Introduction to Git and Github

Google

Contents

[Module 1 – Introduction to Version Control 3](#_Toc54204727)

[Section 1 – Course Introduction 3](#_Toc54204728)

[Section 2 – Before Version Control 3](#_Toc54204729)

[Keeping Historical Copies 3](#_Toc54204730)

[Diffing Files 3](#_Toc54204731)

[Applying Changes 5](#_Toc54204732)

[Practical application of diff and patch 6](#_Toc54204733)

[Section 2 - Version Controls Systems 6](#_Toc54204734)

[What is version control? 6](#_Toc54204735)

[Version Control and Automation 7](#_Toc54204736)

[What is Git? 8](#_Toc54204737)

[Installing Git 8](#_Toc54204738)

[Section 3 – Using git 9](#_Toc54204739)

[First steps with git 9](#_Toc54204740)

[Tracking Files 11](#_Toc54204741)

[The Basic Git Workflow 13](#_Toc54204742)

[Anatomy of a Commit Message 16](#_Toc54204743)

[Module 2 – Using Git locally 21](#_Toc54204744)

[Section 1 – Advanced Git Interaction 21](#_Toc54204745)

[Skipping the staging Area 21](#_Toc54204746)

[Getting More information about our changes 21](#_Toc54204747)

[Deleting and renaming files 21](#_Toc54204748)

[Section 2 – Undoing Things 23](#_Toc54204749)

[Section 3 – Branching and Merging 23](#_Toc54204750)

[What is a branch? 23](#_Toc54204751)

[Creating new Branches 24](#_Toc54204752)

[Working with branches 25](#_Toc54204753)

[Merging 27](#_Toc54204754)

[Merge Conflicts 28](#_Toc54204755)

[Module 3 – Working with remotes 29](#_Toc54204756)

[What is Github? 30](#_Toc54204757)

[Basic interactions with Github 30](#_Toc54204758)

[Section 2 - Using a remote repository 32](#_Toc54204759)

# Module 1 – Introduction to Version Control

## Section 1 – Course Introduction

Version control systema (VCS)

## Section 2 – Before Version Control

### Keeping Historical Copies

This is the primitive way of version control

“First, you need to remember to make the copy. Second, you usually make a copy of the whole thing, even if you're only changing one small part. And third, even if you're emailing your changes to your colleagues, it might be hard to figure out at the end who did what, and more importantly, why they did it.”

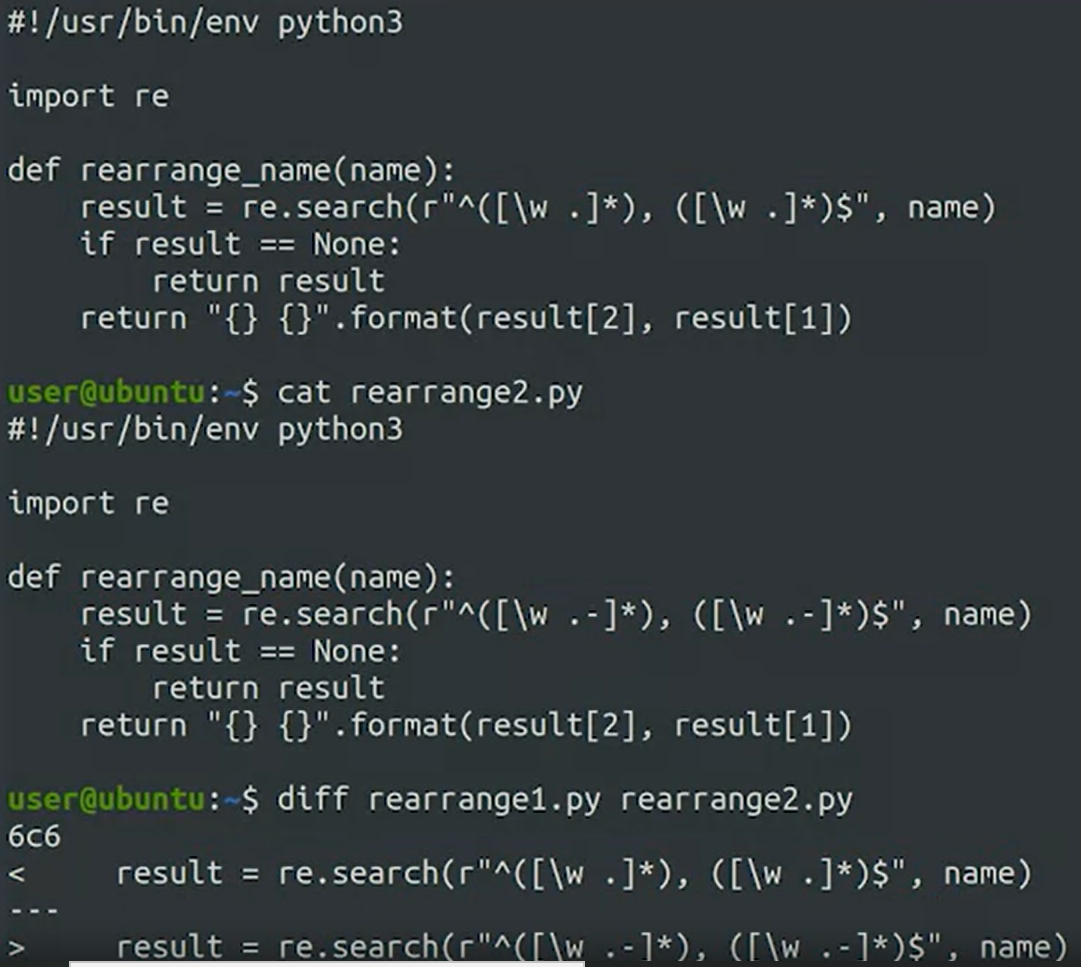
A version control system allows us to keep track of when and who did what changes to our files. Those can be code, configuration, images, or whatever else we need to track.

### Diffing Files

“We can use the diff command line tool to take two files or even 2 directories, and show the differences between them in a few formats.”

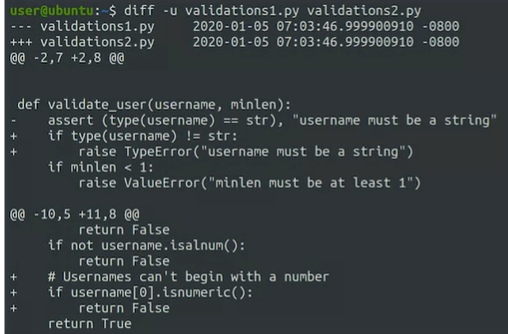
*At this point he starts assuming that we took the previous course in de specialization by showing some command line stuff. It might be better now to take that course at this point*

“When we call the diff command, we get only the lines that are different between two files. It's much easier to find the difference when we just have two lines. See the symbols at the beginning of each of those lines? The less than symbol tells us that the first line was removed from the first file, and the greater than symbol tells us that the second line was added to the second file. In other words, the old line got replaced by the new one. In this example, we had one line that was replaced with a new one.”:



*Another thing we can type to check changes more intuitively*

“To understand this better we can use the **dash u flag** to tell diff to show the differences in another format. Let's check that out. This unified format is pretty different from the one that we saw before. It shows the change lines together with some context, using the minus sign to mark lines that were removed, and the plus sign to mark lines that were added.”



“There are a lot of tools out there to compare files. Diff is the most popular one, but not the only one available. For example, wdiff highlights the words that have changed in a file instead of working line by line like diff does. To help us even more, there are bunch of graphical tools that display files side by side and highlight the differences by using color. Some examples of this include: meld, KDiff3, or vimdiff.”

