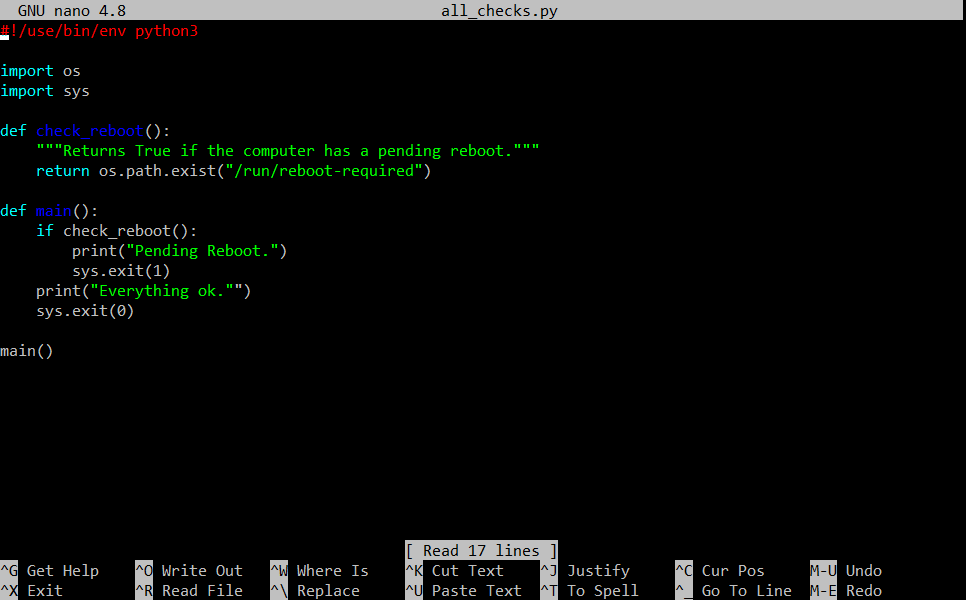
Undoing Things

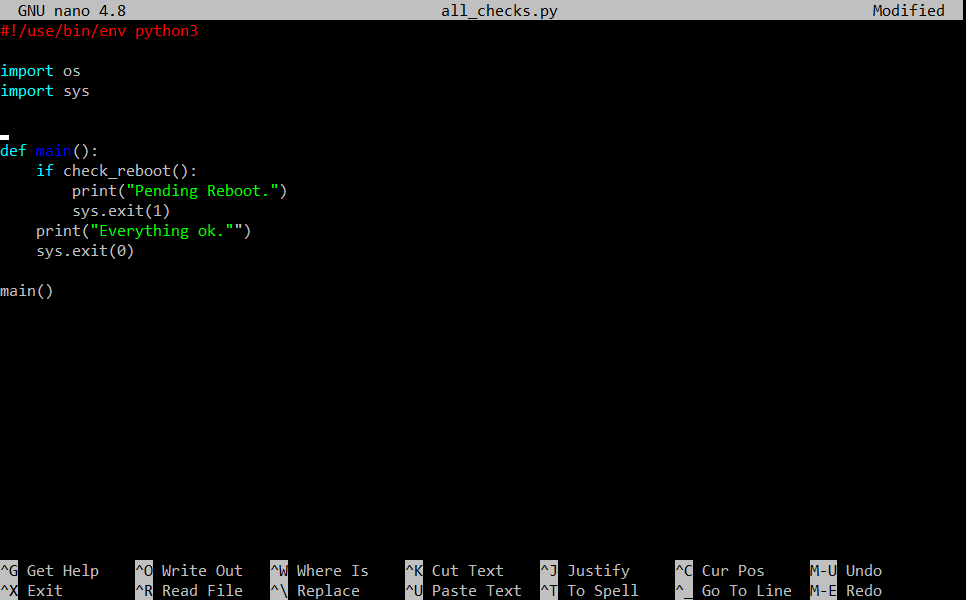
**Undoing Change before Committing**

Being able to revert our changes is one of the most powerful features offered by version control systems. There's a bunch of different techniques available depending on which changes we need to undo. In this exercise and the next few coming up, we'll talk about the most common ways to revert changes in Git and when to use each approach. For example, we might find ourselves in a situation where we’ve made a bunch of changes to a file but decide that we don't want to keep them. We 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. Speaking of, let's try this out using our scripts repository.

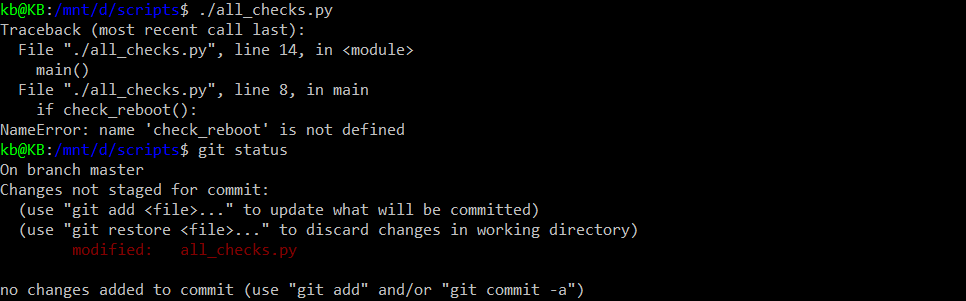
**git checkout** restores files to the latest stored snapshot, reverting any changes before staging. It reverts changes to modified files before they are staged.



