with a forward slash which tells the system that this is a relative rather than absolute path.

Suppose that there exists a file whose absolute path is `/home/roman/song.mp3`. If the working directory is `/home/roman/text_files`, the relative path to the file is:

```
../song.mp3
```

The double period notation (`..`) represents the parent directory of the working directory. Multiple instances of `..` can be used to construct a relative path to any file in the system. For example, the following relative path

```
../../../my_file.dat
```

refers to the file `/my_file.dat` in the root directory if `/home/roman/text_files` is the working directory.

On most Unix-like systems, each user is assigned a personal directory for their files known as their *home* directory. In my case, the path to my home directory is `/home/roman`, which may differ on other operating systems and for other users. However, the home directory can always be accessed with the `~` shorthand, such as

```
~/text_files/readme.txt
```

The system will replace `~` with the correct path to the home directory of the active user regardless of the current working directory.

It is now time to practice navigating the file system in the terminal. Open a new terminal window and run the following command:

```
pwd
```

`pwd` (print working directory) displays the current working directory on the screen as the name suggests. By default, this will likely be your home directory which is `/home/roman` in my case (on *Mac OS* you might see `Users` instead of `home`).

You can view all files and directories in the working directory by running the following command:

```
ls
```

If you use the Ubuntu app on *Windows*, the command above might return nothing at all since the home directory will be empty on a fresh setup. You can get a more representative output by listing the root directly instead of the current working directory with `ls /`.

By default, the `ls` (short for *list*) command only lists the names of files and directories and not much else. To see more information, run this command with the `-l` flag:

```
ls -l
```

As before, you can list the root directory with `ls / -l` in case your working directory has no files. Now, the output is a table with multiple columns. For example, you may see something resembling the following:

```
drwxr-xr-x 22 roman roman   4096 Aug 11  2019  anaconda3
-rw-r--r--  1 roman roman      0 Jul  5 23:02  readme.txt
```

In this example, there are two items in the working directory named `anaconda3` and `readme.txt` as listed in the rightmost column of the output. The first character in each line tells us whether the item is a directory (`d` as is the case for `anaconda3`) or a file[3] (`-` as is the case for `readme.txt`). The rest of the first column (e.g. `rwxr-xr-x`) specifies access permissions that will not be discussed here. Disregarding columns 2 through 4, the fifth column displays the size of each item in bytes. The size of `readme.txt` appears to be 0 bytes, which implies that this file is empty. The size of `anaconda3` appears to be 4096 bytes and must be understood as the size of the directory itself and **not** its contents[4]. Finally, the sixth column stores the date of last access.

Finally, Unix-like systems often implement so-called *hidden* files and directories, which are files and directories not revealed by the `ls` command by default. Such files are identified by file names that start with a period (e.g. `.bashrc`). In some sense, hidden files only have extensions and not stems. `Git` uses hidden files for a variety of purposes. To reveal hidden files, you must run `ls` with the `-a` flag. E.g.

```
ls -a
```

As opposed to the regular `ls` call, you should now see multiple files and directories whose names begin with a period including the shorthands `.` and `..` that refer to the working directory and its parent directory respectively. You may also combine multiple flags (e.g. `-l` and `-a`) as follows:

```
ls -la
```

You can create a new empty file in the working directory with the `touch` command:

```
touch my_first_file.dat
```

which creates a new blank file called `my_first_file.dat` (you can choose any stem and extension). Upon creating the file, use the `ls` command with relevant flags to verify that the file has been created and is 0 bytes in size.

To create an empty directory, use the `mkdir` command. For example:

```
mkdir my_first_directory
```

[3] In addition to `-` and `d` you may also run into some `l`, `b` and more. Those are special files that will not be discussed here, but see online.

[4] The file size of a directory is typically determined by the file system configuration (see StackExchange). Run `du -s <path_to_directory>` to get the total size of the contents of the directory rather than the directory itself (`du` stands for *disk usage* and the `-s` flag specifies that we only want to see the summary of the disk usage analysis rather than a detailed report)

We can now change the working directory to be the newly created directory with the cd command:

```
cd my_first_directory
```

Double check that the working directory has changed with the pwd command and that it is empty with the ls command (should you run ls with the -a flag, you may still see . and .. even if the directory is empty). You can change the working directory back either with relative paths:

```
cd ..
```

or by using the home directory shorthand:

```
cd ~
```

Once back in the home directory, you can delete my_first_file.dat and my_first_directory with the rm (*remove*) command as follows:

```
rm my_first_file.dat
rm -r my_first_directory
```

Note that the -r flag is necessary when deleting directories. We recommend that you master other basic commands such as cp (copy files and directories) and mv (move and rename files and directories) using official documentation if necessary. The documentation for most commands can be obtained by running

```
<command_name> --help
```

where <command_name> is replaced with the command of interest such as ls.