### Applying Changes

*This is in reference to telling/showing someone what is wrong with their code*

“To make the change clear, you could send them a diff with the change so that they can see what the modified code looks like. To do this, we typically use a command line like diff-u old\_file new\_ file > change.diff. As a reminder, the greater than sign redirects the output of the diff command to a file. So with this command, we're generating a file called change.diff with the contents of diff-u command. By using the -u flag, we include more context which helps the person reading the file understand what's going on with the change. The generated file is usually referred to as a diff file or sometimes a patch file. It includes all the changes between the old file and the new one, plus the additional context needed to understand the changes and to apply those changes back to the original file.”



“Now, say you're the one receiving a diff file with a change and you want to apply it to a script you wrote. You could read the diff file you receive carefully and then manually go through the file that needs to be changed, and apply the modifications. But it sounds like a lot of manual work that could be automated, don't you think? Well, it sure is. There's a **command called patch** to do exactly this. Patch takes a file generated by diff and applies the changes to the original file.”

“So we have the diff file and we want to apply it to our script. How do we do that? We'll use the patch command. We'll pass the name of the file that we want to patch in this case, cpu\_usage.py, as the first parameter to the command and then we'll provide the diff file through standard input. Do you remember how to do that? We will use the less than symbol to redirect the contents of the file to standard input. Let's check this out. So we told patch to apply the changes that come from cpu\_usage.diff to our cpu\_usage.py file. We get one single line that says the file was patched, which means that we've successfully applied the changes.”:



“While diff is the command that generates the difference between two files, patch is the command that applies those differences to the original file.”

“You might be wondering, why go through all this trouble diffing, and patching, and not just send the whole file instead? There are a few reasons for this. The main reason is that the original code could have changed. In our example, it's possible that the code our colleague was using to prepare the fix wasn't the latest version. By using a diff instead of the whole file, we can clearly see what they changed, no matter which version they were using. The patch command can detect that there were changes made to the file and will do its best to apply the diff anyways. It won't always succeed but in many cases it will. Another reason is structure. In this case we're patching a single small file. But sometimes, you might be modifying a bunch of large files inside of a huge project. Say you are changing four files in a project tree that contain 100 different files, arranged in different directories according to what they do. If you were to send the whole files, you'd need to specify where those files were supposed to be placed. As we called out, we can diff whole directory structures and in that case the diff file can specify where each change file should be without us having to do any manual juggling.”

### Practical application of diff and patch

*Essentially just went through a practical application of eventually doing this:*



## Section 2 - Version Controls Systems

### What is version control?

“We've seen up till now, how we can use existing tools to extract differences between versions of files and apply those changes back to the original files. Those tools are really useful. But most of the time, we won't be using them directly. Instead, we'll use them through a Version Control System, or VCS. A Version Control System keeps track of the changes that we make to our files. By using a VCS, we can know when the changes were made and who made them. It also lets us easily revert a change if it turned out not to be a good idea. It makes collaboration easier by allowing us to merge changes from lots of different sources. At first-look, a Version Control System can seem like a complicated, possibly intimidating tool. But if you look closer, you'll see that it's really just a system that stores files. However, unlike a regular file server which only saves the most recent version of a file, a VCS keeps track of all the different versions that we create as we save our changes. There are many different version control systems, each with their own implementation and with their own advantages and disadvantages. But, no matter how the VCS is implemented internally, they always access the history of our files. Let us retrieve past versions of the file or directory and see who changed which files, how each file was changed and when the file was changed.”

“On top of this, we can make edits to multiple files and treat that collection of edits as a single change which is commonly known as a, **commit**.”

“A VCS even provides a mechanism to allow the author of a commit to record why the change was made, including what bugs, tickets or issues were fixed by the change. This information can be a lifesaver when trying to understand a complex series of changes, or to debug some obscure issue.”

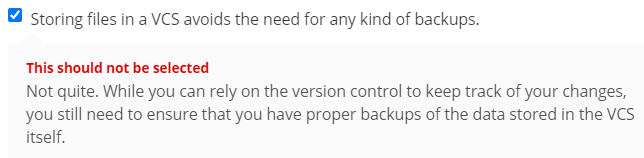
“In any organization that produces software, a VCS is a key part of managing the code. Files are usually organized in repositories which contains separate software projects or just group all related code. If there's a lot of people involved in developing software, some developers may have access to only some of the repositories.”

“A Version Control System can be used to store much more than just code. We can use it to store configuration files, documentation, data files, or any other content that we may need to track. Because of the way tools like diff and patch work, a VCS is especially useful when tracking text files, which can be compared with diff and modified with patch. We can also store images, videos or any other complex file formats in a VCS, but, it won't be easy to check the differences between versions when comparing these file formats. It might not be possible to automatically merge changes made to older versions of a file.”

### Version Control and Automation

“A VCS can be invaluable, even in a one-person IT department. A VCS stores your code and configuration. It also stores the history of that code and configuration. A version control system can function a lot like a time machine, giving you insights into the decisions of the past. Whenever you write a commit message, after making a change, it's as if the current version of yourself is explaining your decisions to a future you or others who might work on the same scripts and configurations in the future. This can help you avoid finding yourself staring at a piece of code that you or someone else wrote three months ago and puzzling over how it works or even why it exists. With a VCS, you can view, track and select snapshots from the history of your project. So nothing you do is lost, and since we can use a VCS to store both code and configuration files, we can make the overall IT systems more scalable and reliable.”

*This is important for me to remember*



### What is Git?

“Unlike some version control systems that are centralized around a single server, Git has a distributed architecture. This means that every person contributing to a repository has full copy of the repository on their own development machines.

Collaborators can share and pull in changes that others have made as they need. And because the repositories are all local to the computer being used to create the files, most operations can be done really fast. If you want to collaborate with others, it usually makes sense to set up a repository on a server to act as a kind of hub for everyone to interact with. But Git doesn't rely on any kind of centralized server to provide control organizations to its workflow. Git can work as a standalone program as a server and as a client. This means that you can use Git on a single machine without even having a network connection. Or you can use it as a server on a machine where you want to host your repository. And then you can use Git as a client to access the repository from another machine or even the same one.

Git clients can communicate with Git servers over the network using HTTP, SSH or Git's own special protocol.”

“When looking for information online you might notice that the official Git website is called git-scm.com. And wonder what's the SCM at the end for? It's actually another acronym similar to VCS. It stands for Source Control Management. While both terms mean the same, we generally prefer VCS, because as we call that already, these systems can actually be used to store much more than just source code. In this course we chose Git for its popularity, multi platform support and robust set of features. As with most things in the IT world, though, there are plenty of other tools that can be used to accomplish the same task. There are other VCS programs like Subversion or Mercurial.”

### Installing Git

“The first step is to check whether you already have it installed. You can do this by running git --version.”

“One interesting thing about the Windows installation is that it comes preloaded with an environment called MinGW64. This environment lets us operate on Windows with the same commands and tools available on Linux. So you can practice some Linux command line tools on your Windows machine. After installing Git on your Windows machine, you'll be able to use Git from the Linux command line. If you selected the default option for the path environment question, you'll be able to also run it from the PowerShell command line.”

## Section 3 – Using git

### First steps with git

“Remember when we said that a VCS tracks who made which changes, for this to work, we need to tell Git who we are. We can do this by using the Git config command and then setting the values of user.email and user.name to our email and our name like this.”



“We use the dash dash global flag to state that we want to set this value for all git repositories that we'd use. We could also set different values for different repositories.

“There are two ways to start working with a git repository. We can create one from scratch using the git init command or we can use the git clone command to make a copy of a repository that already exists somewhere else.”