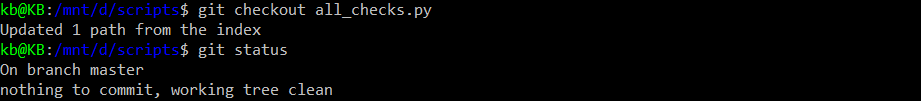




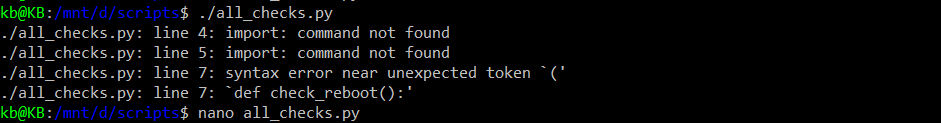
* We'll edit our all\_checks.py script and remove the check reboot function, then save and go back to the command line.
* Now, we've made our change. Let's try our script and see what happens.



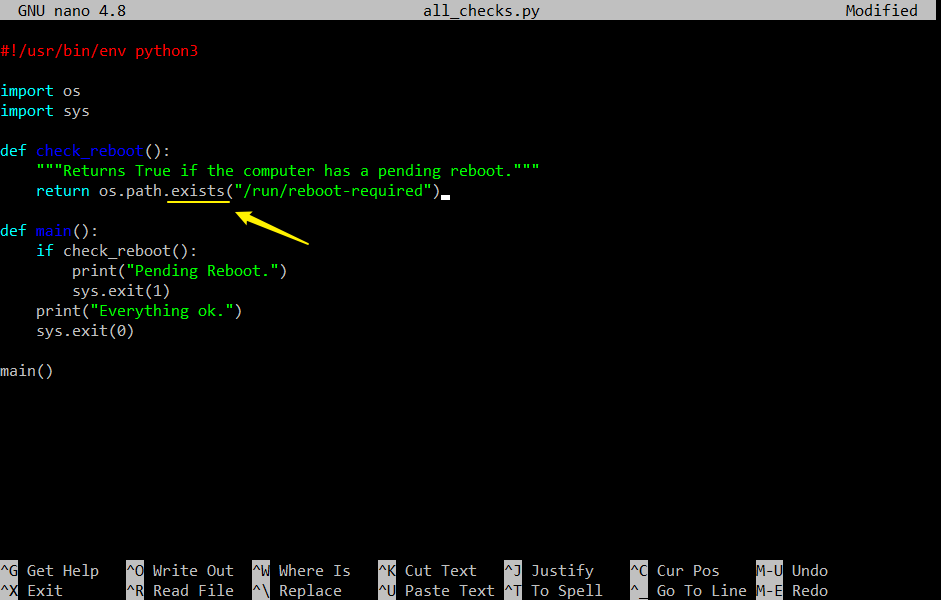
* By deleting that function, we've actually broke the script.
* Let's see what **git status** has to say about this. As expected, we see that our file is modified and the changes aren't staged yet. Check out how git gives us a couple helpful tips on what to do now.
* We can run **git add** to stage our changes or we can run **git checkout** to discard them. If we need help remembering what this command does, think of it this way, we’re checking out the original file from the latest storage snapshot.



* Let's do that now. We'll check out at the original file and then take a look at what **git status** has to say about it and finally retry our script.



* Looks like we have a typo. Let's go back to the all\_checks.py and fix it.

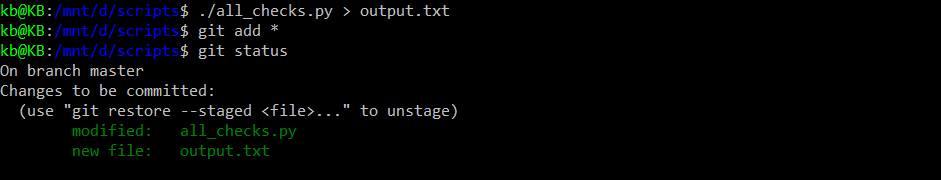


* Should be “exists” instead of “exist”

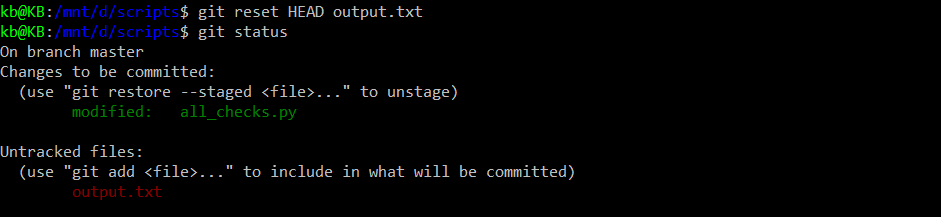


* Done. With that, we've demonstrated how we can use git checkout to revert changes to modify files before they get staged. This command will restore the file to the latest storage snapshot, which can be either committed or staged.
* So if we’ve made additional changes to a file after we’ve staged it, we can restore the file to the earlier stage version. If we need to check out individual changes instead of the whole file, we can do that using the -p flag. This will ask us change by change if we want to go back to the previous snapshot or not.

