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.



* We use the **--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.

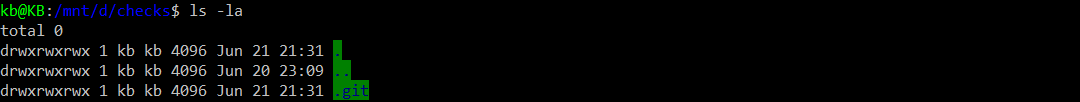
With that done, there are two ways to start working with a git repository. We can create one from scratch using the git init command or we can use the git clone command to make a copy of a repository that already exists somewhere else.



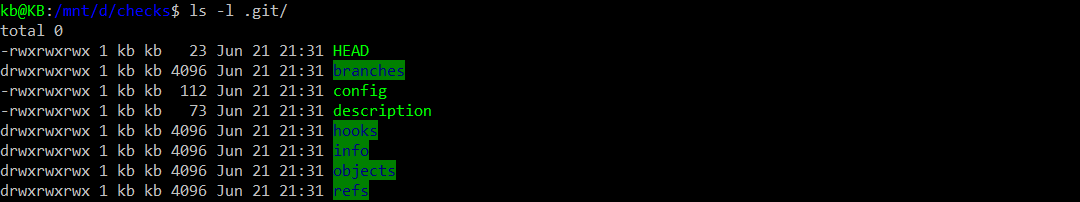
* For now, let's start by creating a new directory and then a git repository inside that directory.



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**

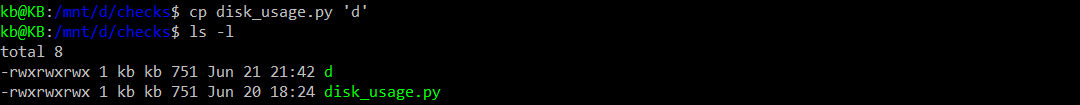


* We can check that this directory exist using the **ls -la** command which lists files that start with a dot.



* 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 we clone a repository, this git directory is copied to our computer. Whenever we 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 our project. We can think of it like a workbench or a sandbox where we perform all the modification we want to our 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.

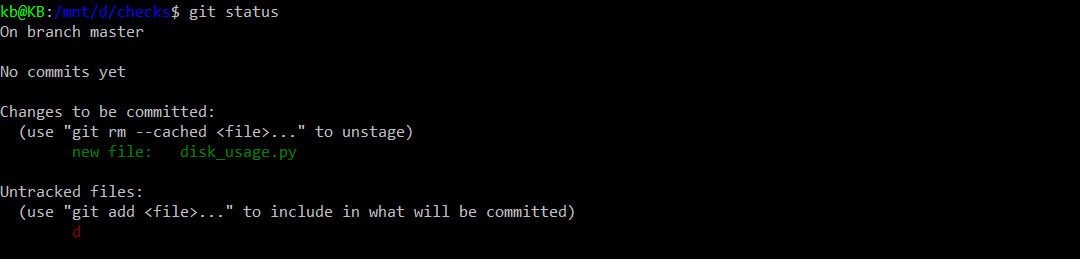


* Right now our working tree is empty. Let's change that by copying the disk\_usage.py file that we used in an earlier exercise into our current directory. We now have file and a working tree but it's currently untracked by Git.



* 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.