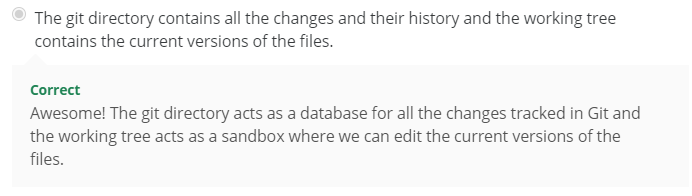


So when we run git init we initialize an empty git repository in the current directory. The message that we get mentions a directory called. git. We can check that this directory exist using the ls-la command which lists files that start with a dot. We can also use the ls-l.git command to look inside of it and see the many different things it contains. This is called a Git directory. You can think of it as a database for your Git project that stores the changes and the change history. We can see it contains a bunch of different files and directories. We won't touch any of these files directly, we'll always interact with them through Git commands. So whenever you clone a repository, this git directory is copied to your computer. Whenever you run git init to create a new repository like we just did, a new git directory is initialized.”

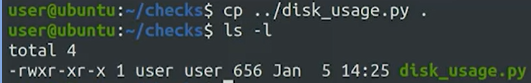


“Whenever you run git init to create a new repository like we just did, a new git directory is initialized. The area outside the git directory is the working tree. The working tree is the current version of your project. You can think of it like a workbench or a sandbox where you perform all the modification you want to your file. This working tree will contain all the files that are currently tracked by Git and any new files that we haven't yet added to the list of track files.”





“Right now our working tree is empty. Let's change that by copying the “diskusage.py” file that we saw in an earlier video into our current directory. We now have file and a working tree but it's currently untracked by Git.”

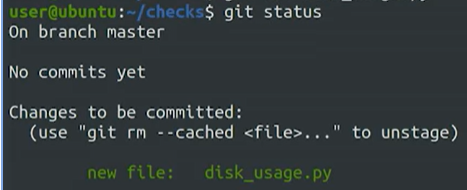


“We now have file and a working tree but it's currently untracked by Git. To make Git track our file, we'll add it to the project using the git add command passing the file that we want as a parameter. With that, we've added our file to the staging area. “



“The staging area which is also known as the index is a file maintained by Git that contains all of the information about what files and changes are going to go into your next command.”

“We can use the git status command to get some information about the current working tree and pending changes.”



“We see that our new file is marked to be committed, this means that our change is currently in the staging area. To get it committed into the.git directory, we run the git commit command.”



(This brings up Nano for us to write a description of what we did)

### Tracking Files

“In our last video, we mentioned that any Git project will consist of three sections. The **Git directory, the working tree, and the staging area**. The Git directory contains the history of all the files and changes. The working tree contains the current state of the project, including any changes that we've made. And the staging area contains the changes that have been marked to be included in the next commit.”

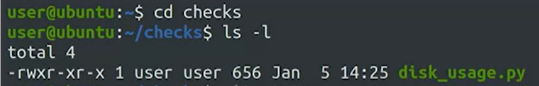
“Each time you make a commit, Git records a new snapshot of the state of your project at that moment. It's a picture of exactly how all these files looked at a certain moment in time. Combined, these snapshots make up the history of your project, and it's information that gets stored in the Git directory.”

“Now, let's dive into the details of how we track changes to our files. When we operate with Git, our files can be either **tracked or untracked**. Tracked files are part of the snapshots, while untracked files aren't a part of snapshots yet. This is the usual case for new files.”

“Each track file can be in one of three main states, **modified, staged or committed**.

If a file is in the modified state, it means that we've made changes to it that we haven't committed yet. The changes could be adding, modifying or deleting the contents of the file. Git notices anytime we modify our files. But won't store any changes until we add them to the staging area. So, the next step is to stage those changes. When we do this, our modified files become stage files. In other words, the changes to those files are ready to be committed to the project. All files that are staged will be part of the next snapshot we take. And finally, when a file gets committed, the changes made to it are safely stored in a snapshot in the Git directory. This means that typically a file tracked by Git, will first be modified when we change it in any way. Then it becomes staged when we mark those changes for tracking. And finally it will get committed when we store those changes in the VCS.”

“Let's see this in action in our example Git repo. First, let's check the contents of the current working tree using ls-l. “



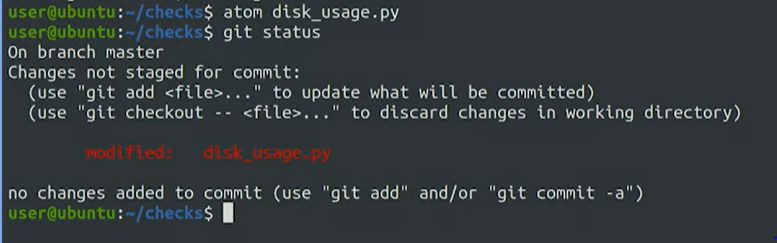
“And then the current status of our files using the Git status command. When we run Git status, Git tells us a bunch of things, including that we're on the master branch. We'll learn about branches later in the course. For now, notice how it says that there's nothing to commit and that the working tree is clean.”



Now an example of a modification, let’s say we have an existing file named disk\_usage.py.

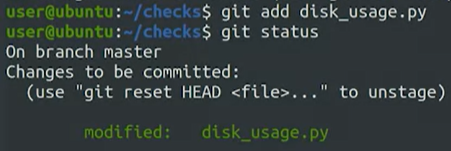
We type the commando “atom disk\_usage.py” to open the file in Atom and to make some modifications.

Then on the command line we use **git status** to get some information, and notice the text in red that will say that the file was modified

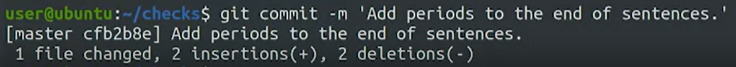


*Not on how to change that red modified message:*

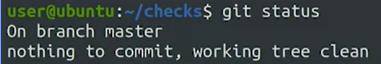
“Let's change [*the red modified message*] by running the **Git add command**, passing the disk usage py file as a parameter. When we call Git add, we're telling Git that we want to add the current changes in that file to the list of changes to be committed. This means that our file is currently part of the staging area, and it will be committed once we run the next Git command, **Git commit**.



In this case, instead of opening up an editor, let's pass the commit message using the dash m flag, stating that we added periods at the end of the sentences.So, we've now committed our stage changes. This creates a new snapshot in the Git directory. The command shows us some stats for the change made.



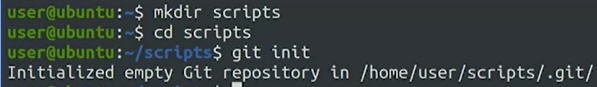
*When we check the status again, it will say that no changes are available to commit since we just finished the process*



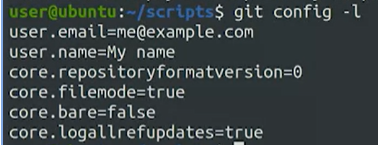
### The Basic Git Workflow

“We saw that each repository will have a Git directory, a working tree, and a staging area. And we called out that files can be in three different states, modified, staged, and committed”

“First, all the files we want to manage with Git must be a part of a Git repository. We initialize a new repository by running the **git init command** in any file system directory. For example, let's use the **mkdir command** to create a directory called scripts, and then change into it and initialize an empty Git repository init. Our shiny new Git repository can now be used to track changes to files inside of it.”

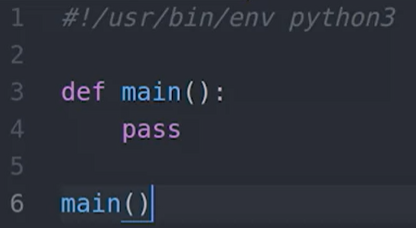


“But before jumping into that, let's check out our current configuration by using the git config -l command.”



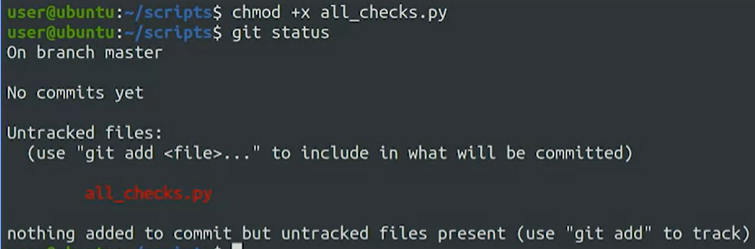
“For now, pay special attention to the user.email and the user.name lines, which we touched on briefly in an earlier video. This information will appear in public commit logs if you use a shared repository. For privacy reasons, you might want to use different identities when dealing with your private work and when submitting code to public repositories.”