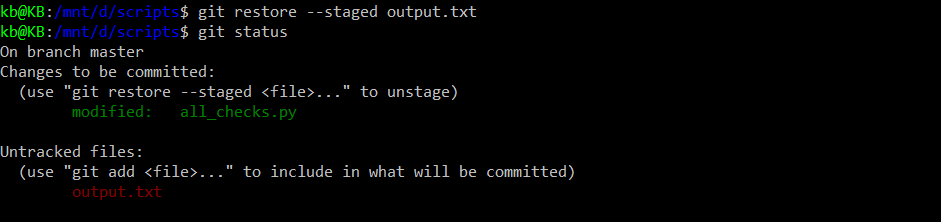
That's it for undoing unstaged changes. What if we added the changes to the staging area already? Don't stress. If we realize we've added something to the staging area that we didn't actually want to commit, we can unstage our changes by using the **git reset** command. Staging changes that we don't actually intend to commit happens all the time. Especially if we use a command like **git add \***, where the **\*** is a file glob pattern used in Bash that expands to all files. This command will end up adding any change done in the working tree to the staging area. While sometimes that might be what we want, it can also lead to some surprises. Let's try it out with an example.



* First, we'll pretend we're trying to debug a problem in our script. For that, we create a temporary file with the output of our script. Then, we'll add all unstaged changes in our working tree using **git add \***. Finally, check the status using **git status**.
* We can see that this output file, which was supposed to be a temporary file for debugging, has now been staged in our repo but we didn't want to commit it.
* Conveniently, the **git status** command tells us how to unstage the file right there in the output. The example output mentions the HEAD alias. Remember what that means? That's right. It's the current checked out snapshot. So by running the suggested command, we're resetting our changes to whatever's in the current snapshot. Let's try it out.
* Note: The suggestion in tutorial shows “Use git reset HEAD <files>..” to unstage, but for my suggestion in Ubuntu shows use "git restore --staged <file>..." to unstage



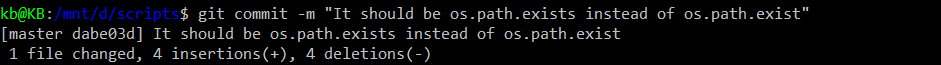
1. Option 1: This is running according by tutorial **git reset HEAD output.txt**



1. Option 2: This is running according to my Ubuntu’s suggestion **git restore --staged output.txt**

* Looks like both **git reset HEAD <filename>** and **git restore --staged <filename>**.
* The file is once again untracked in our working tree and no longer staged.

You can think of reset as the counterpart to add. With add, you can well add changes to the staging area. With reset, we remove changes from the staging area. We can use **git reset -p** to get git to ask you which specific changes we want to reset.



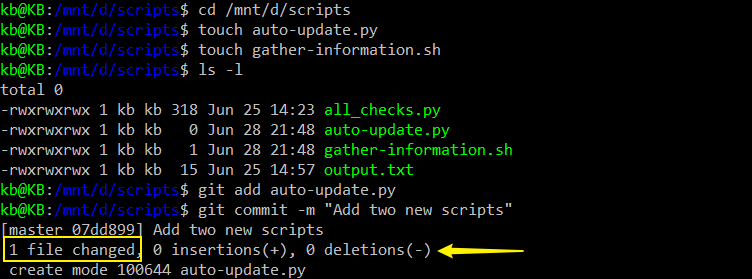
* But wait, let's remember to commit our typo fix.

With that, we've seen how we can revert unstaged and stage changes. But what if we’ve already created a commit with the changes that you want to undo? Great question. That's coming up in the next exercise.

**Amending Commits**

In general, we try to make sure our commits include all the right changes and descriptions. But we're all human and we make mistakes. It's not uncommon for developers and IT specialists to realize that there is an error in a recent commit, which is why it's important to know how to take action and fix it. Let's say you just finished committing our latest batch of work, but we’ve forgotten to add a file that belongs to the same change. We’ll want to update the commit to include that change. Or maybe the files were correct, but we realize that our commit message just wasn't descriptive enough. So we want to fix the description to add a link to the bug that we’re solving with that commit. What can you do? We can solve problems like these using the **--amend** option of the git commit command. When we run git commit **--amend**, git will take whatever is currently in our staging area and run the git commit workflow to overwrite the previous commit. **git commit --amend** allows us to overwrite/modify the previous commit and add changes to the most recent commit.

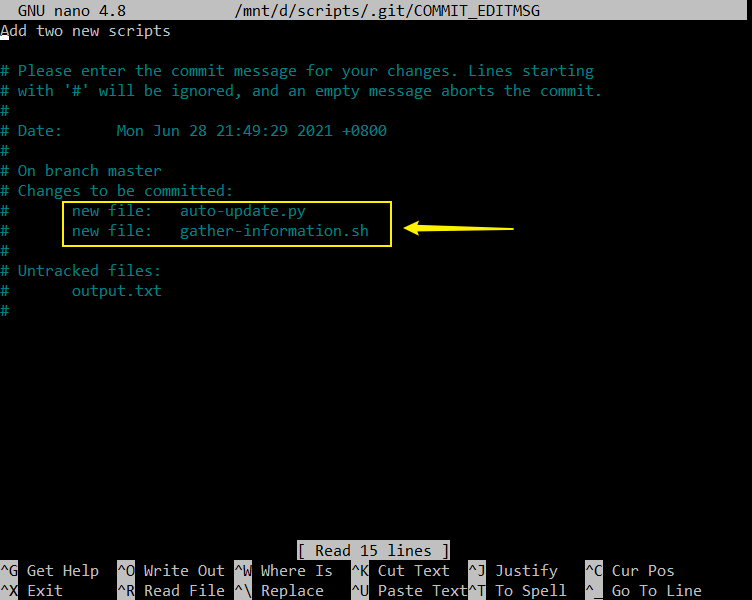
Let's see this in an example.



* We'll go to our scripts directory and create two new files using the touch command.
* Then list the contents of the directory using ls at our Python script and commit it saying that we've added two files.
* As we can see, the message printed by git says that only one file was added. Our commit message said that we added two files, but we forgot to add one of them.

