Class Notes: Intro to version control with Git

Pranav Minasandra

July 10, 2023

Note 1: Throughout this document, commands are written in `typewriter-text`, and must usually be entered as they are in your command-line. Within the typewriter text, some things might need to be modified, and are indicated using `<>`, like so: `command1 option1 <you need to change this>`. Sometimes, commands are long and need to be written in separate lines. When I write these, commands will begin with a `$`, like so:

```
$ meaningof argument1 "Life" argument2 "The Universe" argument3 "Everything"
```

In such cases, you only enter the command after the `$`, but don't have to type the `$` itself.

Note 2: While this document might seem full of command line arguments, in reality, there are just the following ten git commands:

Command	Use
`config`	Sets up things at the beginning
`init`	Starts git up
`add` and `rm`	Add and delete files from tracking
`status`	Displays current status
`log`	Displays history
`branch` and `switch`	Creating and switching between branches
`merge`	Joins two branches
`restore`	Lets you go back to a previous point

Contents

1 What is version control, and why do I need it?

If you write code at all, chances are you don't write it all at once. For any project, software is a slowly evolving, developing entity that hits several milestones, learns from its past, and takes on varying forms. Several times, you might wish to have multiple versions of your code, so that you can try out different ideas and see what works best. Other times, you might want to back up an existing project before trying out a crazy, risky idea. In such cases we often end up with folders like these:

```
Super_Cool_Project/
|--- Super_Cool_Project_FINAL
|--- Super_Cool_Project_FINAL_FINAL
|--- Super_Cool_Project_Hyenas
|--- Super_Cool_Project_Hyenas_FancyAnalysis
|--- Super_Cool_Project_Hyenas_old
|--- Super_Cool_Project_Hyenas_old_old
|--- Super_Cool_Project_Hyenas_old_SuggestionsFromFriend
|--- Super_Cool_Project_Hyenas_SuggestionsFromFriend
|--- Super_Cool_Project_SuggestionsFromSuperVisor
|--- Super_Cool_Project_v2
|--- Super_Cool_Project_v3
|--- Super_Cool_Project_Songbird
```

In 3-4 years, which is the average lifespan of a research project, there will be more versions scattered over a million folders on various computing devices. Different versions will be saved in e-mails you have sent to others, and there will be no realistic way to quickly access an arbitrary version of your code.

Enter version control. Version control software basically automate the storage of your code's history and all its versions, also making the writing of collaborative code easier. Moreover, using version control is admitting that your code is a living breathing entity, and that you have slowly and arduously built it to be what it is. Version control is a way of showing your work, and demonstrating your good scientific practices.

Git is the industry and academic standard for version control. It is a very fast version control software with very powerful capabilities designed for version control and collaboration even for very very large projects like the Linux Kernel. With Git, you can have all versions of your project, and all its documented history, in one normal folder on your computer. You can choose to activate any version of your choice when you need it, and that is exactly what you will get in the form of a simple folder with a specific desired version of your code in it. All the Git magic happens behind the scenes.

2 Setting up

Depending on your operating system, you can install git on your system using one of the following methods.

2.1 Linux

If your system is based on Debian, you can use `sudo apt-get install git`. If your system is based on some other flavour of linux, you can use your typical installation method.

2.2 Mac

You can use `brew install git` to install git on Mac. That's what the internet says. I have never used a Mac.

2.3 Windows

Git says that you can install git for windows. Please do this, and make sure you have the software GitBash installed. GitBash is the command-line for running git on Windows, and is where you will run all your commands. It should automatically be installed when you install Git for Windows from the link above.

3 Getting help

If you need git-related help after the session on July 10, 2023, you can use the following resources.

- `man git`
- `man gittutorial`
- Chacon & Straub, *Pro Git* (Apress Publications, 2014) (freely available online as a PDF)

4 Preparing for the session

To prepare for the session, make sure you have a command-line working where the command

```
$ git --version
```

when typed in your shell (GitBash in case of Windows) gives you a version number and not an error. Also, to simulate the process of writing code at high speed, we will be making and sharing small text-files.[1] Please choose a multi-stanza poem of your choice, in any language, that will serve as an analog for code in our tutorial session.

Everything that follows from this point will be covered in the workshop.