“Okay, our repo is ready to work, but it's currently empty. Let's create a file in it, we'll start with a basic skeleton for a Python script, which will help us demonstrate the Git workflow. As with any Python script, we'll start with the shebang line. For now, we'll add an empty main function, which we'll fill in later. And at the end, we'll just call this main function.”



“This is a script that we'll want to execute, so let's make it executable. And then let's check the status of our repo using git status command.”

*To make it excecutable, we need the “chmod +x…” code. This was introduced in Course 2*

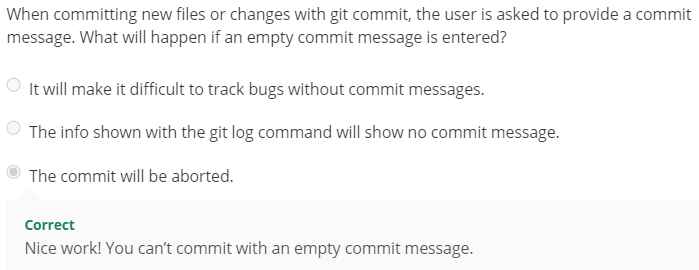


To make git track our file “we need to call the git add command. This command will immediately move a new file from untracked to stage status. And as we'll see later, it will also change a file in the modified state to staged state. Remember that when a file is staged, it means it's been added to the staging area and it's ready to be committed to the Git repository.



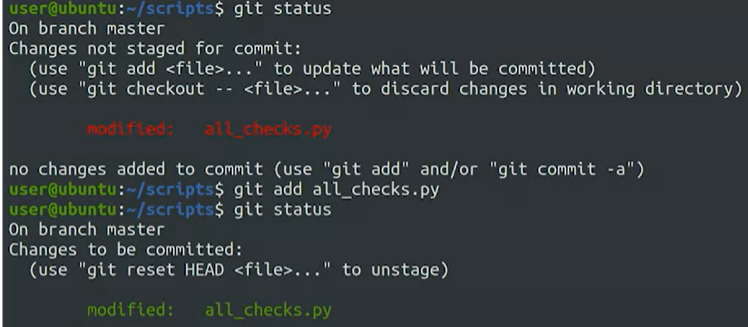
To initiate a commit of staged files, we issue the git commit command. When we do this, Git will only commit the changes that have been added to the staging area, untracked files or modified files that weren't staged will be ignored.

“Calling git commit with no parameters will launch a text editor, this will open whatever has been set as your default editor.”

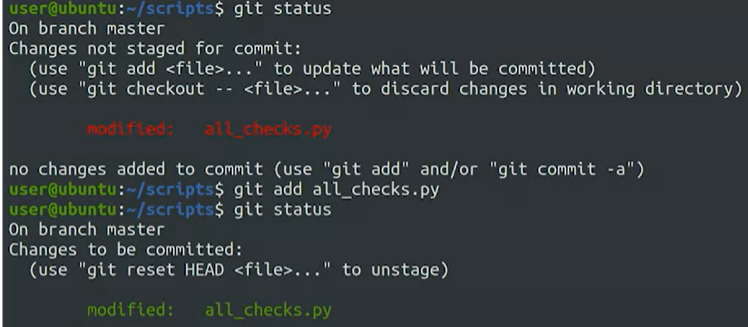


“Okay, that's how we add new files, but usually we'll modify existing ones. So let's add a bit more content to our script to see that in action.”

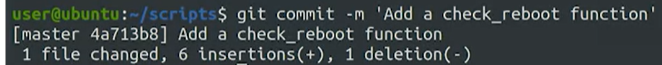
“We've added a function to our file. Let's check the current status using git status again. Our file's modified, but not staged.”



“To stage our changes, we need to call git add once again. Okay, our changes our now staged.”



“We have to call git commit to store those changes to the Git directory. This time, we'll use the other way of setting the commit message. We'll call git commit -m, and then pass the commit message that we want to use. So in this case, we'll say that we've added the check\_reboot function.”



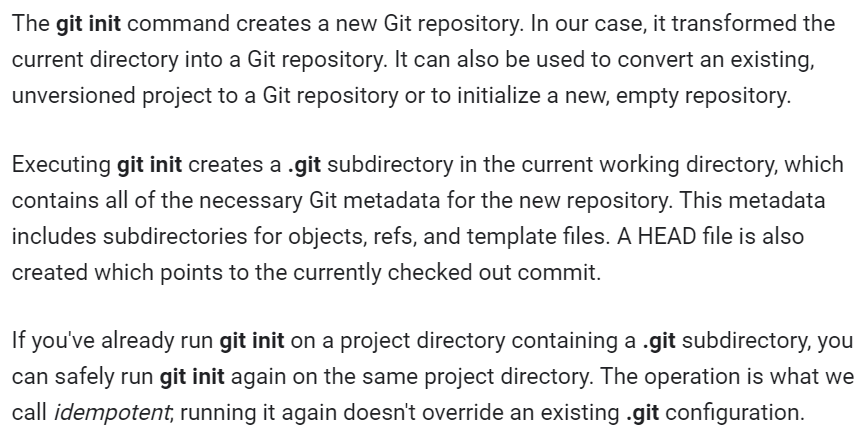
“we've demonstrated the basic Git workflow. We make changes to our files, stage them with git add, and commit them with git commit.”

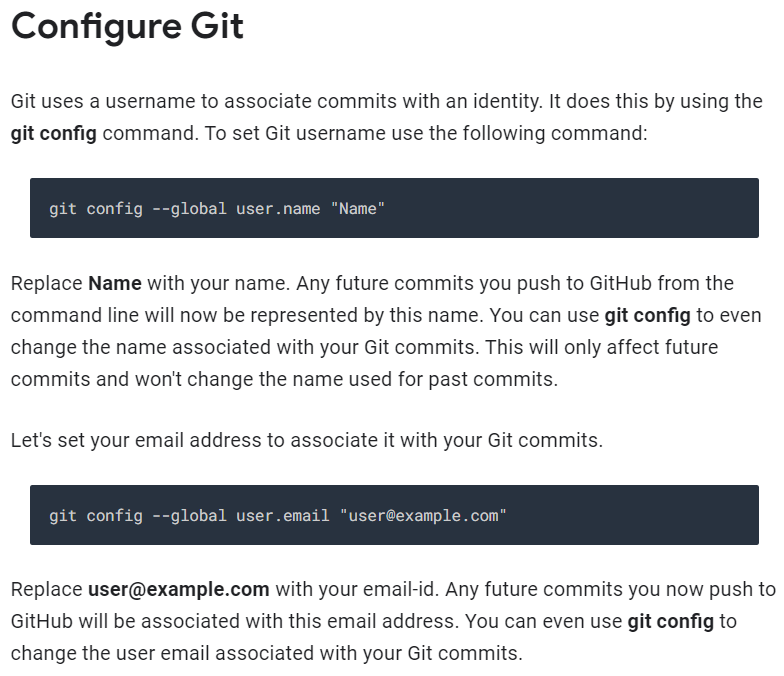
### Anatomy of a Commit Message

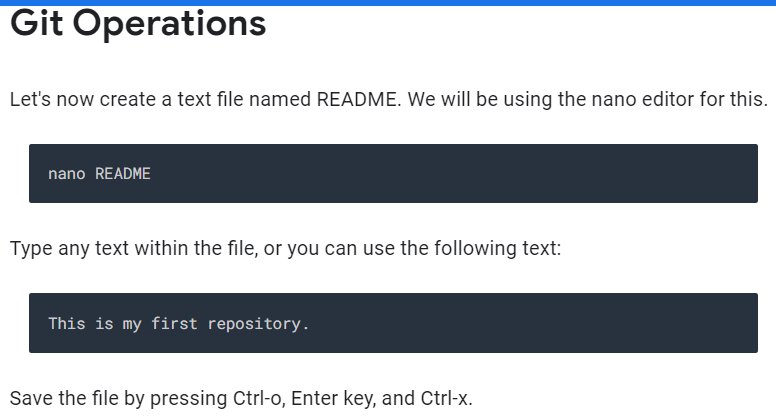
“A commit message is generally broken up into a few sections. The first line is a short summary of the commit followed by a blank line. This is followed by a full description of the changes which details why they're necessary and anything that might be especially interesting about them or difficult to understand. When you run the git commit command, Git will open up a text editor of your choice so you can write your commit message.”

