Advanced Git Interaction

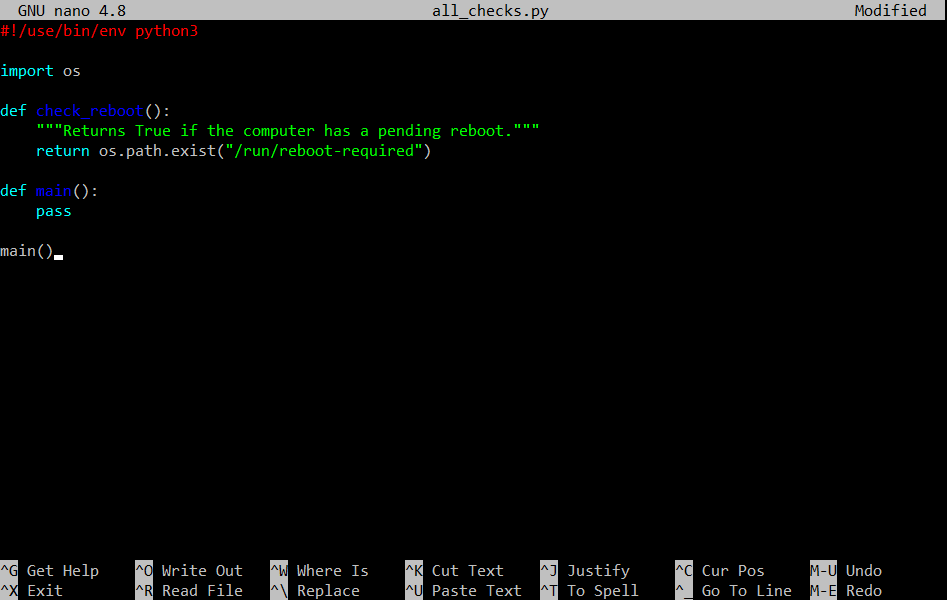
**Skipping the Staging Area**

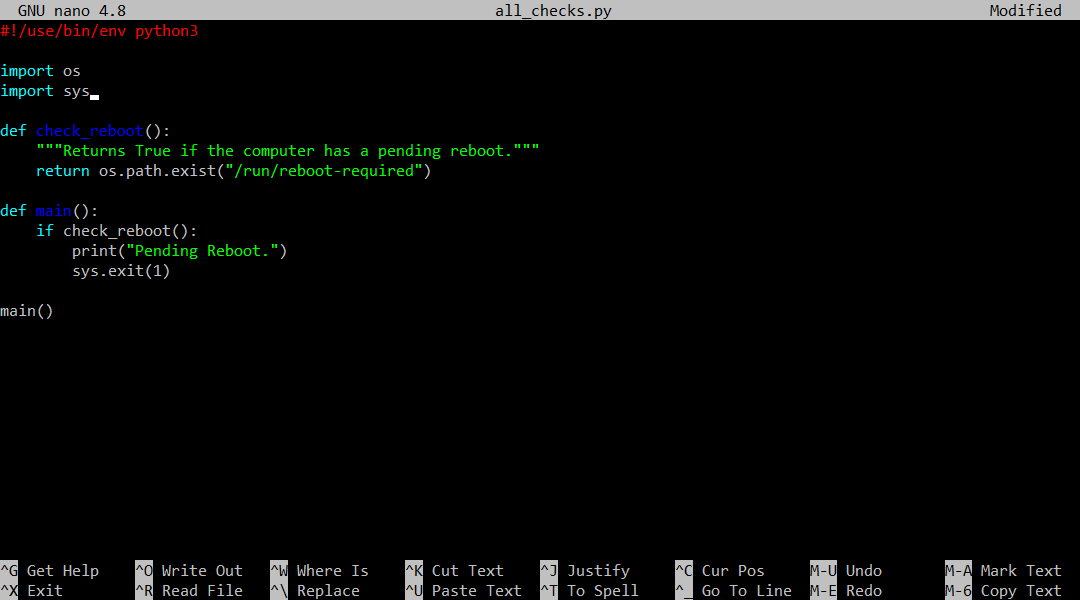
The basic Git workflow, we called out that the process is usually to make changes, stage them, and then commit them. The separate step between staging and committing allows us to stage several changes in one commit. But if we already know that the current changes are the ones that we want to commit, 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 -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.



* So let's make a change to our example script – all\_checks.py from an earlier exercise and try out this new flag.