---

5 Introducing yourself to Git

Run the following commands

[1] Since all code is, in the end, text.

```
$ git config --global user.name "Pranav 'Baba' Minasandra"
$ git config --global user.email "pminasandra@ab.mpg.de"
```

This is done because, when we eventually start putting up our code online, or sharing with collaborators, it is important to make sure that we know who made what edits. Make sure that you use your own name and e-mail in place of the placeholders.

Another important thing we will need to do is edit text, to tell git exactly what we have been up to. For this, let us tell git what kind of text editor we like to use.

For linux, type `git config --global core.editor <editor name>`, where the editor name can be `vim`, `nano`, `gedit`, or whatever else you use.

For Windows, the internet and the book Pro Git recommend the command

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad
   ++.exe' -multiInst -notabbar -nosession -noPlugin"
```

Of course, if you are using notepad++ with the above command, make sure that notepad++ is already installed on your system. (It ideally should be, but if it's not, just install it)

Note: In the past we have had some problems with the Windows editor. If you face this again, I can help out during the class. # Starting a git project

Projects version-controlled by git are called git repositories. Choose a folder, and navigate to it using the command `cd </path/to/folder>`. For the purposes of this workshop, we will use a new, empty folder for this. Once you have `cd`ed into your folder, enter the command

```
$ git init
```

to start git-based version control in this folder. Git should now ideally say something like

`Initialized empty Git repository in <your folder here>`.

6 Creating and committing your first few files

In this workshop, we will be using poetry as an alternative for writing code. Good code looks and feels exactly like poetry, and very bad poetry looks and feels exactly like code. Our goal here is to divide a poem's stanzas across several files, and use a single command to print the poem together. This command, shown below, is analogous to actually running your code. Use notepad, vim, or any editor of your choice to create two text files with names `file01.txt` and `file02.txt`. In each of these, add the first and second stanzas of your poem.

Since the poem is going to be torn up and its pieces put in all these different files, we need some quick way to display the poem in its entirety. The command `cat *.txt` will display the contents of all files in this folder in alphabetical order of file names, so that you can view the poem as one chunk

```
$ cat *.txt

She walks in beauty, like the night
Of cloudless climes and starry skies;
And all that's best of dark and bright
Meet in her aspect and her eyes;
```

Now type the command

```
$ git status
```

If all has gone according to plan, git will now say something like:

```
On branch master
```

```
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file01.txt
    file02.txt

nothing added to commit but untracked files present (use "git add" to
    track)
```

Read that message carefully, and see what it says. Git is telling us that you, the user, have introduced two files into the repository, but that Git has not yet been asked to take care of (i.e., version control) these files. Now, you need to add files to git to be tracked (like it just told us). One way to do this is to add them all individually:

```
$ git add <file1> <file2>
```

But this would be a pain if you had many many files. Sometimes during a good coding session, we might as well modify and create up to 10 files at once. A much easier way is to do this:

```
$ git add .
```

where `.` stands for 'the entire *current* directory'. Your files have been added to what is called the 'staging area', a temporary modification-tracking space. At this point, if we run

```
$ git status
```

again, we will see something like

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   file01.txt
    new file:   file02.txt
```

Now the new files have been recognised as changes for Git to remember as a new 'version' of the code. At this point, we can tell git to 'commit' all our changes like so:

```
$ git commit
```

Git should now prompt you to enter a message. A `git commit` can be thought of as "With this code, I have reached a point where I might want to share this progress with others, or a point I might want to come back to in the future." So in this sense a commit is a commitment from Git: you will always be able to restore or access any committed version of your code.

Now, as an exercise, modify one of your files (maybe add the name of the author and the title of the poem to the first file, file01.txt), and then add the modified file to staging (`git add`), and commit your changes (`git commit`). Look at the output of `git status` before and every step.