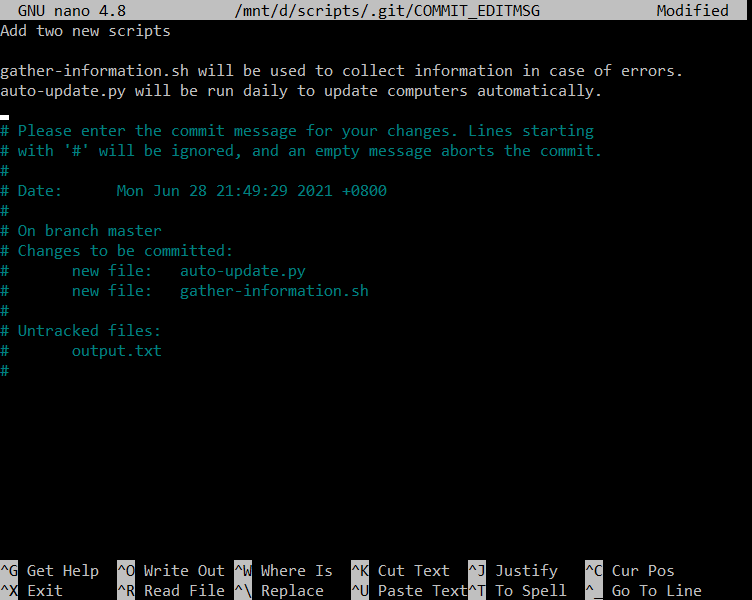


* We can fix it. We'll start by adding the missing file and then amending our commit.
* We call **git commit --amend** and…

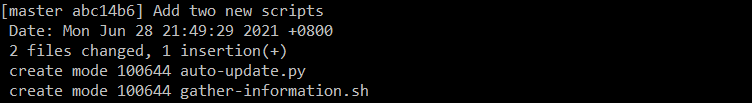


* … an editor opened up showing the commit message and the stats about the commit that we're working with.
* The list of added files for this commit now includes both files that we wanted to add.

Yay. Now that the files have been added, we can also improve our initial commit message which was a bit too short.



* We'll keep the existing description as the first sentence of our commit, and then add a line of description about the intended purpose of each file.
* With that, our commit is ready to be amended. Let's save the new description as usual.



* After save and exit, we will come back to the main page.
* We've amended our previous commit to include both files and a better message.

We could also just update the message of the previous commit by running the **git commit --amend** command with no changes in the staging area. An important heads up. While **git --amend** is okay for fixing up local commits, we shouldn't use it on public commits. Meaning, those that have been pushed to a public or shared repository. This is because using **--amend** rewrites the git history removing the previous commit and replacing it with the amended one. This can lead to some confusing situations when working with other people and should definitely be avoided.

So remember, fixing up a local commit with amend is great and we can push it to a shared repository after we fixed it. But we should avoid amending commits that have already been made public.

**Rollbacks**

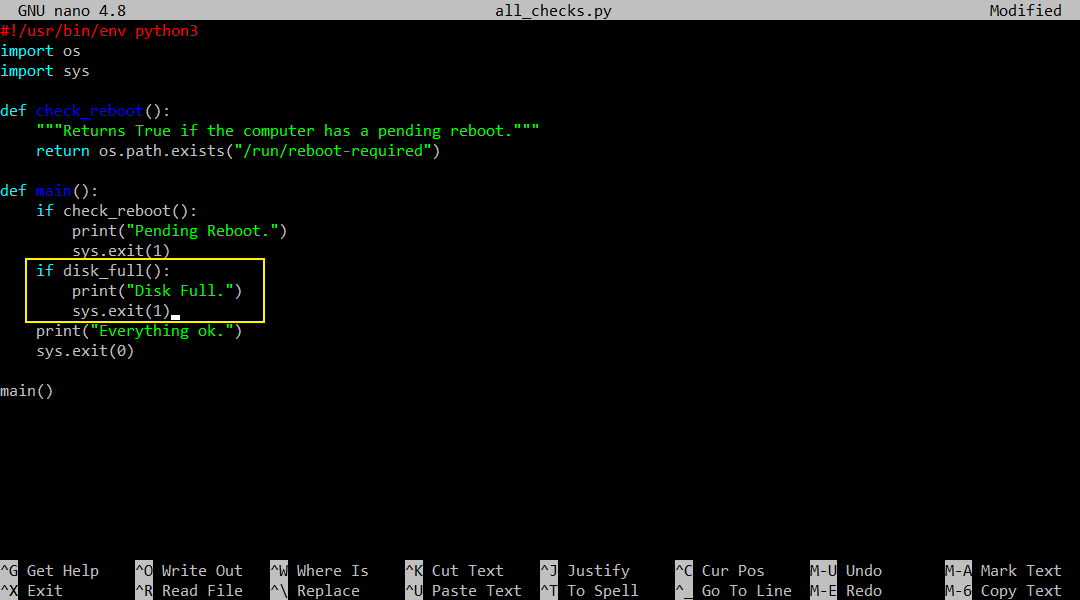
Fixing our work before you commit is good. But what happens if it's already been snapshotted by Git? Let's say you host to Git repository on a company server that contains all kinds of useful automation scripts that you and our co-workers use. One morning before coffee, you make a few changes to one of these scripts and commit the updated files. A few hours later, you start to receive tickets from users indicating some part of the script is broken. From the errors they describe, it sounds like the problem is related to our recent changes. Oh-oh, you could look at the code you updated to see if we can spot the bug. But more tickets are pouring in and we want to fix the problem as fast as possible. You decided it's time for a rollback.