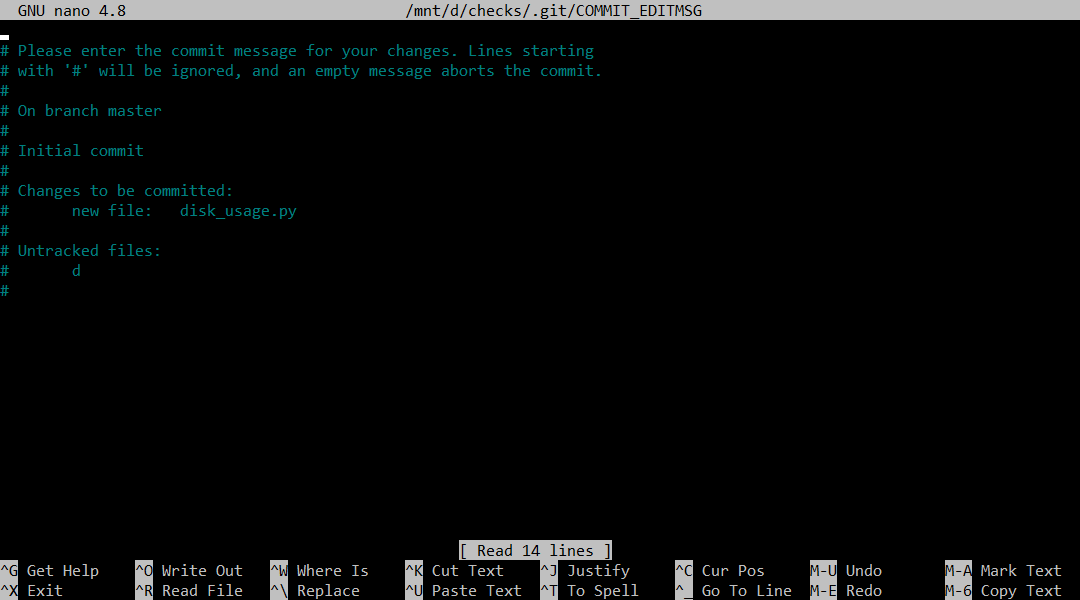
The staging area which is also known as the index is a file maintained by Git that contains all of the information about what files and changes are going to go into your next command.



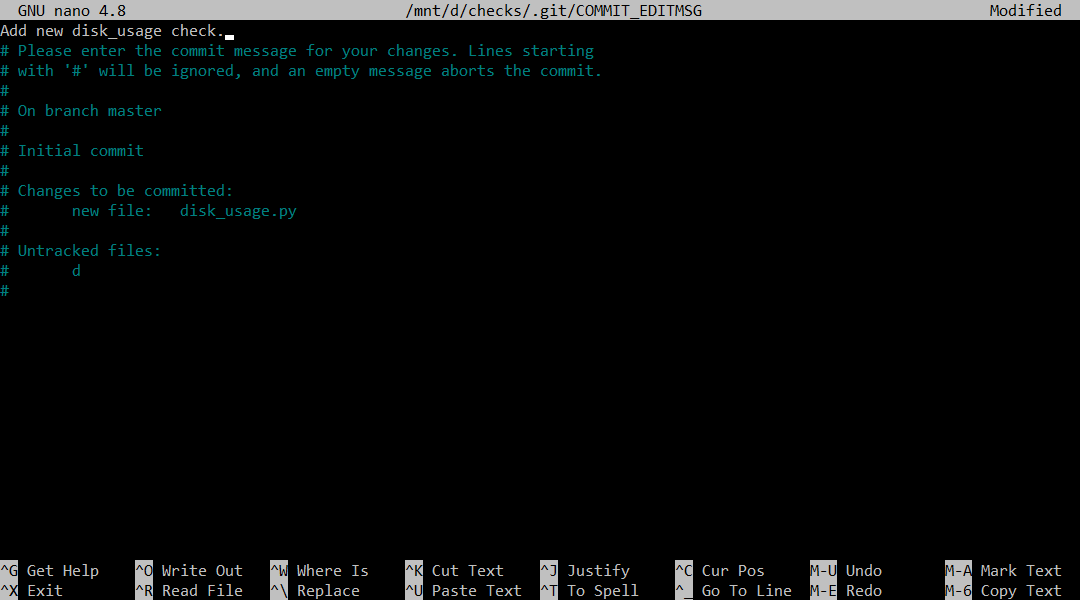
* We can use the **git status** command to get some information about the current working tree and pending changes.
* We see that our new file is marked to be committed, this means that our change is currently in the staging area.



* To get it committed into the git directory, we run the **git commit** command.



* After we run **git commit** command, it will go to this page and we also tell Git that we want to save our changes. It opens a text editor where we can enter a commit message. If you want, you can change the editor used to your preferred editor. In our case, this computer has nano configured as a default editor. The texts that we get tells us that we need to write a commit message and that the change to be committed is the new file that we've added.



* For now, let's enter a simple description of what we did which was to add this one file.



* And then exit the editor, saving our commit message by hitting Ctrl+O > Enter > Ctrl+X. And with that we've created our first git commit!

Summary:

* The git directory acts as a database for all the changes tracked in Git and the working tree acts as a sandbox where we can edit the current versions of the files.