You can get Git to remove a file from tracking at any point, but keep in mind that that file will still be 'remembered' by Git in past commits. To remove a file from your repository, you can use

```
$ git rm <filename>
```

Use this to remove a file, and then add and commmit yet another file with some contents to your repository. By this point, you should have committed three times. Run the command

```
$ git log --graph
```

```
* commit 133f68541ad6febfeb8a215a70bb433fc6c6d8bf (HEAD -> master)
| Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
| Date:   Wed Jul 5 12:48:13 2023 +0200
|
|     Added title and Lord Byron's name
|
* commit 6b0716418dea9b44a56be3d7d433af4d63d1a064
  Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
  Date:   Wed Jul 5 12:47:09 2023 +0200

      First 4 lines added
```

This might seem like an unhelpful message with random garbled text in it, but I assure you, it's not. In class, we will discuss the meanings of the commit IDs (the jumbles of alphabets and numbers), the words 'HEAD', 'master', and 'graph'. We will see the full use of this stuff when we move on to:

7 Creating parallel universes (git branch)

Branching is useful when you want to *safely* try out an alternative approach to something. Imagine that you have the ability to split reality itself into two alternating universes, and you write different code in each. To create out first branch, let's use:

```
$ git branch <your-branch-name>
```

Note that `<your-branch-name>` must absolutely not contain spaces or special characters. Use _ if you have to.

Typing the following command will print out all the branches that exist

```
$ git branch
```

The output should look something like this:

```
  <your-branch-name>
* master
```

`master` is the name of the default branch in git, which was created when we first initialised this repository.[2] The `*` before `master` says that we are still on that branch. This is because while we created a separate branch, we still haven't moved to it. We move to the other branch by using

```
$ git switch <your-branch-name>
```

and running `git branch` again will show you that you have switched.

Now, choose a file and modify it. I want to modify file01.txt and add an underline after the title and author of the poem. Add another new file, `file03.txt` with the next stanza. Add and commit both these files (i.e., with the correct use of the commands `git add` and `git commit`. After this, carefully examine the poem with `cat *.txt`, before switching back to the master branch using

```
$ git switch master
```

and look at your poem again. You will notice that the files have gone back to what they used to be, and the additional file you created in your separate branch has vanished. Here, I will show you a few of my steps:

```
$ git branch gaudy_day
$ git branch
```

[2] While git and things like GitLab call the default branch 'master', github has recently started referring to it as 'main'. This is extremely annoying.

```
* gaudy_day
  master

# I added file03.txt here with the next stanza
# Also, I added an underline with '--' under the author's name
$ cat *.txt

SHE WALKS IN BEAUTY
    by Lord Byron
--------------------

She walks in beauty, like the night
Of cloudless climes and starry skies;
And all that‘s best of dark and bright
Meet in her aspect and her eyes;
Thus mellowed to that tender light
Which heaven to gaudy day denies.

$ git add .
$ git commit

$ git switch master

Switched to branch 'master'


$ cat *.txt

SHE WALKS IN BEAUTY
    by Lord Byron

She walks in beauty, like the night
Of cloudless climes and starry skies;
And all that‘s best of dark and bright
Meet in her aspect and her eyes;

# Since file03.txt was only added in a different branch,
# the changes from there don't appear in the branch 'master'.
```

8 Making universes collide (git merge)

The best way to write code is using branches. We try out different ideas in different branches, and each branch reaches some conclusion. If we are happy with how a branch turned out, we then want to bring those changes onto the branch `master`. Remember the file you modified in your new branch, (file01.txt in my case)? It hasn't been changed in `master`, because using our older analogy, we made the changes in a parallel universe. I will modify file01.txt, and add an underline below the author name here as well, but with the '.' character in place of '-'. Once again, add and commit this change. Now as it stands, you have two different versions of one file in two branches (file01.txt), and you also have an additional file in your branch that doesn't exist in `master` (file03.txt). All good so far? Good. It's time to make universes collide.

First, make sure you're on the branch `master` with the command `git branch` again.

Then, use the command

```
$ git merge <your-branch-name>
```

to bring the two universes together. You notice that git complains immediately:

```
$ git merge gaudy_day

Auto-merging file01.txt
CONFLICT (content): Merge conflict in file01.txt
Automatic merge failed; fix conflicts and then commit the result.
```

The first thing to notice is that your extra file (file03.txt) from your branch has been brought over to `master`, no questions asked. However, Git has no idea how to put together the two versions of file01.txt, which are said to be 'in conflict'. The file you modified is an issue, and Git makes it easy for you to solve this issue by highlighting all the lines where conflicts happen. If we now ask the poem to be viewed in its entirety, we get:

```
$ cat *.txt
SHE WALKS IN BEAUTY
    by Lord Byron
<<<<<<< HEAD
.....................
=======
--------------------
>>>>>>> gaudy_day

She walks in beauty, like the night
Of cloudless climes and starry skies;
And all that's best of dark and bright
Meet in her aspect and her eyes;
Thus mellowed to that tender light
Which heaven to gaudy day denies.
```