* We'll now modify our main function and make it call the check reboot function that we wrote before. If a reboot is pending, we'll print a message and then exit our program with an exit status of one. Since we're using the sys module, we'll need to import it.



* All right. Now that we've made the change, we're ready to try out the new **-a** flag. We'll also use the **-m** flag to add the commit message directly. This time, we'll say that we're calling check underscore reboot and exiting with one on the error condition.
* These shortcuts are useful when making small changes that we know we'll want to commit directly without keeping them in the staging area and having to write long and complex descriptions.

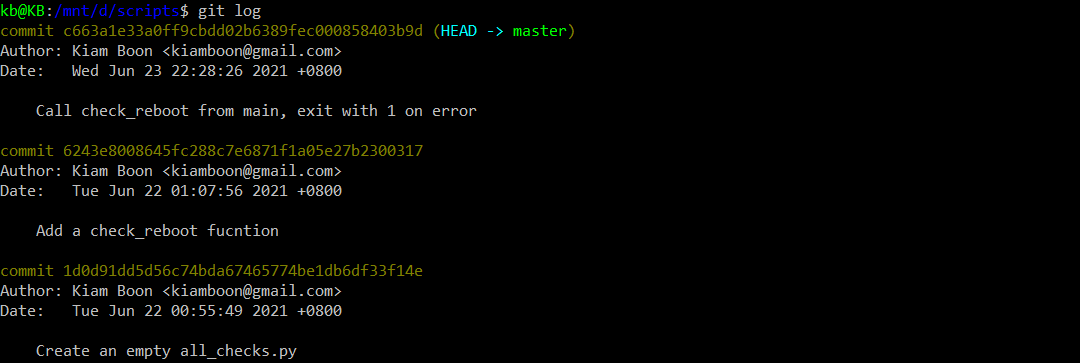
Keep in mind that when we use the **-m** shortcut, we can only write short messages and can't use the best practices regarding commit descriptions that we talked about earlier. So it's best reserved for truly small changes that don't require extra context or explanation, short and sweet.

Heads up, when we use the **-a** shortcut, we skip the staging area. Meaning, we can't add any other changes before creating the commit. So we need to be sure that we’ve already included everything we want to include in that commit.

*If we're making a small change and want to skip the staging step, these two flags we need to add to the git commit command:*

* *The -a flag lets us add and commit in the same step.*
* *The -m flag allows us to directly add the commit message to the command.*

In the end, using a shortcut like **-a** is just like using the regular commit workflow. The commit will show up in the log along with the message just as usual. Let's check that out.



* See how our latest commit was added to the top of the list of commits and notice how the head indicator has now moved to the latest commit.

We might be wondering, what is this HEAD and where is it heading?

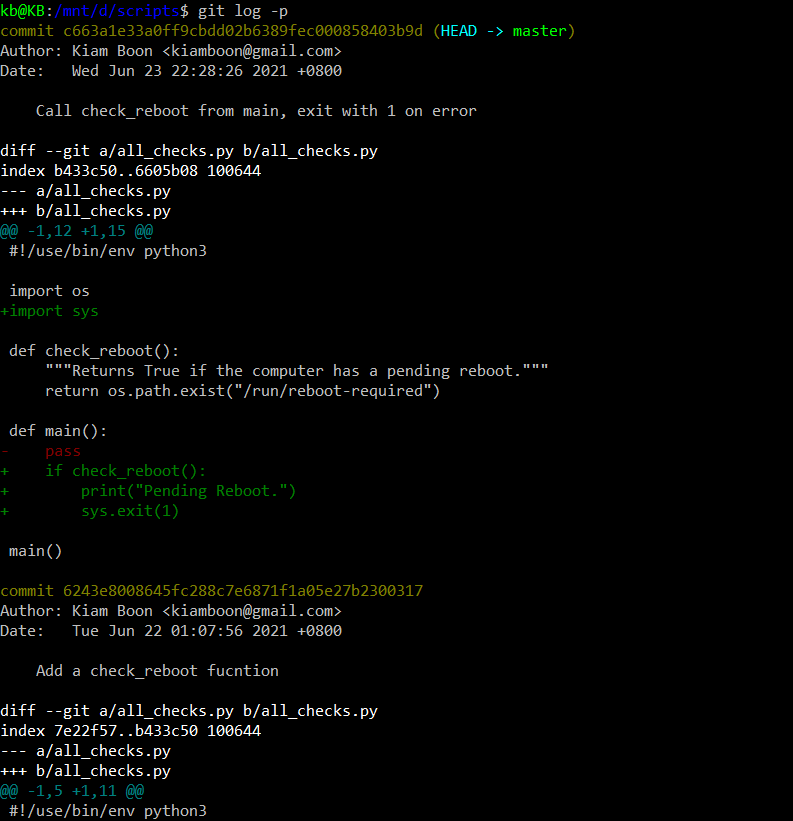
Git uses the **HEAD** alias to represent the currently checked out snapshot of our project. This lets us know what the contents of our working directory should be. In this case, the current snapshot is the latest commit in the project.