**Tracking Files**

Any Git project will consist of 3 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.

This can still be confusing. So, it might be helpful to think about Git as representing our project which is the code and associated files and a series of snapshots. Each time we make a commit, Git records a new snapshot of the state of our 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 its information that gets stored in the Git directory.

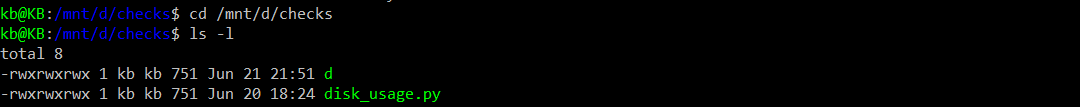
Now, let's dive into the details of how we track changes to our files. When we operate with Git, our files can be either tracked or untracked. Tracked files are part of the snapshots, while untracked files aren't a part of snapshots yet. This is the usual case for new files. Each track file can be in one of three main states, **modified**, **staged** or **committed**.

**Modified:** 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.

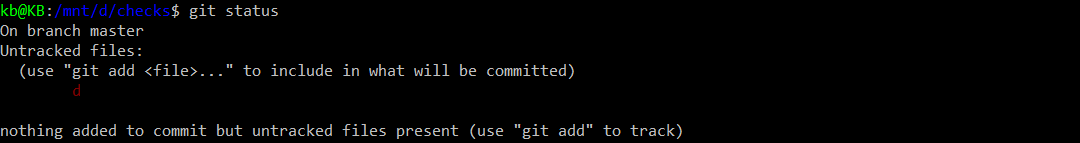
**Staged**: So, the next step is to stage those changes. When we do this, our modified files become stage files. In other words, the changes to those files are ready to be committed to the project. All files that are staged will be part of the next snapshot we take.

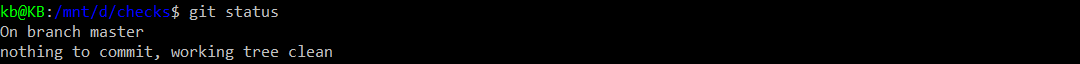
**Committed**: And finally, when a file gets committed, the changes made to it are safely stored in a snapshot in the Git directory.

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.

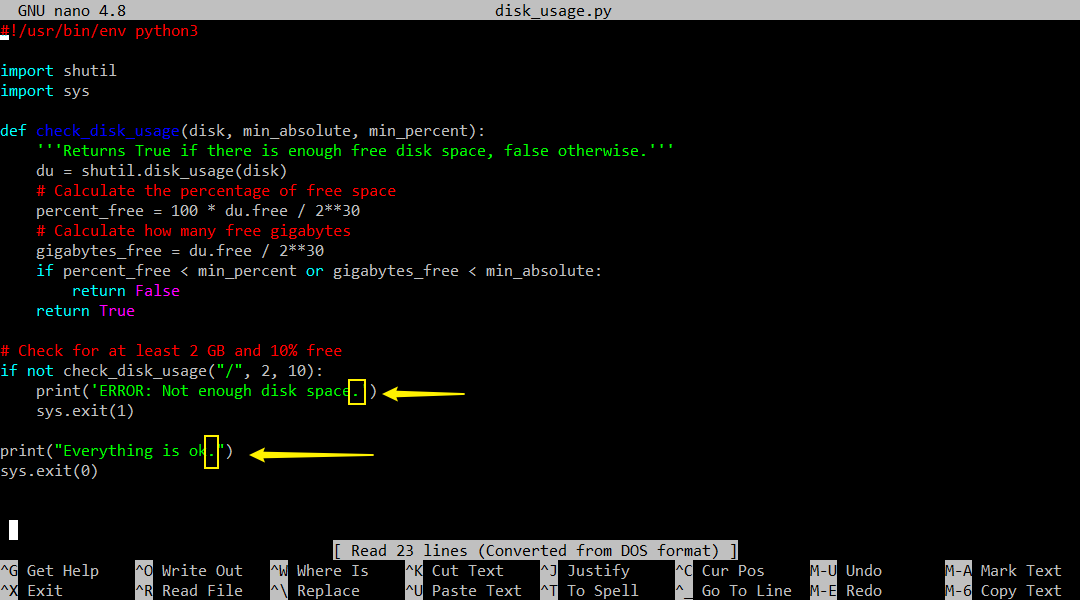


* Let's see this in action in our example Git repo. First, let's check the contents of the current working tree using **ls -l**.