There are a few ways to rollback commits in Git. For now, we'll focus on using the **git revert** command. **Git revert** doesn't just mean undo. Instead, it creates a commit that contains the inverse of all the changes made in the bad commit in order to cancel them out. With git revert, a new commit is created with inverse changes. This cancels previous changes instead of making it as though the original commit never happened.

For example, if a particular line was added in the bad commit, then in the reverted commit, the same line will be deleted. This way you get the effect of having undone the changes, but the history of the commits in the project remains consistent leaving a record of exactly what happened. So **git revert** will create a new commit, that is the opposite of everything in the given commit. We can revert the latest commit by using the **HEAD** alias that we mentioned before. Since we can think of **HEAD** as a pointer to the snapshot of our current commit, when we pass head to the revert command we tell Git to rewind that current commit, makes sense? To check this out, we'll first add a faulty commit to our example repo.



* Go into all\_chcks.py and edit it.



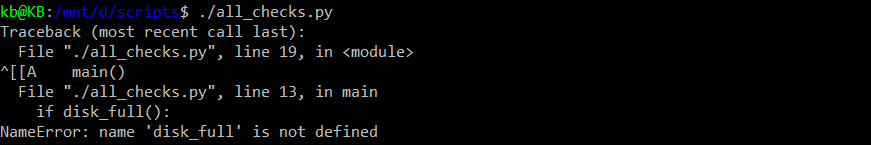
* We've added some code to our script, then save it.



* and then commit this.

So now, our code is committed. We didn't even test it which is a bad idea if we’re doing this for real. You might have already spotted the problem with our code. This is where users start filing

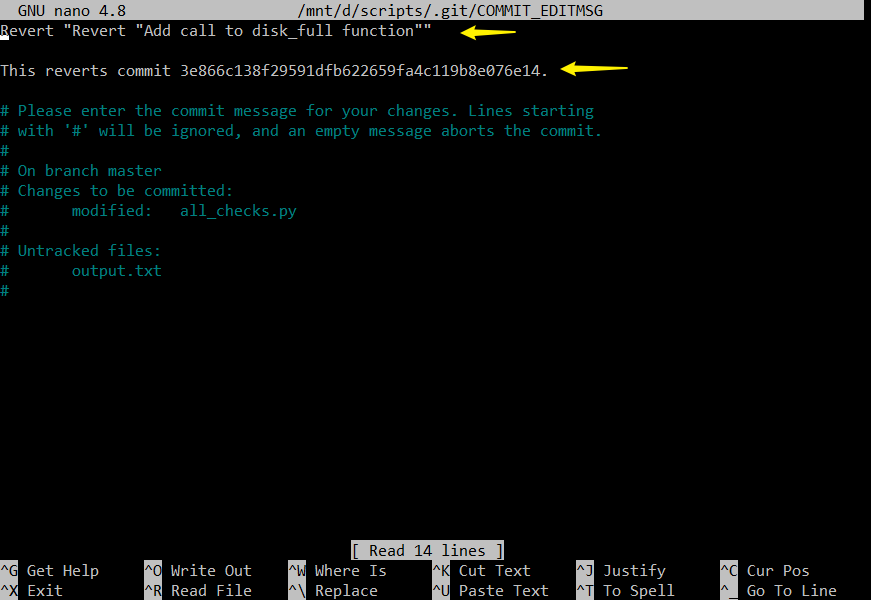
tickets and saying that things are broken…



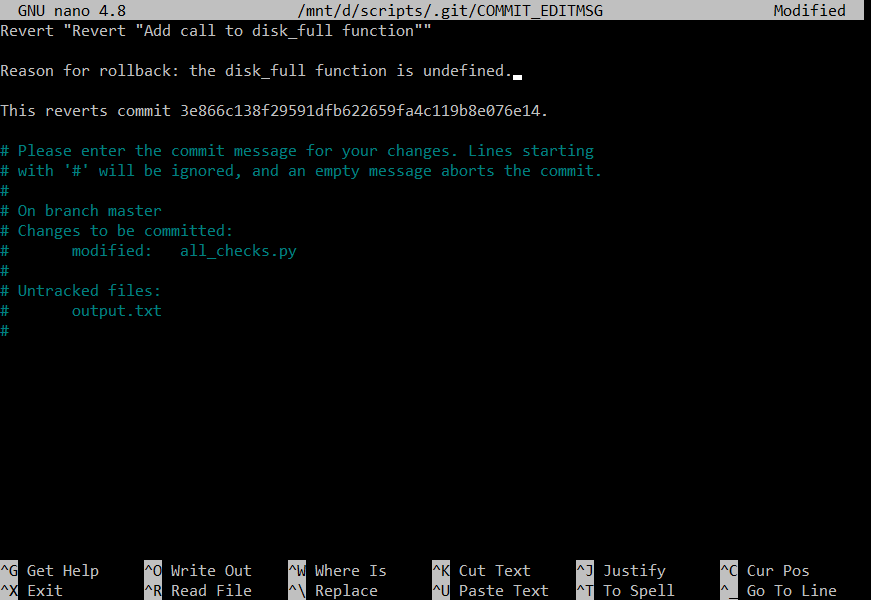
* ...And so we run our script to see what happens. Oops, we use the function that we forgot to define.