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 our place in the project.

Think about it as a bookmark that you can use to keep track of where you are. Even if we have multiple books to read, the bookmark allows us to pick up right where we left off. When we 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 exercises. 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.

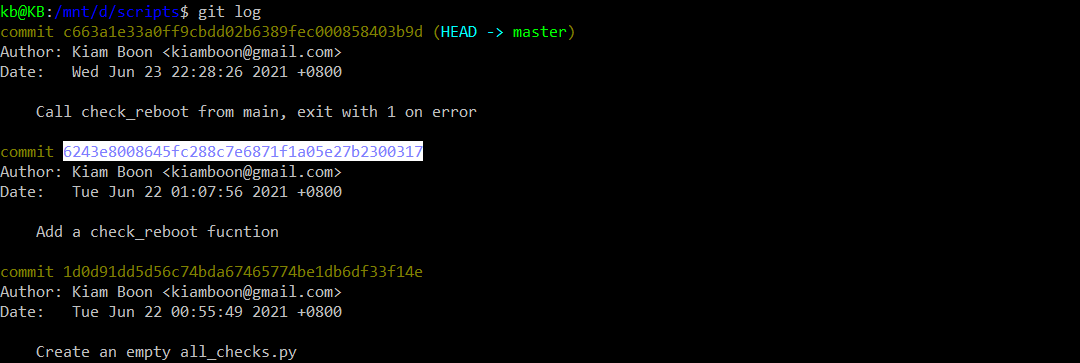
**Getting More about Our Changes**

We've seen how **git log** shows us the list of commits made in the current Git repository. By default, it prints the commit message, the author, and the date of the change. This is useful, but if we're combing through a history of changes in a repo to try and find what caused the latest outage, we'll probably also need to look at the actual lines that changed in each commit. To do this with **git log**, we can use the **-p** flag. The p comes from patch, because using this flag gives us the patch that was created.



* The format **git log -p** is equivalent to the **diff -u** output that we did earlier.
* It shows added lines with plusses and remove lines with dashes. Because the amount of text is now longer than what fits on our screen, Git automatically uses a paging tool that allows us to scroll using page up, page down, and the arrow keys.
* We still have one commit below the other, but now each commit takes up a different amount of space, depending on how many lines were added or removed in that commit.
* Using this option, we can quickly see what changes were made to the files in our repository. This can be especially useful if we're trying to track down a change that recently broke our tools. To exit, press q.

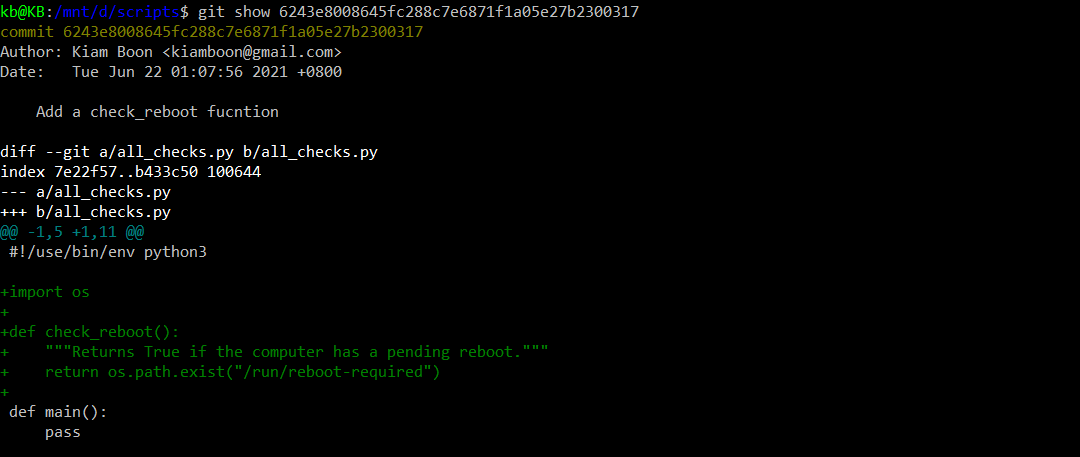
If we don't want to scroll down until we find the commit that we're actually interested in, another option is to use the **git show** command. This command takes a commit ID as a parameter, and will display the information about the commit and the associated patch. We'll talk more about commit IDs in a later exercise. But for now, remember that this is an identifier that we see next to the word commit in the log.