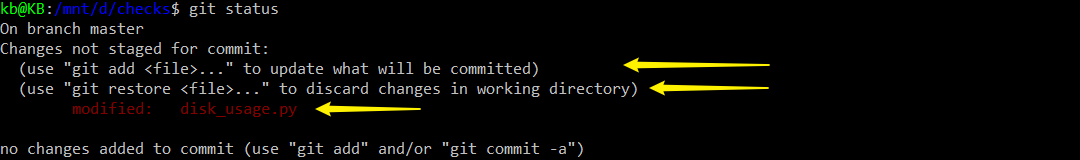




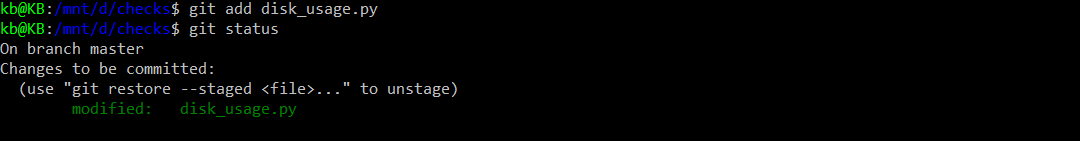
* And then the current status of our files using the **git status** command. When we run **git status**, Git tells us a bunch of things, including that we're on the master branch. We'll learn about branches later in the course. For now, notice how it says that there's nothing to commit and that the working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



* Use **nano disk\_usage.py** to go in here. For example, we'll just add periods at the end of the message that our script presents to the user, and then save it before exit.



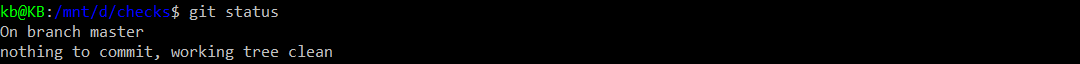
So, now that we've made the change, let's call **git status** again and see the new output. Again, Git tells us a lot of things, including giving us some tips for commands that we might want to use. These tips can come in real handy, especially when we're familiarizing ourselves with Git. We can see how the file we changed is now marked as modified. And that it's currently not staged for commit.



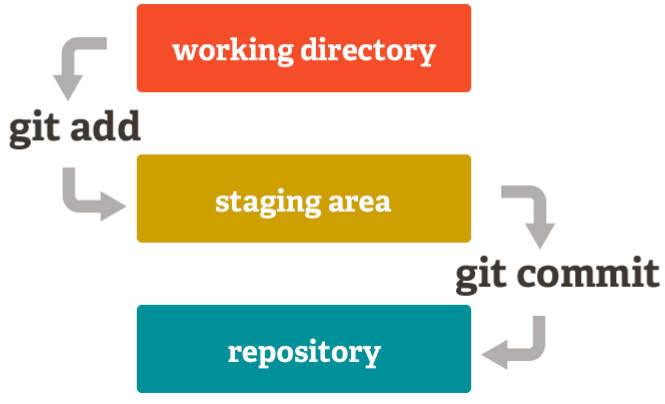
* Let's change that by running the **git add** command, passing the **disk\_usage.py** file as a parameter.
* When we call **git add**, we're telling Git that we want to add the current changes in that file to the list of changes to be committed. This means that our file is currently part of the staging 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.



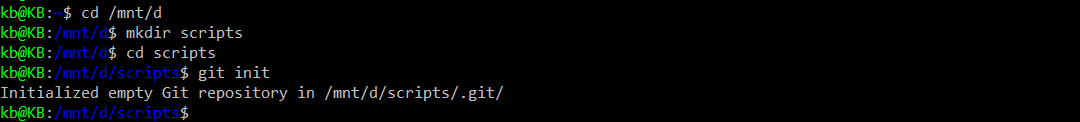
* Let's do one last status check – **git status**. We see that once again, we have no changes to commit. Because the change we made has gone through the full cycle of modified, staged and committed.