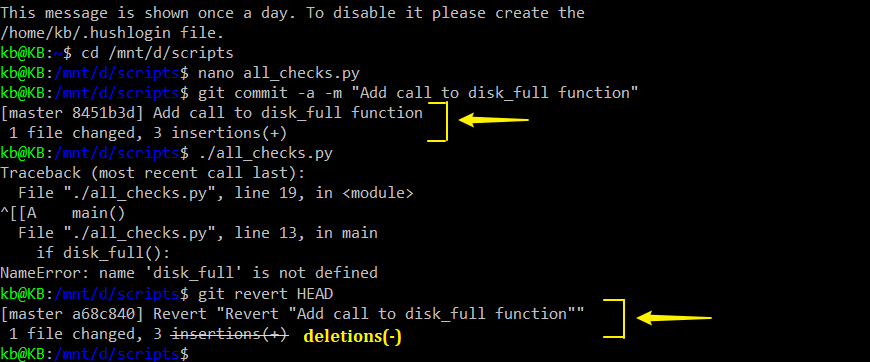
* Okay. It's rollback time. Let's get rid of this faulty code by typing **git revert HEAD**.



* Once we issue that **git revert** command, we're presented with the text editor commit interface that we've all seen before.
* In this case, we can see that git has automatically added some text to the command indicating it's a rollback.
* The first-line mentions that it's reverting the commit we just did called Add call to disk full function.
* The extra description even includes the identifier of the commit that got reverted.



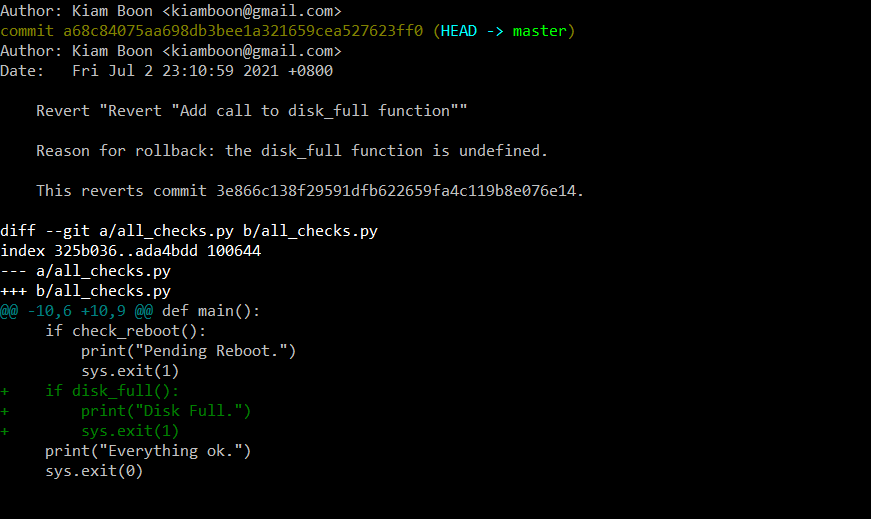
* While we could use this description as is, it's usually a good idea to add an explanation of why we're doing the rollback. Remember that the goal of these descriptions is to help our future selves understand why things happen.
* In this case, we'll explain that the reason for the rollback is that the code was calling a function that wasn't defined.
* Once we're done entering the description, we can exit and save as usual.

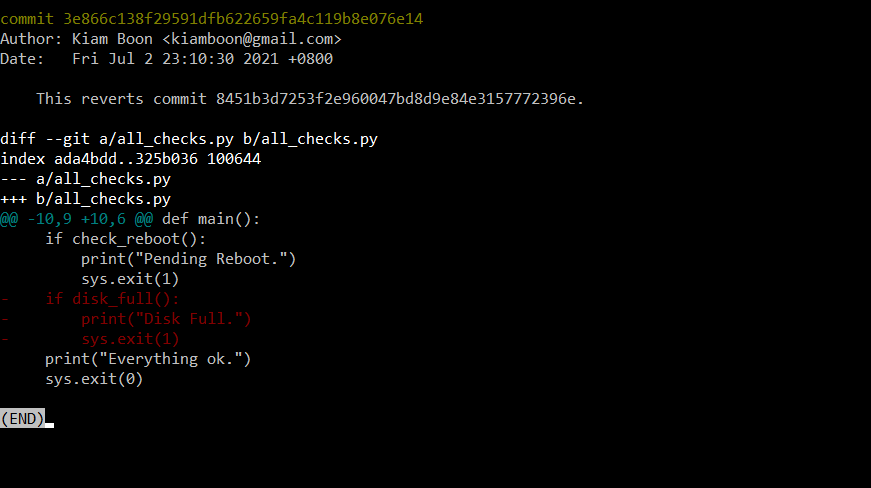


* We’ll notice the output that we get from the **git revert** command looks like the output of the **git commit** command. This is because **git revert** creates a commit for us. Since a revert is a normal commit, we can see both the commit and the reverted commit in the log.



Let's look at the last two entries in the **git log** using **-p** and **-2** as parameters.

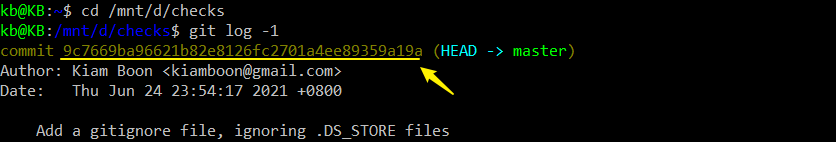




* As demonstrated before, the **-p** parameter lets us see the patch created by the commit while the **-2** perimeter limits the output to the last two entries.
* So in this log, we can see that when we called revert, git created a new commit that's the inverse of the previous one.
* The text in red with minus sign (-) means removing the lines that we added in the previous commit.
* We can see that the original commit shows the lines we added by preceding them with a plus sign in green text.
* Just like that, the bad commit is reverted and the error stopped.