See the chaotic-looking blurb near the beginning? Git is telling us that `master`, the currently active branch has a line that looks like '.....', whereas the branch `gaudy_day` has one that looks like '——-'. Go ahead and open the `file01.txt`, and you will see that git has helpfully annotated the conflicting lines for you. As an exercise, modify the file by choosing one of the modifications, and remove all of Git's extra annotations. Then add and commit everything to master again. We can see everything we've done so far in graphical representation now using

```
$ git log --graph

*   commit 304935924b49b612c511bfb7daecac358478a55c (HEAD -> master)
|\  Merge: b272fda 092bf05
| | Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
| | Date:   Wed Jul 5 17:12:04 2023 +0200
| |
| |     Merge branch 'gaudy_day'
| |
| |     We chose to go with the '....' based underline
| |
| * commit 092bf05f5ddb8b11ade61463008c19d5f0e03dfc (gaudy_day)
| | Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
| | Date:   Wed Jul 5 12:54:52 2023 +0200
| |
| |     Added decoration to first file and then third stanza
| |
* | commit b272fdaab9c0100268ce27fc29198aec2737eb34
|/  Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
|   Date:   Wed Jul 5 17:03:36 2023 +0200
|
|       Added ...... to underline the poem title
```

```
|
* commit 133f68541ad6febfeb8a215a70bb433fc6c6d8bf
| Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
| Date:   Wed Jul 5 12:48:13 2023 +0200
|
|     Added title and Lord Byron's name
|
* commit 6b0716418dea9b44a56be3d7d433af4d63d1a064
  Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
  Date:   Wed Jul 5 12:47:09 2023 +0200

      First 4 lines added
```

Now that we have brought over all the changes from our extra branch, we don't need it any more. We can delete branches using

```
$ git branch -d <your-branch-name>
```

Git is safe, it only lets you delete branches after they have been merged back into their parent branch, Of course, sometimes we do crazy stuff in branches and don't want to merge them to our `master`. If you absolutely want to remove an experimental branch, you can be very very careful and use

```
git branch -D <crazy-experimental-branch> #but be careful
```

9 Time travel (undoing mistakes)

The biggest use of Git for most people is, of course, a way to reliably make mistakes and recover from them. What if you had a long coding day, and everything went well up to 15:00, but all your work from 15:00 - 19:00 was useless? There are many ways of 'going back in time' using Git. We will discuss mainly three of these.

All such methods rely on you periodically making commits in your code. Basically, all of Git is self-motivated. It isn't automatic the way Google version controls Google Docs. Git saves things when you tell it to save them. This ensures that your Git repository adheres to your way of working. Train yourself to add commits after each small milestone. Got your code to read in data? That's a commit. Added code to put the data in an appropriate format? That's a commit too. Commits are essentially snapshots of time that can be visited by your Git time machine.

9.1 Fixing recent errors (git restore)

Suppose, after your most-recent commit, you modified a particular file and messed up everything. You can call the command

```
$ git restore <modified-file>
```

to revert that file back to its state at the most recent commit. If you have made a lot of edits, and somehow everything broke. If you can't find out how to fix this stuff, you can discard all new changes and go back to the most recent commit by running

```
$ git restore . # but be VERY VERY careful
```

in your repository's central folder. This will however destroy all changes you made since your most recent commit.

9.2 Going back to an old commit (git branch)

Let's say you've been trying out a whole new approach, and you've created many new code files and in the end, have found out that the whole approach was incorrect, or wrongly implemented. Add `file04.txt` with the next stanza of your poem, but mess up this stanza somehow, and then stage and commit it. For good measure, also mess up things in `file01.txt` in another commit. Ensure that you have made two separate commits.

```
$ cat *.txt
DOES SHE WALKS IN BEAUTY??
    by Lord Byron
..................

She walks in beauty, like the night
Of cloudless limes and tarry skies;
And all that‘s best of dark and bright
Meet in her aspect and her eyes;
Thus mellowed to that tender light
Which heaven to gaudy day denies.

1 shade the more, 1 ray the less,
Had much impaired the nameless grace
Which waves in every raven dress,
Or sbrightlyly lightens over her damn face;
```

As you see, the title and first stanza from `file01` are messed up, and the last paragraph is unusable due to `file04`. Obviously, Lord Byron would not like this. Like you see in `git log --graph`, each commit is accompanied by a commit ID, more properly called a 'checksum'. Something like: `304935924b49b612c511bfb7daecac358478a55c`.