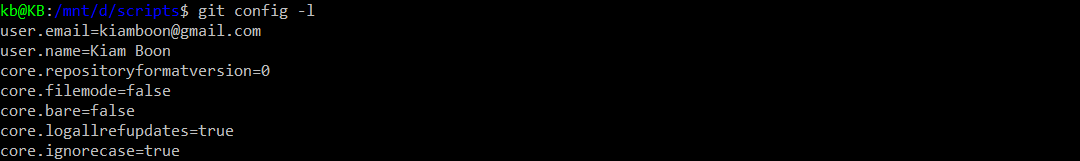
* So to sum up, we work on modified files in our **working tree** (working directory). When they're ready, we staged these files by adding them to the **staging area**. Finally, we commit the changes sitting in our staging area, which takes a snapshot of those files and stores them in the database that lives in the **Git directory** (repository).

**The Basic Git Workflow**

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.



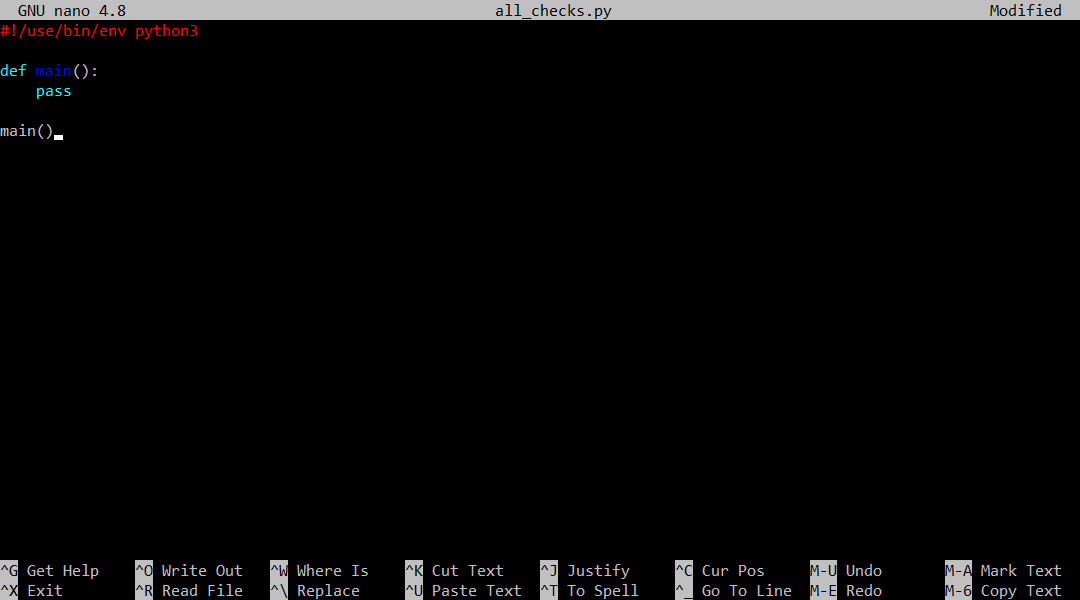
For example, let's use the mkdir command to create a directory called scripts, and then change into it and initialize an empty Git repository init. Our shiny new Git repository can now be used to track changes to files inside of it. But before jumping into that, let's check out our current configuration by using the **git config -l** command.



* Our current configuration by using the **git config -l** command shows bunch of info in there. For now, pay special attention to the user.email and the user.name lines, which we touched on briefly in an earlier exercises. 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.



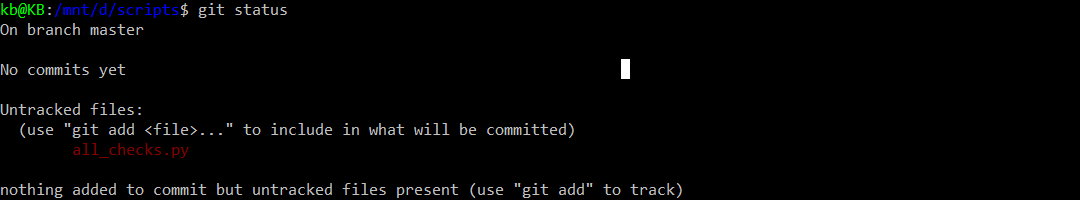
* Our repo is ready to work, but it's currently empty. Let's create a file in it, we'll start with a basic skeleton for a Python script, which will help us demonstrate the Git workflow.