* First listing the current commits by using **git log** in the repo to display the committed log
* Copy the commit ID (committed memory location)

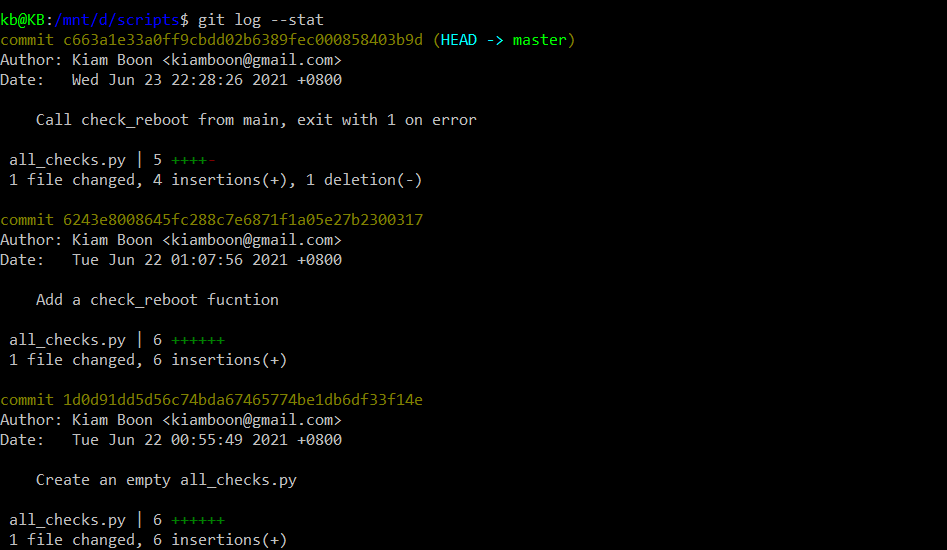


* And then calling **git** **show** by pasting the copied commit ID followed by the **git** **show**



* It will show the same as **git log -p** (part of, rather than showing all like **git log -p**)
* **git show** will show information about the commit and its associated patch

We've shown how we can use git log for listing commits, and **git log -p** for showing the associated patches. Another interesting flag for **git log** is the **--stat** flag. This will cause git log to show some stats about the changes in the commit, like which files were changed and how many lines were added or removed. Let's try it with our repo.



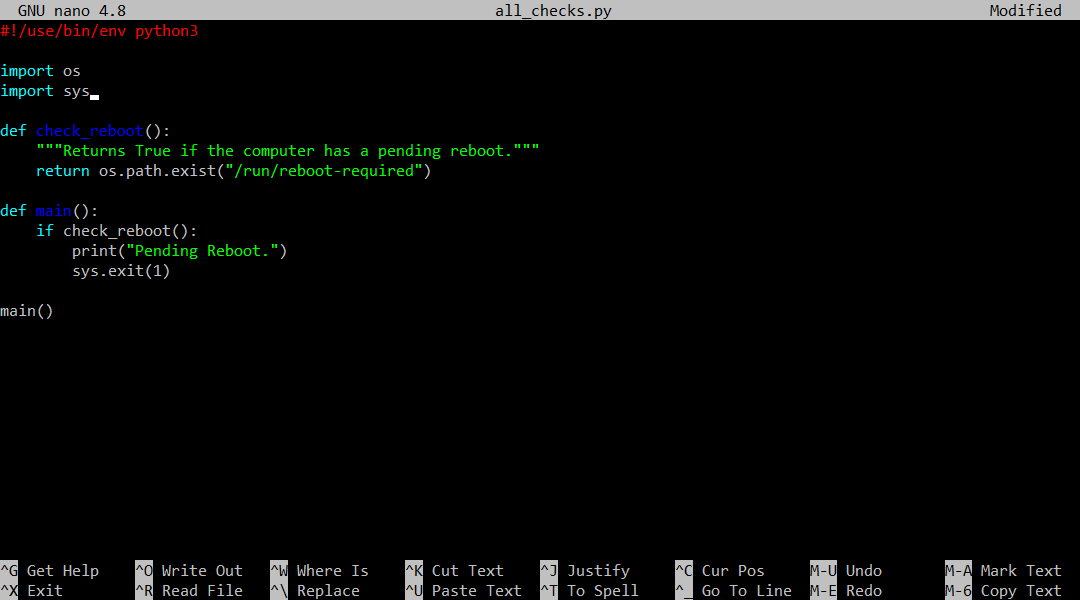
* **git log --stat**
* There are a bunch of other options to **git log** (This tutorial won’t cover all)