4 Starting a new project

To initiate a new project with version control, we must create an empty directory and ask Git to create a *repository* for it, i.e. a place where all the *versions* of the project will be stored. First, create a new directory called my_project and make it the current working directory:

```
cd ~
mkdir my_project
cd my_project
pwd
```

The commands above change the working directory to the home directory (in case that is not where we already are), create a new directory, change the working directory to it and double check that the change has been made with the pwd command (not necessary). We now enable version control for this directory with the following command:

```
git init
```

If the command above ran successfully, a new hidden directory called `.git` must appear in the working directory which may be seen with `ls -a` (but not `ls`). We may now create new files in this directory as we please. For example, we can create a blank file called `hello_world.py`:

```
touch hello_world.py
```

You may also put some simple Python code in this file for completeness. You can use a graphical text editor of your choice or a number of text editors available in the terminal. Arguably the most user-friendly terminal text editor is `nano` which may be launched as follows:

```
nano hello_world.py
```

You may then use your keyboard to type some simple Python code *into the file* such as:

```
print('Hello World!')
```

Press ctrl+O to save the changes. The editor will ask you to choose the name for the file defaulting to `hello_world.py`. Accept the default by pressing ↵. Finally, use ctrl+X to quit `nano` and return back to the terminal. You can check that the file was saved correctly by running it:

```
python hello_world.py
```

The text "*Hello World!*" should be printed onto the screen.

By creating a new file and filling it with content we produced a new *version* of the project. We may now want to save this new version in the repository. Run the following command to see what `Git` "thinks" of our project at the moment:

```
git status
```

The output should be similar to the following:

```
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        hello_world.py
nothing added to commit but untracked files present (use "git add" to track)
```

Notice that `Git` sees the newly created file `hello_world.py` and the file is *untracked*, which means that `Git` does not consider this file to be a part of the new version of the project. In order to include this file, we must explicitly tell `Git` to track it as follows:

```
git add hello_world.py
```

If we run `git status` again, the file will be listed as “new file” rather than an untracked file (try it!). In principle, we can continue editing the project and enabling tracking for all individual changes. Once we feel that the new version is ready to be added to the repository, we can tell `Git` to *commit* the changes as follows:

```
git commit -m "My first commit" -m "Added a new Python script that prints
 ↪  Hello World"
```

Here, the `-m` flag is used twice with a string argument each time. The first argument contains the title of the commit and the second contains its description. Both must make it clear why this commit is being made.

And this is it! A new version of the project (a *commit*) has been created in the repository. Try running `git status` one more time to make sure that there are no untracked files or changes to be committed. To make things more interesting, use `nano` (or any other text editor) to introduce another change to `hello_world.py`. For example, I changed the printed line from “Hello World!” to “Goodbye World”. Once again, run `git status` to look at the project through the eyes of `Git`: you should see that the file has been modified but is not yet *staged* for a commit, which is equivalent to saying that the new change to the file is not yet tracked. Use the `git add` command as before to *stage* the change. Note that when `git add` is applied to a directory, it automatically applies to its contents as well, which means that instead of specifying the name of the file, you can run

```
git add .
```

to stage all changes at the same time. Use `git status` once more to confirm that `hello_world.py` is now listed as modified and to be committed. Finally create a second commit with an appropriate title and description. For example:

```
git commit -m "My second commit" -m "Changed Hello World to Goodbye World"
```

If everything went well, the repository now has two separate commits. The history of commits may be viewed with the following command:

```
git log
```

In my case, I get the following output:

```
commit 37d9d692d7f564274c86033d93bb7f7a957edf7d (HEAD -> master)
Author: Roman Gerasimov <romang@ucsd.edu>
Date:   Tue Jul 6 00:44:35 2021 -0700
    My second commit
    Changed Hello World to Goodbye World

commit 0bc195d10bd72f14aeaca30016f56e3521e2bda2
Author: Roman Gerasimov <romang@ucsd.edu>
Date:   Tue Jul 6 00:29:44 2021 -0700
    My first commit
    Added a new Python script that prints Hello World
```

Both commits can be seen. The first line stores the long and cumbersome *hash* of the commit (e.g. `37d9d692...`). You can think of the hash as the unique identifier of each commit. `(HEAD -> master)` indicates that this commit represents the most recent version of the project (so-called *head*). Finally, we get the author of the commit as well as the date when the commit was made and the description.