* 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. Ctrl+O > Enter > Ctrl+X.



* All right, we've created our file. 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.



* As we called out before, when we create a new file in a repository, it starts off as untracked. We can make all kinds of changes to the file, but until we tell Git to track it, Git won't do anything with an untracked file. Do you remember what command we have to use to make Git track our file? That's right, we need to call the **git add** command.

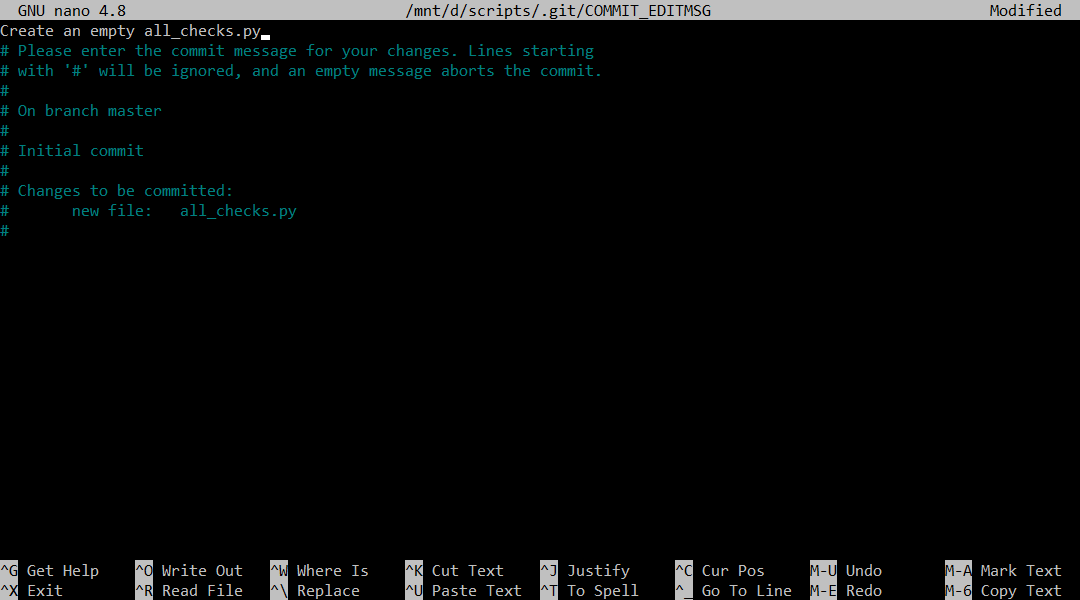


* This **git add** command will immediately move a new file from untracked to stage status. And as we'll see later, it will also change a file in the modified state to staged state.

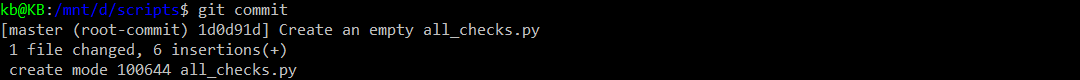
Remember that when a file is staged, it means it's been added to the staging area and it's ready to be committed to the Git repository. To initiate a commit of staged files, we issue the git commit command. When we do this, Git will only commit the changes that have been added to the staging area, untracked 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 our default editor. If the default editor is not the one we'd like to use, there are a bunch of ways to change it. We'll include more info about changing the default editor in the next reading.