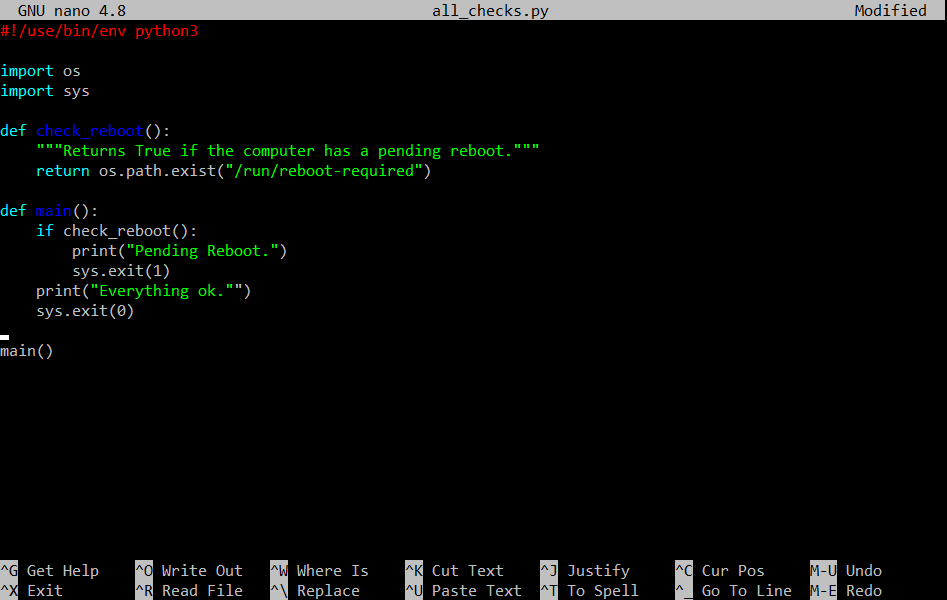
Now, what about changes that haven't been committed yet? Until now, whenever we've made changes to our files, we've either added them to the staging area with **git add** and committed them with **git commit**, or committed them directly using **git commit -a**. This works fine, but it means we have to know exactly which changes we've made.

Sometimes it can take a while until we're ready to commit. We call these “commitment issues”. But imagine we've been working on adding a new complex feature to a script and it requires thorough testing.

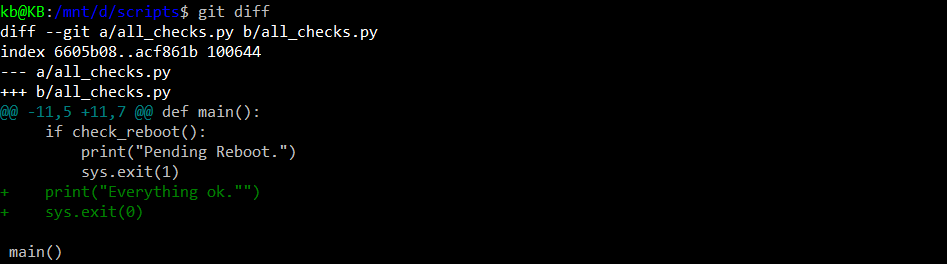
Before committing it, we need to make sure that it works correctly. Check that all the test cases are covered and so on and so on. So while doing this we find bugs in our code that we need to fix. It's only natural that by the time we get to the commit step we don't really remember everything you changed. To help us keep track git gives us the **git** **diff** command.







* Let's make a new change to our script and then try this command out. We'll add another message to the user to say that everything is okay when the check is successful and then exit with 0 instead of 1.
* Okay, we've made the change. Let's now save it and check out what git diff shows us.