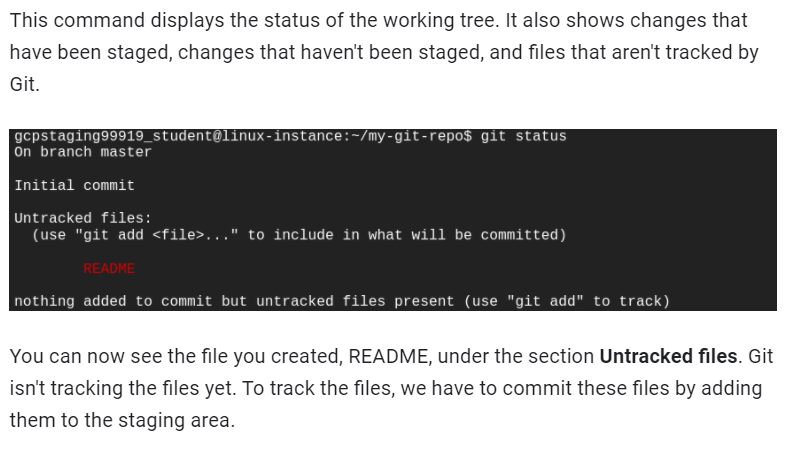
**The Assignment**

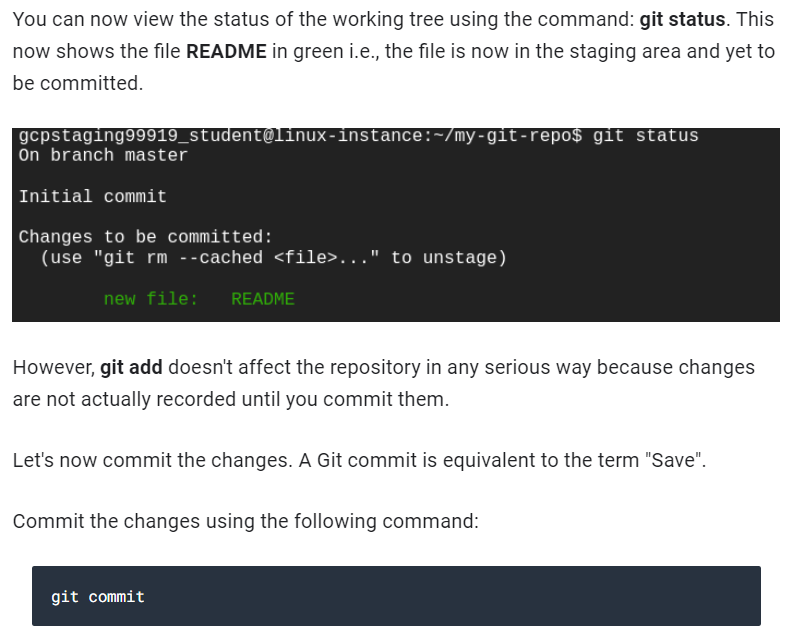


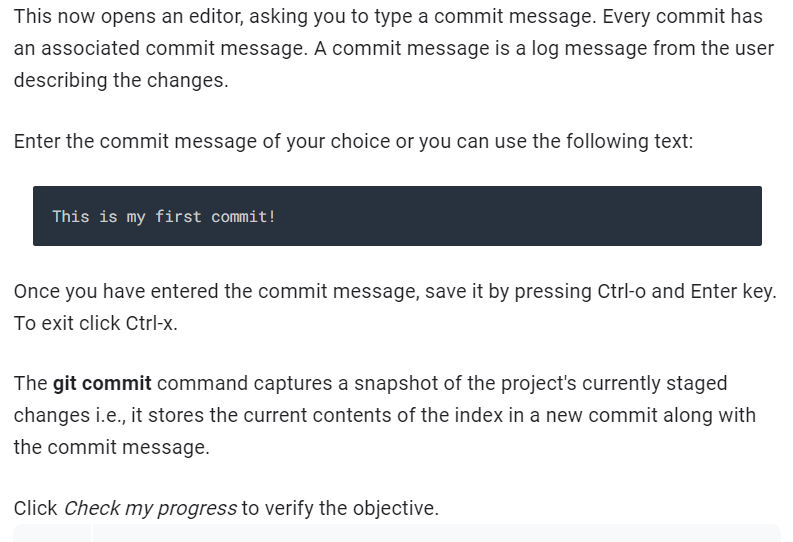












# Module 2 – Using Git locally

Learning Objectives

* Have a advanced understanding of Git
* Skip the staging area to delete and move files within Git
* Amend and roll back commits
* Understand the concept of branching and merging
* Create new branches and use merging to combine branched data
* Manage and handle merge conflicts

## Section 1 – Advanced Git Interaction

### Skipping the staging Area

“The separate step between staging and committing allows us to stage several changes in one commit. But if we already know that the current changes are the ones that we want to commit, we can skip the staging step and go directly to the commit. No dress rehearsals. We do this by using the **dash a-flag** to the git commit command. This flag automatically stages every file that's tracked and modified before doing the commit letting it skip the git add step. At first, you might think that git commit dash a is just a shortcut for git add followed by git commit but that's not exactly true. Git commit -a doesn't work on new files because those are untracked. Instead, git commit -a is a shortcut to stage any changes to tracked files and commit them in one step. If the modified file has never been committed to the repo, we'll still need to use git add to track it first.”

What is **head** in the git log context? “Git uses the head alias to represent the currently checked out snapshot of your project. This lets you know what the contents of your working directory should be. In this case, the current snapshot is the latest commit in the project. We'll soon learn about branches. In that case, head can be a commit in a different branch of the project. We can even use git to go back in time and have head representing old commit from before the latest changes were applied. In all cases, head is used to indicate what the currently checked out snapshot is. This is how git marks your place in the project. Think about it as a bookmark that you can use to keep track of where you are. Even if you have multiple books to read, the bookmark allows you to pick up right where you left off. When you run git commands like diff, branch, or status, git will use the head bookmark as a basis for whatever operation it's performing. We'll see Head used when we learn how to undo things and perform rollbacks. We'll talk more about branches in later videos. As a shortcut, it's generally easy to think of head as a pointer to the current branch, although it can be more powerful than that.”

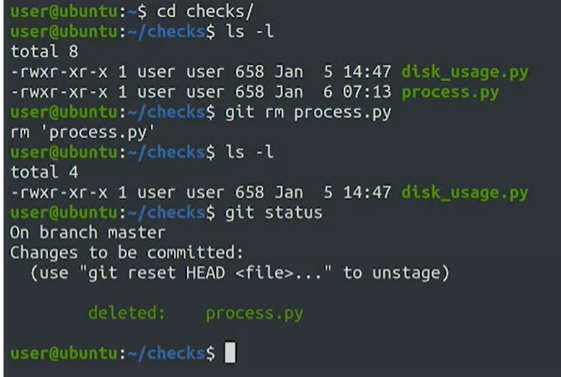
### Getting More information about our changes

*Skipped for now*

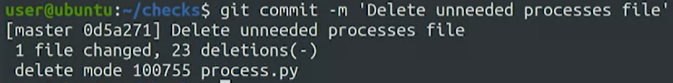
### Deleting and renaming files

“You can remove files from your repository with the **git rm command**, which will stop the file from being tracked by git and remove it from the git directory. File removals go through the same general workflow that we've seen. So you'll need to write a commit message as to why you've deleted them.”