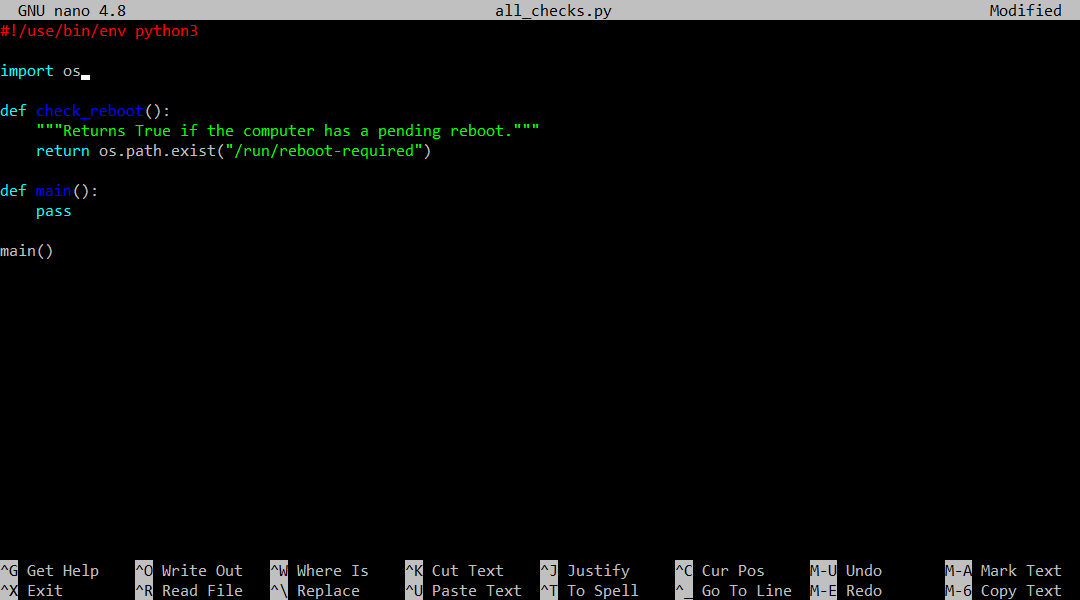


For now, let's edit our message with **nano**, which is the current default for this computer. We'll say that our change is creating an empty all\_checks.py file, then save and exit.

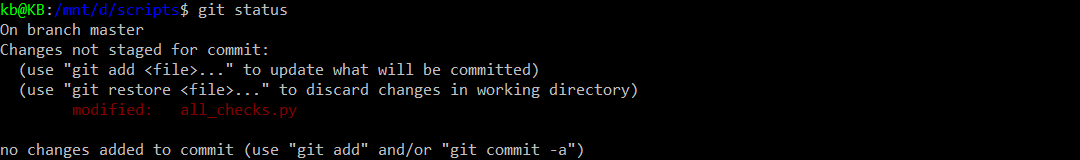


* We've just recorded a snapshot of the code in our project, which is stored in the Git directory.

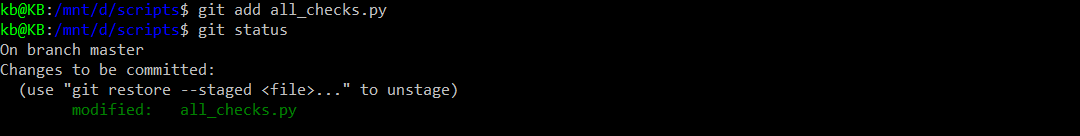
Remember that every time we commit changes, we take another snapshot, which is annotated with a commit message that we can review later. That's how we add new files, but usually we'll modify existing ones.



* So let's add a bit more content to our script to see that in action. **git commit** can’t get it, I have to use **nano** to get in to text editor.
* We'll add a function called check\_reboot, that will check if the computer is pending a reboot. To do that, we'll check if the run/reboot-required file exists. This is a file that's created on our computer when some software requires a reboot. And of course, since we're using os.path.exists, we need to add import os to our script.
* All right, 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. What do we need to do next? 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 <commit msg>**, pass in the commit message that we want to use. So in this case, we'll say that we've added the check\_reboot function.

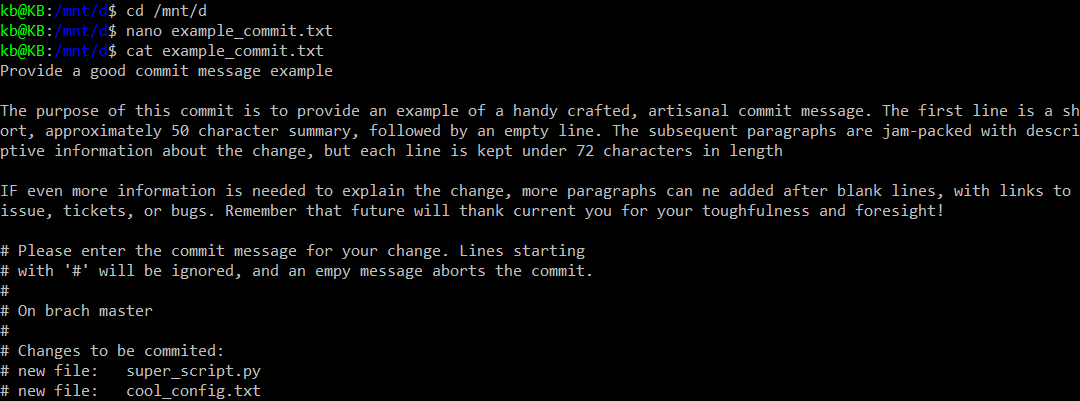
With that, we've demonstrated the basic Git workflow. We make changes to our files, stage them with **git add**, and commit them with **git commit**.