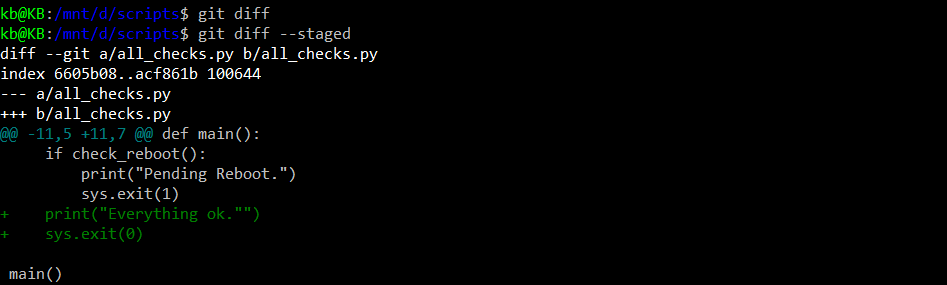
* Again, this format is equivalent to the **diff -u** output that we did in an earlier exercise.
* In this case, we see that the only change is the extra lines (in green plus sign) that we've added.



* If our change was bigger and included several files, we could pass a file by parameter to see the differences relevant to that specific file instead of all the files at the same time. Something else we can do to review changes before adding them is to use the **git add** **-p** command.
* When we use this -p flag, git will show us the change being added and ask us if we want to stage it or not. This way we can detect if there's any changes that we don't want to commit.
* We've staged our change and it's now ready to be committed.



* If we call **git diff** again, it won't show any differences, since **git diff** shows only unstaged changes by default.



* Instead, we can call **git diff --staged** to see the changes that are staged but not committed. With this command, we can see the actual stage changes before we call **git commit**.

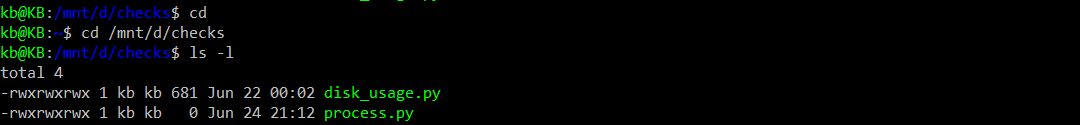


* Let's commit these changes now so that they aren't pending anymore.
* We'll say that we've added a message when everything's okay.

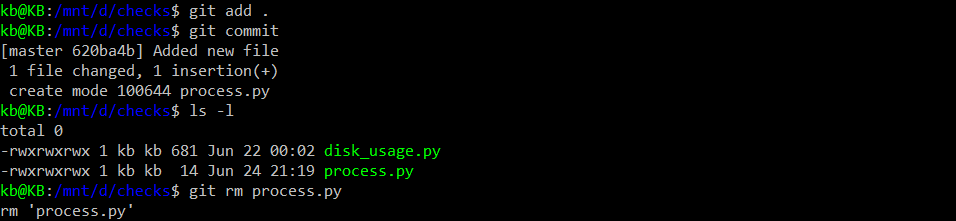
**Deleting and Renaming Files**

Let's say that we’ve decided to clean up some old scripts and want to remove them from our repository. Or we’ve done some refactoring, which makes that particular file obsolete. We can remove files from our repository with the **git rm** command, which will stop the file from being tracked by git and remove it from the git directory or the working tree and from the index.

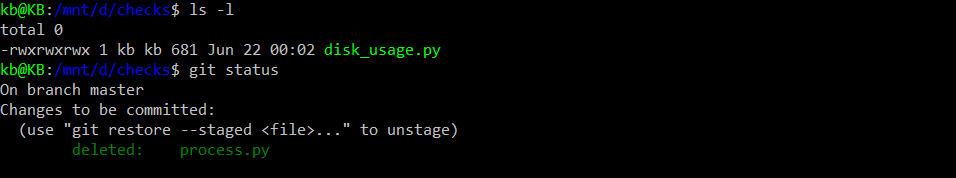
File removals go through the same general workflow that we've seen. So we’ll need to write a commit message as to why we’ve deleted them. Let's try this out in our checks repository that contains a file we decided we actually don't want.