A particularly useful command to remember is `git blame`. For example:

```
git blame hello_world.py
```

will display which commit introduced each line of the code as well as when and by whom.

5 Branches

Let's say that we now want to introduce a new feature to the project. For example, in addition to printing "Hello World" (or "Goodbye World" in the most recent version) we also want the script to do some basic calculation such as adding 2 and 2 to get 4. In principle, you can follow the procedure above one more time and create a third commit; however, there may be a number of reasons why you would want to keep your work on the new feature separated from the main version history. For example:

- The main version is used by other people and you do not want them to deal with an unfinished new feature.
- You would like to keep the work on new feature separated to easily compare the functioning of the code with the new feature and without it.
- You need your line manager to approve the new feature before it can be added to the main version of the application.

For any of the reasons above and more, you can create a new *feature branch*. All subsequent commits of your work on the new feature will then go into this new branch without disturbing the main branch. Finally, when the new feature is ready and has been fully tested, the new branch can be merged with the main branch.

In `Git`, the main branch is called `master`[5]. When a new repository is created with `git init`, it will be the only existing branch and all commits will go into it by default. You can verify that we are currently on branch `master` by running `git status`. To create a new branch, run the following command:

```
git checkout -b feature/add_two_and_two
```

Here the `-b` flag tells `Git` that we want to create a new branch and `feature/add_two_and_two` is the name of the new branch. In principle, any name can be chosen; however, the standard

[5]While the current version of `Git` still uses the name `master` for the main branch, this convention is likely to change in the future due to its racial context. Possible future defaults are `primary` or `main`. The latter is already used as default for repositories managed by *GitHub*

convention is to prefix feature branches with `feature/` and use names that are meaningful. Use `git status` once more to verify that we are on the new branch and not on `master`.

We can now follow the same procedure to create a new commit. Use a text editor to add the new feature to the file as follows:

```python
print('Hello World!')
print('2+2=', 2 + 2)
```

stage the change and commit it with appropriate title and description. Finally, check that the new commit is in place with `git log`. You can now switch between the two branches using the following syntax:

```
git checkout <branch_name>
```

where `<branch_name>` is the name of the branch. As you switch between the two branches, make sure that `git status` displays the correct branch, `git log` shows the third commit disappearing on `master` and reappearing on the feature branch and the Python script printing or not printing `2+2=4`.

Once you are satisfied with the new feature, it is time to merge the feature branch into `master`. To do so, checkout `master` and run

```
git merge feature/add_two_and_two
```

After the above command, the commits from the feature branch are added to `master` and both branches should be identical which you can check with `git log` as well as by running the Python script.

6 Merge conflicts

The aforementioned example of branch workflow corresponds to the ideal case scenario where a new branch is created and developed while the `master` branch patiently waits for the feature branch to stop growing and merge itself in. This will not always be the case when multiple people (or a single poorly organized person) are working on the same project. In such case, the `master` branch may *diverge* from the feature branch through other users adding commits to it or merging other feature branches into it. We will now simulate such scenario to see what happens. First of all, let's "unmerge" the feature branch from `master` which we can do by simply deleting the last commit. Switch to `master` (if you are not already) and run the following:

```
git reset --hard HEAD^
```

Here, we use the `reset` command to reset the branch to some previous state. In this case, we want to reset the branch to a state when the head of the branch was one commit earlier than it is now (`HEAD^`). The `--hard` flag removes the commit from the branch entirely. Recovering a commit deleted in this way is not trivial and hard resets must be done with caution. In this case, it is okay to delete the third commit as it will remain in the feature branch. After running the command above, the `master` branch should revert to the state it was before the merge (check that!).

This time, instead of merging the feature branch right away, let's stay on the `master` branch and create a new file. For example, I created a new file called `readme.txt` and committed it as follows:

```
git checkout master
touch readme.txt
git add readme.txt
git commit -m "Another third commit" -m "This is a third commit on the
↪  master branch that adds a readme file"
```

At this point, both branches have three commits; however, the third commit differs between them! An attempt to merge the feature branch into `master` will work a little different this time:

```
git merge feature/add_two_and_two
```

Instead of `Git` obediently merging the branches in the background, a text editor will launch with the following content:

```
Merge branch 'feature/add_two_and_two'

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

`Git` is no longer able to simply move the third commit from the feature branch into `master` because the two branches have diverged. Instead, it will automatically turn the work we did in the feature branch into a new commit and add that to `master`. In this text editor, `Git` is asking us to provide a title for this automatically generated commit which defaults to `Merge branch 'feature/add_two_and_two'`. You may edit the title or you may leave it as is. At the end, save the changes with ctrl+O followed by ↵ and exit the editor with ctrl+X as before.

Upon running `git log`, you should see five different commits. On my setup, I get the following hash lines (your hashes may be different but the branch designations should be the same):