**Identifying a Commit**

So far we've used the **HEAD** alias to specify the most recently checked out commit in our Git history. In our bad snapshot example, the error also happened to be in the most recently created commit, but errors can sometimes take a while to be detected. And so, we might need to revert other commits farther back in time. We can target a specific commit by using its commit ID. We've seen commit IDs a few times already. They show up when we're running the **git log** command, and we also saw the commit ID of the reverted commit in our last example. Commit IDs are those complicated looking strings that appear after the word commit in the log messages. Let's have a look at the latest log entry in our checks repo.



The commit ID is the 40 characters long string after the word commit, we really can't miss it. This long jumble of letters and numbers is actually something called a hash, which is calculated using an algorithm called **SHA1**. Essentially, what this algorithm does is take a bunch of data as input and produce a 40 characters string from the data as the output. In the case of Git, the input is all information related to the commit, and the 40 characters string is the commit ID.

Cryptographic algorithms like SHA1 can be really complex, so we won't go too deep into what this means. Still you might be wondering, why on earth would you use a long jumble of letters as an ID for commit, instead of incrementing an integer, like 123, etc?

To answer that, let's take a quick look at the reason why Git uses a hash instead of a counter, and how that hash is computed. Although SHA1 is a part of the class of cryptographic hash functions, Git doesn't really use these hashes for security. Instead, they're used to guarantee the consistency of our repository.

*How SHA1 hash numbers that Git uses to identify commits?*

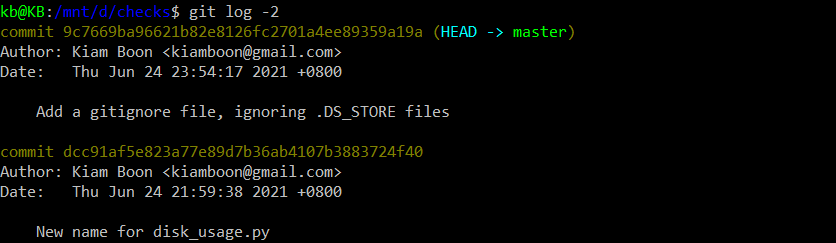
* *They provide the consistency that is critical for distributed systems such as Git.*
* *They are created using the commit message, date, author, and the snapshot taken of the working tree.*
* *They are composed of 40 characters.*

Having consistent data means that we get exactly what we expect. To quote Git's creator, Linus Torvalds, “You can verify the data you get back out is the exact same data you put in.” This is really useful in distributed systems like Git because everyone has their own repository and is transmitting their own pieces of data. Computing the hash keeps data consistent because it's calculated from all the information that makes up a commit. The commit message, date, author, and the snapshot taken of the working tree. The chance of two different commits producing the same hash, commonly referred to as a collision, is extremely small. So small, it wouldn't happen by chance. It'd take a lot of processing power to cause this to happen on purpose. If we use a hash to guarantee consistency, we can't change anything in the Git commit without the SHA1 hash changing too.

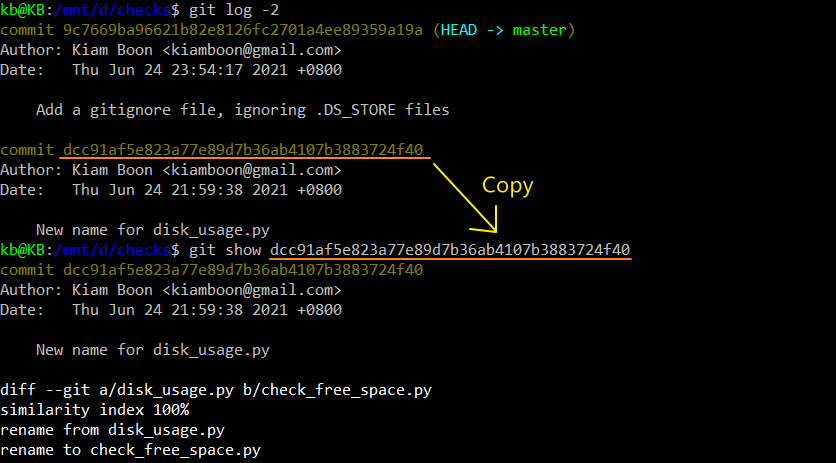
Remember our discussion about fixing commits with the **--amend** command? Each time we amend a commit, the commit ID will change. This is why it's important not to use **--amend** on commits that have been made public.

The data integrity offered by the commit ID means that if a bad disk or network link corrupt some data in our repository, or worse, if someone intentionally corrupt some data, Git can use the hash to spot that corruption.

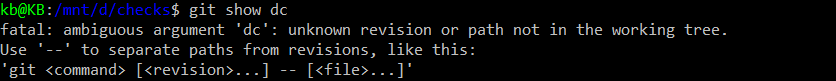
How can we use commit IDs to specify a particular commit to work with, like during a rollback?