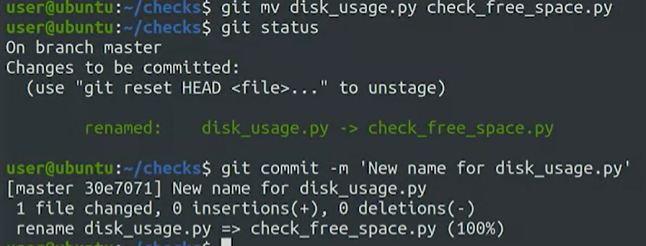
What we are doing below: “Well first look the contents of the directory with ls, then delete the file with git rm, then check the contents with ls again, and finally check the status with git status.”



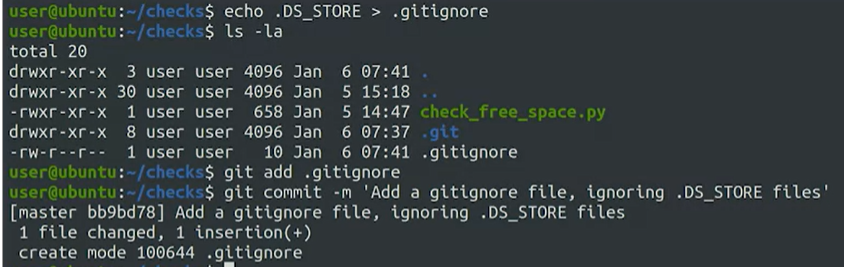
Then we have to commit the file deletion



“You can use the **git mv command** to rename files in the repository. Let's rename our existing script to check\_free\_space.py and check what git status has to say about that.”



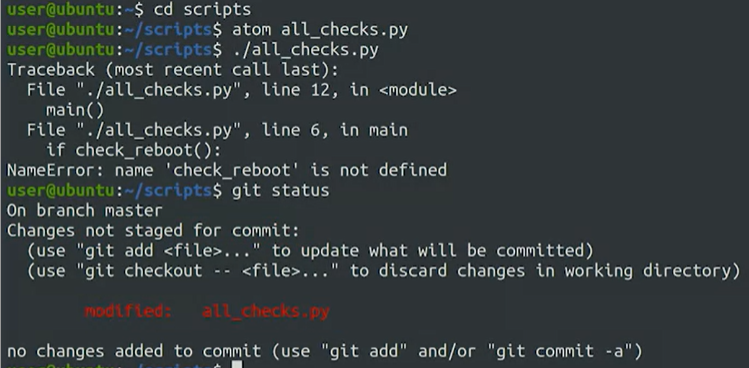
“If there are files that get automatically generated by our scripts, or our operating system generates artifacts that we don't want in our repo, we'll want to ignore them so that they don't add noise to the output of git status. To do this, we can use **the gitignore file**…Remember that the dot prefix in a Unix-like file system indicates that the file or directory is hidden and won't show up when you do the normal directory listing. That's why we have to use ls-la to see all files.”





## Section 2 – Undoing Things

“You might find yourself in a situation where you've made a bunch of changes to a file but decide that you don't want to keep them. You can change a file back to its earlier committed state by using the git checkout command followed by the name of the file you want to revert.”



## Section 3 – Branching and Merging

### What is a branch?

“In Git, a branch at the most basic level is just a pointer to a particular commit. But more importantly, it represents an independent line of development in a project. Of which the commit it points to is the latest link in a chain of developing history. The default branch that Git creates for you when a new repository initialized is called master. All of our examples and development have taken place on this branch so far. The master branch is commonly used to represent the known good state of a project. When you want to develop a feature or try something new in your project, you can create a separate branch to do your work without worrying about messing up this current working state.”

“You can think of a Git project as an assignment your teacher gives you in a class. You do all your work on the assignment in a set of notebooks, each notebook representing a different branch. You use some notebooks to jot down rough drafts in experiments, but you keep one notebook the master branch, in a tidy state and you copy the polish versions of these drafts into it. No doodling in the master note book, please.”

“Branches make it really easy to experiment with new ideas or strategies and projects. When you want to add a feature or fix something, you can create a new branch and do your development there. You can merge back into the master branch, when you've got something you like, or discard your changes without negative impact if they don't work out. In Git, branches are used all the time, as a part of the normal development workflow. As an example, think back to the problematic commit we fixed in an earlier video. We added a call to the disk full function, but forgot to actually define the function. So we had to roll it back because our users we're seeing errors. Knowing what we know now, we could have done that work on a separate branch, maybe called something like add disk full. In that case, we could have iterated on our code there until it was working correctly, without it effecting the master branch. Only after the code is ready to be deployed, we would merge those changes back into the master branch.”

### Creating new Branches

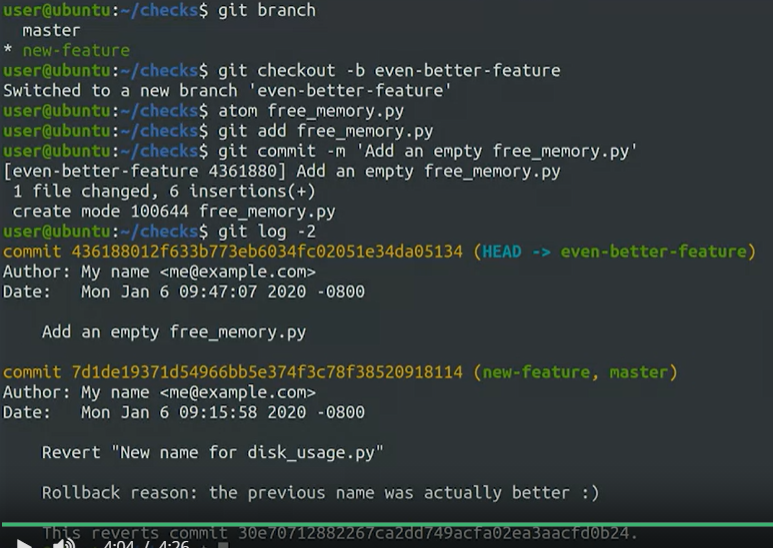
“We can use the git branch command to list, create, delete, and manipulate branches. Running git branch by itself will show you a list of all the branches in your repository.”

“now we want to switch to a new branch. To do that, we'll need to use the **git checkout command**. We saw earlier how we can use git checkout to restore a modified file back to the latest commit. Checking out branches is similar in that, the working tree is updated to match the selected branch including both the files and the git history. If this seems a bit confusing at first, you're not alone. I also found it hard to wrap my head around it first.”

“It might help to remember that we use git checkout to check out the latest snapshot for both files and for branches. All right. Let's switch to our new feature branch by calling git checkout new feature, and then listing our branches once again.”

“Creating a branch and switching to it immediately is a pretty common task. So common that git gives us a useful shortcut to create a new branch and to switch to it in a single command. We can use the **git checkout -b** new branch to do this.”