```
commit da665f89d93ab6fa8704107d0887a362cc649d74 (HEAD -> master)
commit d5ec004b7e656b393a1631edd07d97e5bb6904cb
commit 55df4857086ec238f4e600cbf76d5c8a126ab08a (feature/add_two_and_two)
commit 37d9d692d7f564274c86033d93bb7f7a957edf7d
commit 0bc195d10bd72f14aeaca30016f56e3521e2bda2
```

The head commit is the automatically generated merge commit with the title provided in the text editor. We can see exactly what changes that commit introduces by running the following:

```
git diff HEAD^ HEAD
```

which is asking `Git` to show the difference between the head of the current branch (`HEAD`) and the commit immediately before it (`HEAD^`). The displayed change should be the addition of the $2+2$ calculation merged from the feature branch.

Immediately under the head commit is the original third commit of the master branch that adds the `readme.txt` file. We can see that this is indeed the case by running:

```
git diff HEAD^^ HEAD^
```

which should simply display a message about a new file called `readme.txt` being created.

The third commit from the top, in fact, does not belong to this branch as indicated by `(feature/add_two_and_two)` in the output. This is the third commit from the feature branch that is added here for completeness in order for us to see who created it and when as well as its title and description (the merge commit is missing this information and is, instead, describing when the merge occurred and who did it). The remaining two commits are simply the first two commits we created in the very beginning that are shared by both branches.

The situation considered above with diverged branches is somewhat more complicated; however, `Git` did most of the hard work for us by analyzing the feature branch, distilling the entire feature in it into a single merge commit and adding it to the `master` branch. This is because the feature branch did not contradict the `master` branch in any way (they were modifying completely different files: `readme.txt` in `master` and `hello_world.py` in the feature branch). If the branches happened to contradict each other instead, a *merge conflict* would emerge making the workflow considerably more complicated.

To simulate this situation, let us revert the `master` branch again into the state it was before the merge with

```
git reset --hard HEAD^^
```

This time we are "shaving" two commits off the head: the merge commit created by `Git` and the `readme.txt`-creating commit introduced by us. After the reversion, double check that the `master` branch returned back into its pre-merge state with only two commits. Now let's create a third commit in the `master` branch that contradicts the feature branch. Use the text editor to edit `hello_world.py` to change "Goodbye World!" to "Sleep well World!". Stage the change and create a commit as before. Finally, try merging the branches:

```
git merge feature/add_two_and_two
```

which results in the following output:

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

Resolving merge conflicts can be extremely challenging: do not hesitate to seek help from your mentors! In this particularly simple case however, it is not going to be too hard. Inspect `hello_world.py` in a text editor. You should find the following:

```
<<<<<<< HEAD
print('Sleep well World!')
=======
print('Goodbye World!')
print('2+2=', 2 + 2)
>>>>>>> feature/add_two_and_two
```

Here, `Git` is providing us with two versions of the file: the code between `<<<<<<< HEAD` and `=======` comes from the head of the current branch (i.e. `master`). The code between `=======` and `>>>>>>> feature/add_two_and_two` is from the feature branch. The essence of the conflict is that `Git` does not know which of those versions (or perhaps a combination of both) we intend to have in the end. To resolve the conflict, we must edit the code accordingly and remove all of the text added by `Git`. For example, let's say that we want to use both the new print statement from `master` (`print('Sleep well World!')`) but also the calculation from the feature branch (`print('2+2=', 2 + 2)`). We can edit the file to read as follows:

```
print('Sleep well World!')
print('2+2=', 2 + 2)
```

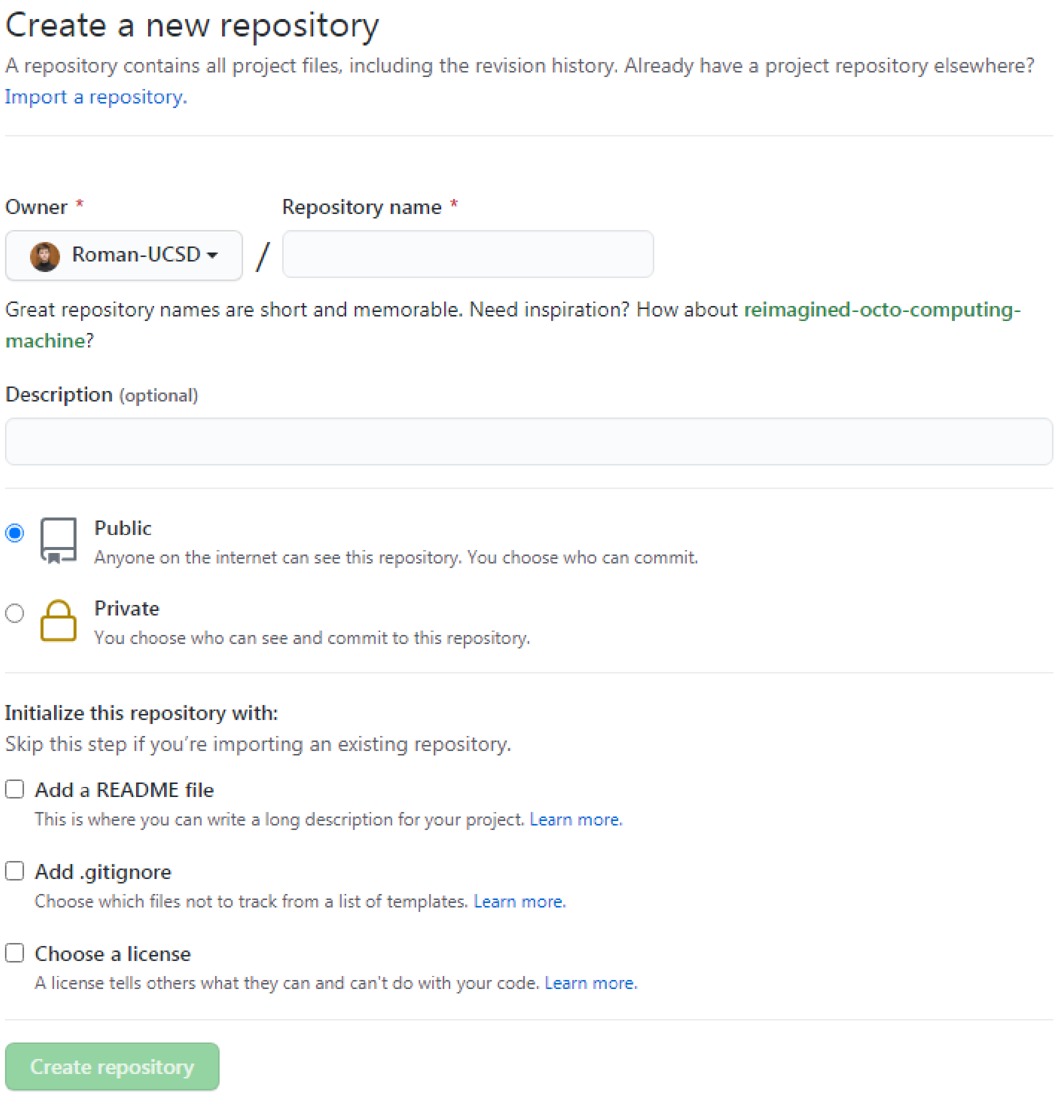
then save the file with ctrl + O, ⏎ and ctrl + X. Lastly, we need to tell `Git` that we're all done:

```
git add hello_world.py
git merge --continue
```

As before, the text editor will open prompting a title for the merge commit. Once provided and saved, a merge commit will be created by `Git` and `git log` should look similar to before with five commits. Double check that the Python script runs as intended, including both a print of "Sleep well World!" and a calculation of 2+2 from the (no longer conflicting) feature branch.

7 Going online

Up to this point, our entire versioning workflow was contained on the local computer. Perhaps the most praised feature of `Git` is the ability for multiple people to collaborate on the same version repository. In a collaborative workflow, each user keeps a copy of the repository on their personal workstation and works with it as described above. However, there also exists the main copy of the repository known as *origin* that is stored on some central server accessible by all collaborators. Every now and then, individual users *push* the changes they make to the local copies of the repository into origin and *pull* changes made by other collaborators from origin.


Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * Repository name *

Roman-UCSD /

Great repository names are short and memorable. Need inspiration? How about reimagined-octo-computing-machine?

Description (optional)

[✓ Public] Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

☐ Add a README file
This is where you can write a long description for your project. Learn more.

☐ Add .gitignore
Choose which files not to track from a list of templates. Learn more.

☐ Choose a license
A license tells others what they can and can't do with your code. Learn more.

Create repository

Figure 1: Online interface for creating a new origin repository on *GitHub*

GitHub is a free web service that will host origin repositories for you and make them accessible to all collaborators over the Internet. First, go to github.com and register a new account. Sign in with your new account and create a new repository (Fig. 1). Choose any name for your repository (e.g. `my_first_repo`) and a meaningful description. You may choose between public repositories that can be viewed (but not changed) by anybody and private repositories that are only accessible to you and collaborators invited by you. Do not check any of the options under *Initialize this repository with*.

After clicking on Create repository, *GitHub* will generate a few terminal commands to *push an existing repository from the command line*. In my case, those commands were:

```
git remote add origin https://github.com/Roman-UCSD/my_first_repo.git
git branch -M main
git push -u origin main
```

Here `https://github.com/Roman-UCSD/my_first_repo.git` is the universal address of your new origin repository on *GitHub* servers that is accessible by your collaborators and you anywhere in the world. The first command links your local repository to the origin at that address. The second command renames the main branch of the local repository from `master` to `main` to comply with the *GitHub* convention. The last command pushes the `main` branch to the origin. For every branch, the last command only needs to be ran once at first push. For all subsequent pushes of that branch, it is sufficient to switch to it and run

```
git push
```

At the moment, only the `main` branch has been pushed to the origin. To push our feature branch as well, run

```
git push -u origin feature/add_two_and_two
```

All subsequent pushes can be done with:

```
git checkout feature/add_two_and_two
git push
```

where the first command is only needed if you are not already on the branch. The message displayed by `git status` should have an extra line:

```
Your branch is up to date with 'origin/feature/add_two_and_two'
```

which tells us that the local and origin copies of the repository are identical.To experiment with interactions between local and origin, try adding a new commit. For example:

```
touch test_file.dat
git add .
git commit -m "Creating a test file" -m "Creating a test file to see how
↪  local and origin interact"
git status
git push
git status
```

Note that before the push, `git status` tells us that we are one commit ahead of origin. After the push, both copies should be up to date. We can also simulate a scenario where the origin is ahead of the local by simply removing the last commit:

```
git reset --hard HEAD^
```

Running `git status` should report that we are, in fact, one commit behind origin. Due to the extra commit on origin, it is no longer possible to push (try it!). Instead, we are expected to pull the latest commit from origin with

```
git pull
```

which should add the deleted head commit right back and bring both copies up to date.

FYI: Force pushes

Ignoring the simulated scenario above, a realistic situation where your local may be behind origin is one where a collaborator creates a new commit on **their** local and pushes it to origin. You are expected to check for new commits on origin and pull them as often as possible; particularly prior to creating new commits to avoid merge conflicts between you and your collaborators. If a conflict does occur, neither pushing nor pulling will be available until the conflict is resolved. Conflicts between local and origin are some of the most common and challenging conflicts in versioning workflow: please contact your mentor to get help resolving them!

In principle, you may add the `--force` flag to your push command to initiate a so-called *force push*. During a force push, the entire copy of the branch at origin is destroyed and replaced by an exact replica of your local copy. In simple terms, a force push means "I couldn't care less about the work of my collaborators, put my work on the remote and forget about theirs". Subject to many jokes, force pushing should never be used in a collaborative environment as it drastically disturbs the workflow for everybody except the collaborator initiating the force push.

To see what a force push looks like, remove the last commit again as in the example above. As before, it is now impossible to push as you are expected to pull the latest commit from the origin instead. However, a push with the `--force` flag will succeed, permanently removing the locally deleted commit from the origin.

---

8 Existing repositories

The previous section described a collaborative workflow for an origin repository owned by us. Let us now consider a different situation where we would like to use a software package created by somebody else that is available on *GitHub*. The most relevant example is SPLAT – the Python code created and distributed by Adam Burgasser and collaborators. The address of the repository is https://github.com/aburgasser/splat.git (you may view it in the web browser if you like).

As opposed to the previous case where we already had a local repository and needed to copy it to origin, here there already exists an origin at the aforementioned address and we need to *clone* it to local. To avoid mixing repositories together, let's first return to the home directory:

```
cd ..
```

and clone the repository:

```
git clone https://github.com/aburgasser/splat.git
```

After a few minutes, a new directory should appear in the working directory called `splat` with the hidden `.git` directory inside (check it!). We want to change the working directory into that new folder:

```
cd splat
```

If our only intention is to merely use the software, we are done! A copy of SPLAT now exists on the computer in the working directory. To access the latest version of the code, we must be sure to stay on the main branch (which in this case is called `master`, check with `git status`) and regularly download updates with `git pull`.

We can also modify the local copy of the repository however we like. For instance, let's create a feature branch and commit some cosmetic change to the codebase:

```
git checkout -b feature/test_feature
touch test_file.txt
git add .
git commit -m "Test commit" -m "Test description"
```

However, an attempt to push this branch will result in a failure:

```
git push -u origin feature/test_feature
```

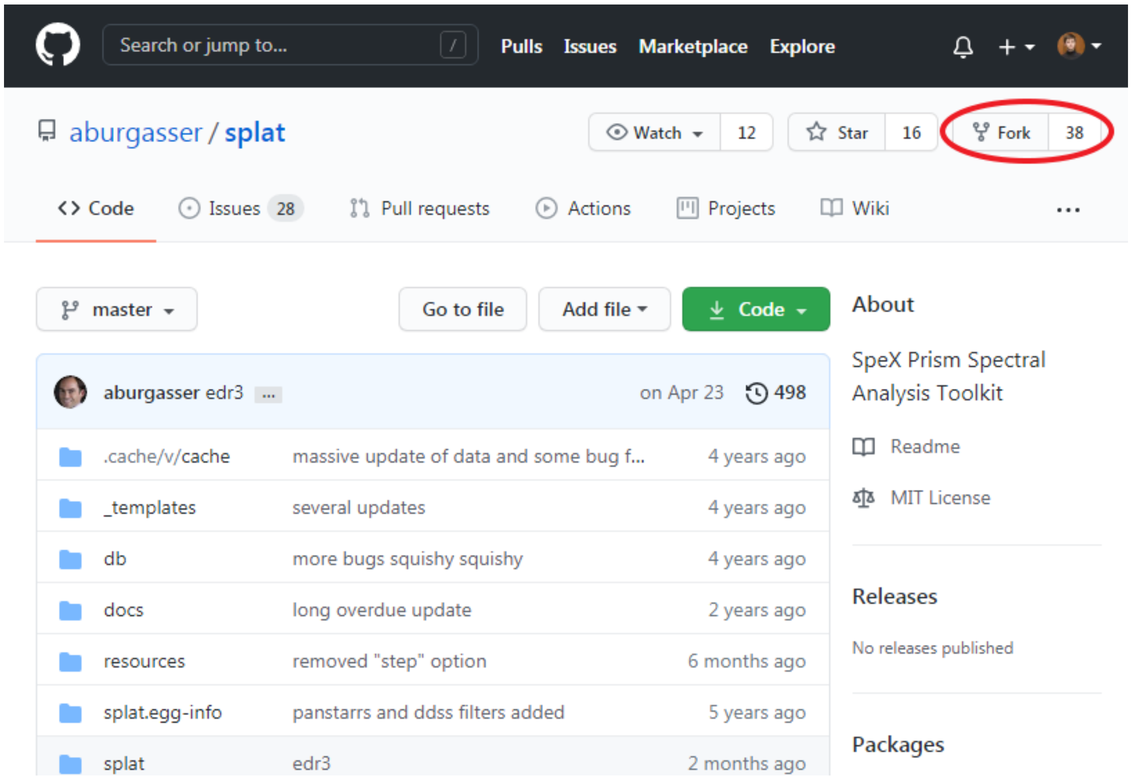
which for me returns:

```
Permission to ... denied to Roman-UCSD
```

This error appears because we are not a registered collaborator on this origin. We can continue altering the repository locally for personal changes. If however you believe that the added feature may be useful to other users, there is a way of submitting your change to the owner of the repository for consideration. To accomplish this, we need to *fork* the origin repository instead of cloning it as we have been doing so far.

A fork is a *GitHub* feature (not `Git`) that allows the user to duplicate the origin of a repository owned by somebody else and gain ownership over the copy. In some sense, a fork is equivalent to downloading the code from an origin belonging to somebody else, creating a new project with a fresh origin and copying the downloaded code in. To fork a repository, open its web page on *GitHub* and click on Fork (Fig. 2). After a few seconds a new repository will be added to your collection on *GitHub* that is an exact copy of the original. Take a note of the address of the new repository displayed in the address bar of the web browser. This new origin repository can be cloned as normal. For example:

```
cd ~
mkdir forked_test_repo
cd forked_test_repo
git clone https://<forked_repository_address>
cd test_repo
```



aburgasser edr3 ... on Apr 23 498

.cache/v/cache	massive update of data and some bug f...	4 years ago
_templates	several updates	4 years ago
db	more bugs squishy squishy	4 years ago
docs	long overdue update	2 years ago
resources	removed "step" option	6 months ago
splat.egg-info	panstarrs and ddss filters added	5 years ago
splat	edr3	2 months ago

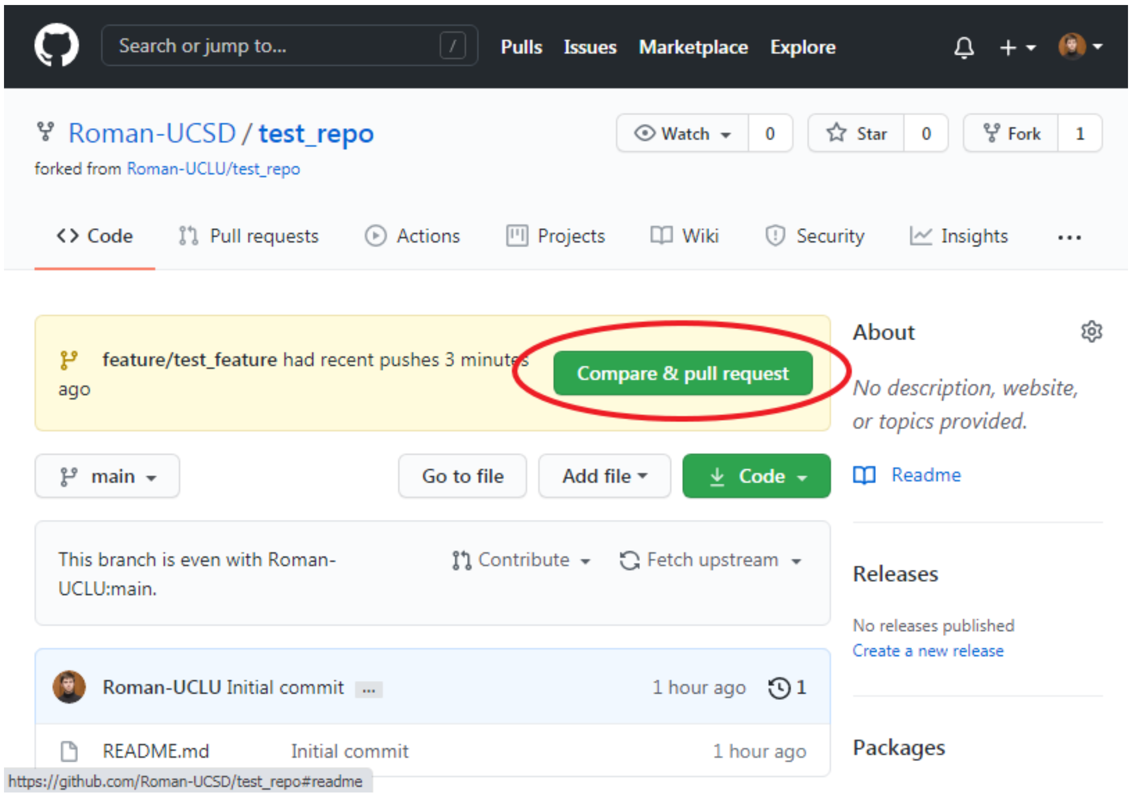
Figure 2: Web page of the SPLAT origin repository on *GitHub*

where `<forked_repository_address>` must be replaced with the address of your new forked origin.

We can now attempt the same edit that did not succeed previously due to us not being the owner:

```
git checkout -b feature/test_feature
touch test_file.txt
git add .
git commit -m "Test commit" -m "Test description"
git push -u origin feature/test_feature
```

The push will now be successful since we own the forked repository. In order to alert the owner of the original repository to our proposed feature branch, go to the web page of the forked repository. *GitHub* identifies the new feature branch automatically and offers us to create a new *pull request* (Fig. 3). When creating a pull request, you will be allowed to come up with a more detailed description of the changes to make a "case" for consideration by the owner of the original repository. Once the pull request is complete, we must wait for the owner to review and approve the changes. The owner will be notified of the new pull request by *GitHub* and will have the option to approve it and merge it into their main branch.


Search or jump to...

Pulls Issues Marketplace Explore

Roman-UCSD / test_repo

forked from Roman-UCLU/test_repo

Watch 0 Star 0 Fork 1

Code Pull requests Actions Projects Wiki Security Insights

feature/test_feature had recent pushes 3 minutes ago

Compare & pull request

main Go to file Add file Code

This branch is even with Roman-UCLU:main.

Contribute Fetch upstream

Roman-UCLU Initial commit ... 1 hour ago 1

README.md Initial commit 1 hour ago

https://github.com/Roman-UCSD/test_repo#readme

About

No description, website, or topics provided.

Readme

Releases

No releases published

Create a new release

Packages

Figure 3: Web page of a test forked repository on *GitHub* immediately after a new feature branch was created and pushed to origin