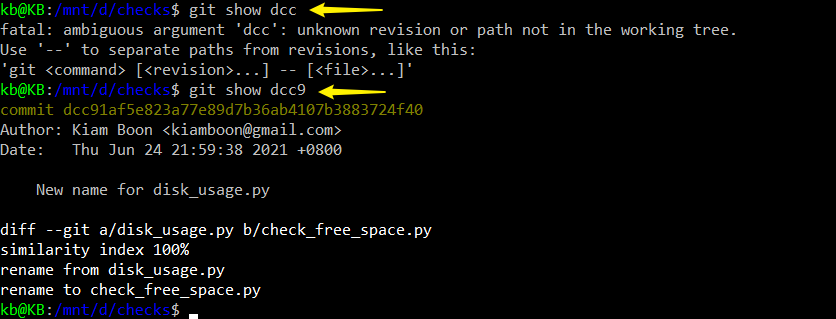
Let's look at the last two entries in our repo using the **git log -2** command.



* Say we realized that we actually liked the previous name of our script, and so we want to revert this commit where we renamed it. First, let's look at that specific commit using **git show**, which we mentioned in an earlier exercise.
* We've copied and pasted the commit ID that we wanted to display, and that works.



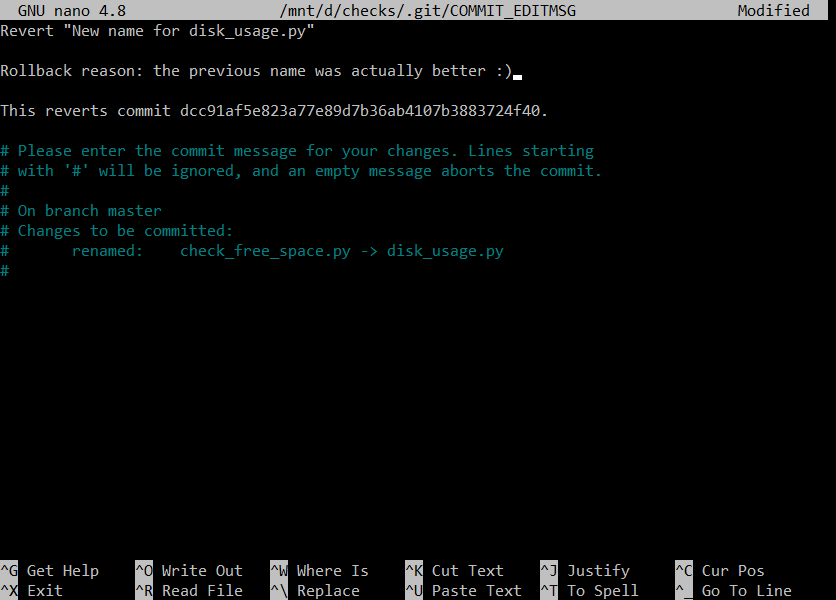
* Alternatively, we could provide just the first few characters identifying the commit to the command, and Git will be smart enough to guess which commit ID starts with those characters, as long as there's only one matching possibility.
* Let’s try using 2 characters for our commit ID by typing ‘dc’ only first…
* 2 characters is not enough, but usually four to eight characters will be plenty.



* Let’s try 3 characters… 3 characters also not enough.
* Let’s try 4 characters… now that we've seen how we can identify the commit that we want to revert…



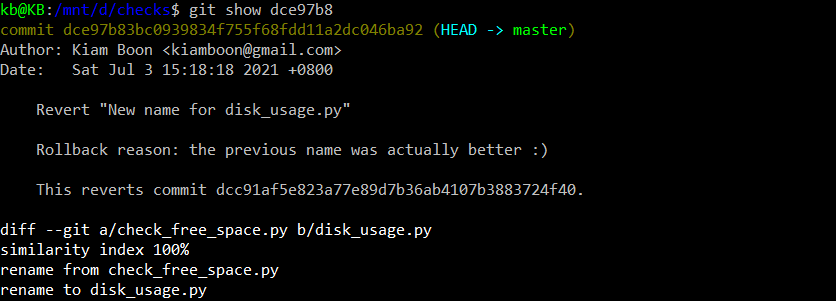
* Let's call the git revert command with this identifier



* As usual, this will open an editor where we should add a reason for the rollback. In this case, we'll say that the previous name was actually better.
* As we called out before, when we generate the rollback, Git automatically includes the ID of the commit that we're reverting. This is useful when looking at a repo with a complicated history that includes a lot of commits. Now, once we save and exit the commit message, Git will actually perform the rollback and generate a new commit with its own ID.



* See how before the name of our commit the revert command already shows the first 7 characters of the commit ID? Let's use git show to look at it.



* All right, we've managed to revert a commit that wasn't the most recent one.

**Git Revert Cheat Sheet**

* [git checkout](https://git-scm.com/docs/git-checkout) is effectively used to switch branches.
* [git reset](https://git-scm.com/docs/git-reset#_examples) basically resets the repo, throwing away some changes. It’s somewhat difficult to understand, so reading the examples in the documentation may be a bit more useful.
* There are some other useful articles online, which discuss more aggressive approaches to [resetting the repo](https://jwiegley.github.io/git-from-the-bottom-up/3-Reset/4-doing-a-hard-reset.html).
* [git commit --amend](https://git-scm.com/docs/git-commit#Documentation/git-commit.txt---amend) is used to make changes to commits after-the-fact, which can be useful for making notes about a given commit.
* [git revert](https://git-scm.com/docs/git-revert) makes a new commit which effectively rolls back a previous commit. It’s a bit like an undo command.
* There are a [few ways](https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things) you can rollback commits in Git. There are some interesting considerations about how git object data is stored, such as the usage of sha-1. Feel free to read more here:

1. <https://en.wikipedia.org/wiki/SHA-1>
2. <https://github.blog/2017-03-20-sha-1-collision-detection-on-github-com/>