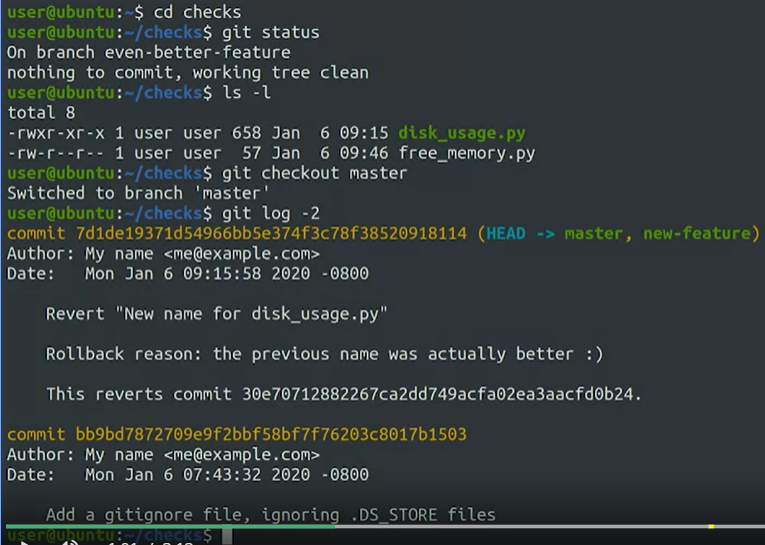
“We see the last two commits in this branch. Notice how next to the latest commit ID, git shows that this is where head is pointing to and that the branch is called “even better feature”. Next to the previous commit, git shows that both the master and the new feature branches are pointing to that snapshot of the project. In this way, we can see that the even better feature branch is ahead of the master branch.”

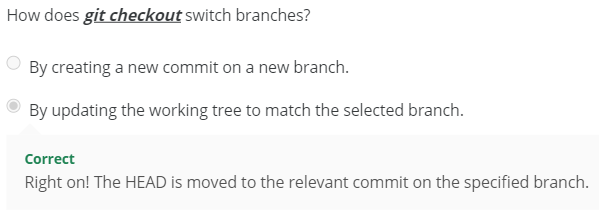


### Working with branches

“Let's check out the current status of our repo by calling git status and ls dash l. So we see that we're on a clean working tree in the even better feature branch, and that a new free memory py file is in our working tree. Let's now change back to the master branch using **git checkout master** and then lists the latest two commits there. When we switch to a different branch using git checkout, under the hood, git changes where head is pointing. Thanks to this checkout, head went from pointing to the latest commit in the even better feature branch to the most recent commit of the master branch. The commit from even better feature doesn't show up at all, and the latest snapshot is the second entry we've seen before. Remember that when we switch branches, git will also change files in our working directory or working tree to whatever snapshot head is currently pointing at.”

?? what is “git log -2”????





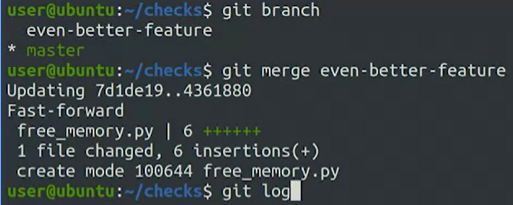
“So what if we want to delete a branch that we don't need anymore? We can do that by using **git branch dash d**. Let's first list the current branches in our repo and then get rid of the new feature branch by calling git branch dash d new-feature…If there are changes in the branch we want to delete that haven't been merged back into the master branch, git will let us know with an error.”

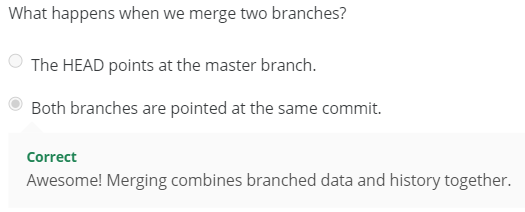


### Merging

“A typical workflow for managing branches in Git, is to create a separate branch for developing any new features or changes. Once the new feature's in good shape, we merge the separate branch back into the main trunk of code. Merging is the term that Git uses for combining branch data and history together. We'll use the git merge command, which lets us take the independent snapshots and history of one Git branch, and tangle them into another.”

“First, will check that we're in master branch, and then we'll call git merge even-better-feature to merge the even-better-feature branch into the master branch. Now we've brought the master branch up to speed, which we can see by looking at the git log.”





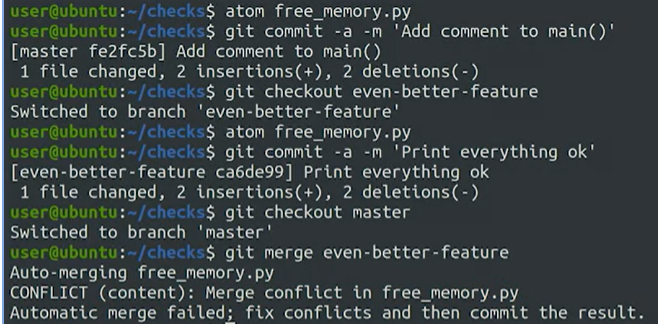
*Technical stuff:*

“Git uses two different algorithms to perform a merge, **fast-forward** and **three-way merge**. The merge we just performed is an example of a fast-forward merge. This kind of merge occurs when all the commits in the checked out branch are also in the branch that's being merged. If this is the case, we can say that the commit history of both branches doesn't diverge. In these cases, all Git has to do is update the pointers of the branches to the same commit, and no actual merging needs to take place. On the other hand, a three-way merge is performed when the history of the merging branches has diverged in some way, and there isn't a nice linear path to combine them via fast-forwarding. This happens when a commit is made on one branch after the point when both branches split.

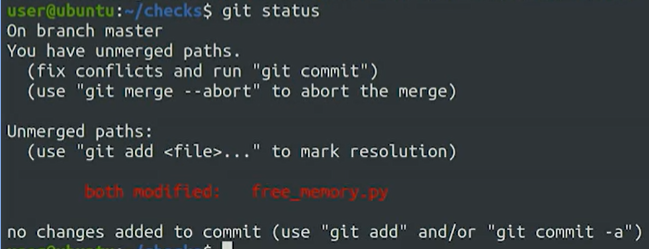
In our case, this could have happened if we made a commit on the master branch after creating the other branches. When this occurs, Git will tie the branch histories together with a new commit. And merge the snapshots at the two branch tips with the most recent common ancestor, the commit before the divergence. To do this successfully, Git tries to figure out how to combine both snapshots. If the changes were made in different files, or in different parts of the same file, Git will take both changes and put them together in the result. If instead the changes are made on the same part of the same file, Git won't know how to merge those changes, and the attempt will result in a **merge conflict**.”

### Merge Conflicts

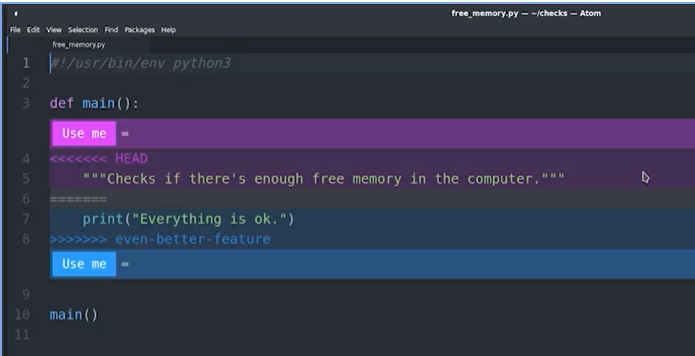
“let's edit the free\_memory.py file in the master branch and replace the pass statement with a comment about what the main function should do. Cool, we made the change so let's save it and commit it back to our master branch. Next, Let's check out the even-better-feature branch and make a change in the same place. In this case, we will replace the call to pass with a call to print, saying that everything is okay. Now, we'll save this other change and commit it to this branch. We are primed for chaos with our file all setup for a merge conflict. Let's check out the master branch again and try to merge the even-better-feature back into it. Git tells us it tried to automatically merge the two versions of the free memory file, but it didn't know how to do it.”