**Anatomy of a Commit Message**

Let's now talk a little bit more about what makes a good commit message. Writing a clear informative commit message is important when you use a VCS, future you or other developers or IT specialists who might read the commit message later on will really appreciate the contextual information as they try and figure out some of the parts of the code or configuration.

So what makes a good commit message? It can be helpful to keep your audience in mind when you write commit messages. What would someone reading a message weeks or months from now want to know about the changes we've made? What might be especially important or tricky to understand about them? Is there extra information that might help the reader out, like links to design documents or tickets in our ticketing system? Similarly to how style guides exist for writing code, your company might have specific rules for you to follow when you write commit messages. Even if they don't, it's good to use a few general guidelines to make sure your commit messages are as clear and useful as possible.

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. A good commit message might look something like this:

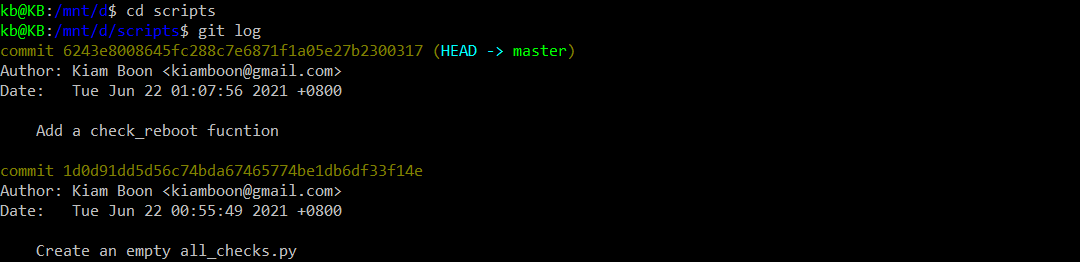


So the first line is usually kept to about **50 characters or less**. The line contains a short description of what the commit changes are about. After the first line, comes an empty line, and the rest of the text is usually kept under 72 characters. This text is intended to provide a detailed explanation of what's going on with the change. It can reference bugs or issues that will be fixed with the change. It can also include links to more information when relevant. The line limits can be annoying but they help in making the commit message be more digestible for the reader. There's a git command used to display these commit messages called **git log**. This command will do any line wrapping for us. Which means that if we don't stick to the recommended line wrapping, long commit messages will run off the edge of the screen and be difficult to read.

Now, take a look at the lines in the commit message that start with the (#) symbol. Just like in Python, this symbol indicates that these lines are comments and won't get included in the commit message. Git shows them to us whenever we're writing a commit message as a reminder of what files were about to commit.

Sometimes it can be tempting to just write something short like update, change or fix as the description of our commit messages. Don't do it. It's super frustrating to go back to repositories history and discover that there's not enough context to understand what was changed and why. It takes only a few more seconds to write a better description. This can be invaluable down the line.

We said that we can check the history of the commits of our project using the **git log** command. Let's go back to our example scripts directory where we performed two commits and check out what git log has to say about those two commits.



* Take a look at what git tracks as part of the log. It's packing a lot of information in just a few lines. The first thing listed for each commit is its identifier, which is a long string of letters and numbers that uniquely identify each commit.
* The first commit in the list also says that the head indicator is pointing to the master branch. If this is gibberish to you, don't worry. We'll talk more about what a head and master means in later videos.
* For each commit, we see the name and the email of the person who made the commit which is indicated as the author. Then we get the date and time the commit was made.
* Finally the commit message is displayed. Our commit messages are very brief as we're just starting to work on our repository. As the work we do becomes more complex, we'll probably write longer descriptions with a lot more detail.