From `git log`, choose a previous commit that you want to go back to, when the poem was last still on track. We then create a different branch, a parallel universe which is basically this universe at a previous point in time.

`$ git branch <branch-name> 304935924b49b612c511bfb7daecac358478a55c`

If you are not very masochistic, you will not want to type out the entire commit ID. Git lets you get away with this by typing the first few characters as well:

`$ git branch <branch-name> 3049359` should do.

```
$ git branch old_works 304935924b
$ git switch old_works

Switched to branch 'old_works'

$ cat *.txt

SHE WALKS IN BEAUTY
    by Lord Byron
..................

She walks in beauty, like the night
Of cloudless climes and starry skies;
And all that‘s best of dark and bright
Meet in her aspect and her eyes;
Thus mellowed to that tender light
Which heaven to gaudy day denies.
```

and as you see, you have now accessed a working copy of your older poem/code again with not much

effort, and this is available to you as a branch.

9.3 Reverting a commit (git revert)

Let us first switch back to `master`, with `git switch master`. All the bad changes to the poem are still there, like you can check. However, we know exactly which commits introduced these terrible changes from `git log --graph`:

```
* commit 75d1c9eeffeb9a1754dfd85a98250d56d664cb1c (HEAD -> master)
| Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
| Date:   Thu Jul 6 14:33:09 2023 +0200
|
|     Messing up the title
|
* commit 844f363132c781c79c78110117c6103e1a234674
| Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
| Date:   Thu Jul 6 14:28:13 2023 +0200
|
|     Added 4th paragraph, but I am drunk
|
*   commit 304935924b49b612c511bfb7daecac358478a55c (old_works)
|\  Merge: b272fda 092bf05
| | Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
| | Date:   Wed Jul 5 17:12:04 2023 +0200
| |
| |     Merge branch 'gaudy_day'
| |
| |     We chose to go with the '....' based underline
[...]
```

Git gives us a way to 'revert' these commits one after the other. I can ask git to revert the previous two commits by saying

```
$ git revert 75d1c 844f3
```

Note that the commit IDs correspond to the 'bad' commits in the above git-log. git will now automatically add two new commits, which basically revert the state of the code to the way it was earlier. My `git log --graph` now starts with

```
* commit 34a4072e2333479f3e6c88d80867282b8a19a9c2 (HEAD -> master)
| Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
| Date:   Thu Jul 6 14:44:57 2023 +0200
|
|     Revert "Added 4th paragraph, but I am drunk"
|
|     This reverts commit 844f363132c781c79c78110117c6103e1a234674.
|
* commit fcfef3e2a3fc07714b2ec6ed8ab4d82cfe89ff3b
| Author: Pranav 'Baba' Minasandra <pminasandra@ab.mpg.de>
| Date:   Thu Jul 6 14:44:50 2023 +0200
|
|     Revert "Messing up the title"
|
|     This reverts commit 75d1c9eeffeb9a1754dfd85a98250d56d664cb1c.
```

and the poem is fixed as well. A permanent record is made of any mistake in the process of coding. This comes in handy more often than not, since sometimes we discard code without knowing its full potential. But, what this means is that any file committed to a git repository can *always* be accessed

in all of its past states. **Never** add sensitive information such as passwords, credentials, or unpublished data to your git repository. It should just be code, and not much else.

9.4 Other ways of going back

There are far more permanent ways of undoing changes than this, such as the infamous `git reset`. However, these methods make you prone to accidentally losing huge amounts of progress. Let us therefore not speak of such blasphemous things, and content ourselves with the `git branch` based method above.

As a quick exercise, modify one of your files, and then restore it to its state during the previous commit. After this, make a branch from your very first commit, switch to it, and ponder about how much git you've learned over the last hours.