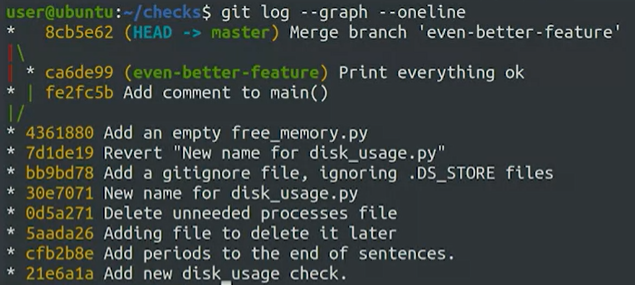
“We can use Git's status to get more information about what's going on. As usual, git status gives us a lot of additional information. It tells us that we have files that are currently unmerged, and that we need to fix the conflicts or abort the merge if we decide it was a mistake. It also tells us that we need to run Git add on each unmerged file to mark that the conflicts have been resolved.”



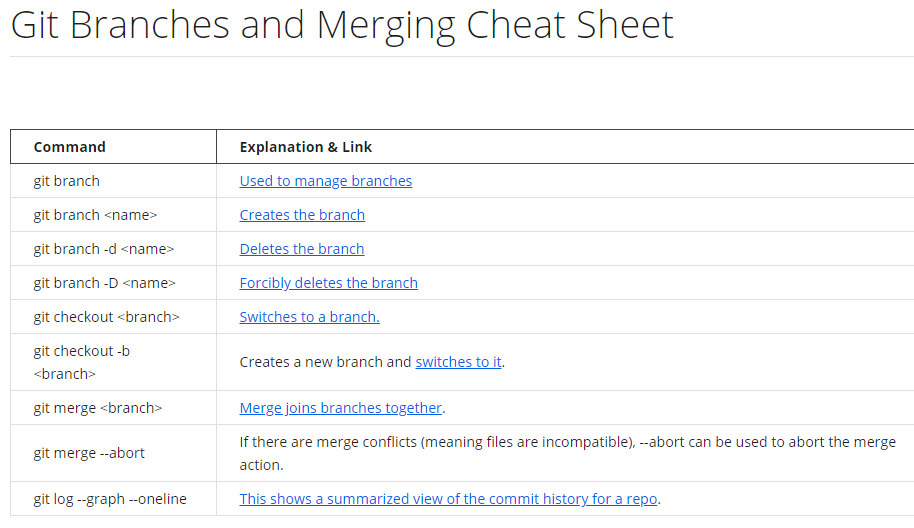
When we go an resolve the issue, git actually marks the conflict nicely:



“To see what the commit history looks like now, we'll use a couple of handy options to the git log command; --graph for seeing the commits as a graph, and --oneline to only see one line per commit. This format helps us better understand the history of our commits and how merges have occurred. We can see the new commit that was added and also the two separate commits that we merged. One coming from the master branch and the other coming from the even-better-feature branch. We can also see that master is pointing to the merge commit but even-better-feature is still pointing to the previous one”



“Merge conflicts can sometimes be tricky, complicated, and spread across multiple files. If you want to throw the merge away and start over, you can use the **git merge --abort command** as an escape hatch. This will stop the merge and reset the files in your working tree back to the previous commit before the merge ever happened.”



# Module 3 – Working with remotes

Learning Objectives

* Understand what GitHub is and how to interact with it
* Understand what a remote repository is
* Utilize remote repositories, fetch new changes, and update local repositories
* Tackle conflicts by utilizing the pull-merge-push workflow
* Push remote branches and understand the advantages of using separate branches
* Understand what rebasing is and utilize the git rebase command

### What is Github?

“In earlier videos, we called out that Git is a distributed version control system. Distributed means that each developer has a copy of the whole repository on their local machine.

Each copy is a peer of the others. But we can host one of these copies on a server and then use it as a remote repository for the other copies. This lets us synchronize work between copies through this server. Any of us can create a Git server like this one, and many companies have similar internal services. But if you don't want to set up a Git server yourself and host your repositories, you can use an online service like GitHub.

GitHub is a web-based Git repository hosting service. On top of the version control functionality of Git, GitHub includes extra features like bug tracking, wikis, and task management. GitHub lets us share and access repositories on the web and copy or clone them to our local computer, so we can work on them. GitHub is a popular choice with a robust feature set, but it's not the only one. Other services that provide similar functionality are BitBucket, and GitLab.”

“GitHub provides free access to a Git server for public and private repositories. It limits the number of contributors for the free private repositories, and offers an unlimited private repository service for a monthly fee. We'll be using a free repository for our examples, which is fine for educational use, small personal projects, or open source development.

A word of caution on how you can manage these repos though. If hackers get hold of information about your organization's IT infrastructure, they can use it to try and break into your network. So make sure you treat this information as confidential. For real configuration and development work, you should use a secure and private Git server, and limit the people authorized to work on it.”

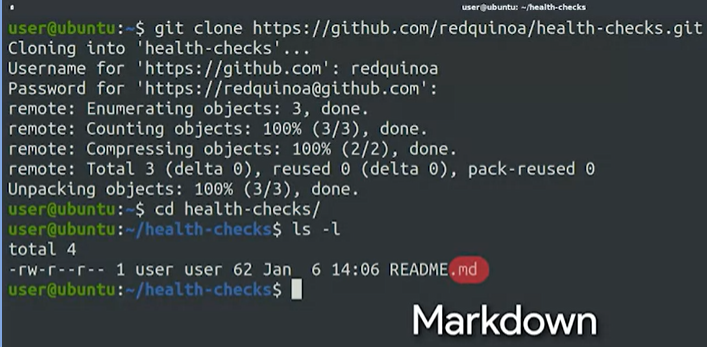
### Basic interactions with Github

“First step is to create a local copy of the repository. We'll do that by using the **git clone command** followed by the URL of the repo. GitHub conveniently lets us copy the URL from our repo from the interface so that we don't have to type it.”

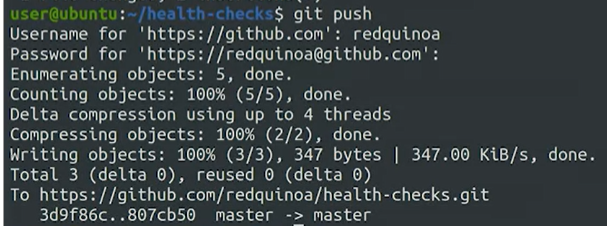


“We're now ready to clone the repo into our computer. We'll do that by calling git clone and paste in the URL we copied. To do this, GitHub will ask for our username and password.

Just like that, we've downloaded a copy of the remote repository from GitHub onto the local machine. This means that we can perform all the git actions that we've learned up till now. Since the repo is called health checks, a directory with that name was automatically created for us and now has the working tree of the Repository in it. So let's change that directory and look at the contents. Our repo is basically empty. It only has the README file that GitHub created for us. This file is in a special format called markdown.”

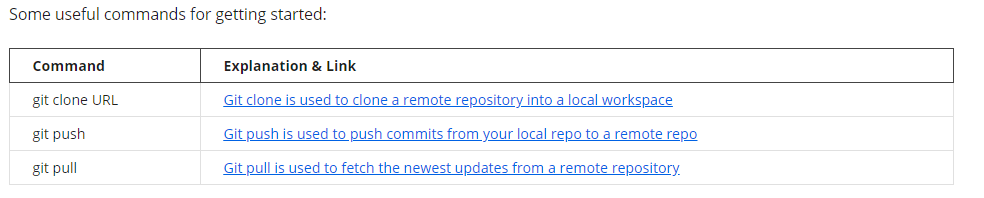


“To push our modified README up to GitHub, we'll just call **git push**.”



“You've probably noticed that we had to enter our password both when retrieving the repo and when pushing changes to the repo. There are a couple ways to avoid having to do this. One way is to create an SSH key pair and store the public key in our profile so that GitHub recognizes our computer. Another option is to use a credential helper which caches our credentials for a time window so that we don't need to enter our password with every interaction. Git already comes with a credential helper baked in. We just need to enable it. We do that by calling git config - - global credential.helper cache”





## Section 2 - Using a remote repository