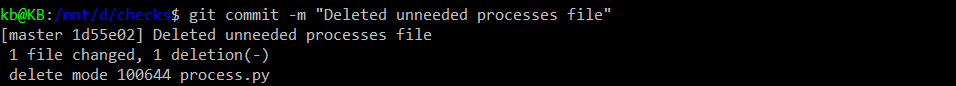
Well first look the contents of the directory with **ls -l**



* Then delete the file with **git rm**
* Note: Because I added the process.py in the last minute, so we need to add our untracked file(s) to which we want to delete by using **git add .** or **git add <filename>**, and then **git commit** or **git commit -m <msg>** the file. We also can delete the file by forcedly using command **git rm -f <filename>**

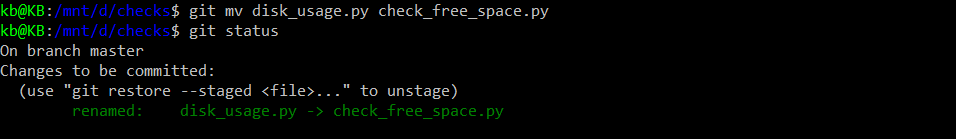


* Then check the contents with **ls** **-l** again, and finally check the status with **git status**.

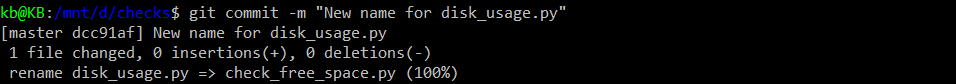


* So, we see that by calling **git rm**, the file was deleted from the directory, and the change was also staged to be committed in our next commit. Let's do that now, by calling **git commit -m <msg>** and indicating that we've deleted the unneeded file.
* As usual, we get a bunch of stats when we do the commit. Check out all the deletions that reported. The 1 deletion(-) means the line(s) in the file that are no longer there, for this case it was only 1 line. And it states the file itself was deleted.

What if we have a file that isn't accurately named? This can happen. For example, if we start writing a script that we thought would only do one thing, and then expands to cover more use cases. Or conversely, if we named our script thinking that it would be very generic, but it ends up being more specific. We can use the **git mv** command to rename files in the repository. Let's rename our existing script disk\_usage.py to check\_free\_space.py and check what **git** **status** has to say about that.



* The status shows us that the file was renamed and clearly displays the old and new names.

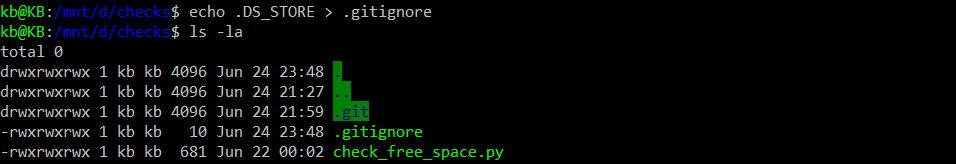


* As with the previous example, the change is staged, but not committed. Let's commit it by calling **git commit** once again.

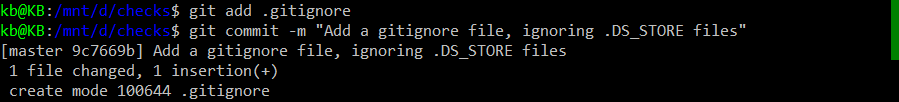
The **git mv** command works in a similar way to the mv command on Linux and so can be used for both moving and renaming. If our repository included more directories in it, we can use the same **git mv** command to move files between directories.

As we can probably tell from our examples, the output of **git status** is a super useful tool to help us know what's up with our files. It shows us which files have tracked or untracked changes, and which files were added, modified, deleted or renamed. It's important that the output of these commands stays relevant to what we're doing. If we have a long list of untracked files, we might lose an important change in the noise. If there are files that get automatically generated by our scripts, or our operating system generates artefacts 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. Inside this file, we'll specify rules to tell git which files to skip for the current repo.

For example, if we're working on an OSX computer, we'll probably want to ignore the .DS\_STORE file, which is automatically generated by the operating system.



* To do this, we'll create a **.gitignore** file containing the name of this file.
* Remember that the dot prefix (eg: .git is folder that is transparent in Windows) in a Unix-like file system indicates that the file or directory is hidden and won't show up when we do the normal directory listing. That's why we have to use **ls- la** to see all files.



* We've added a gitignore file to our repo but we haven't committed it yet. This file needs to get tracked just like the rest of the files in the repo. Let's add it now.

# Advanced Git Cheat Sheet

|  |  |
| --- | --- |
| **Command** | **Explanation & Link** |
| git commit -a | [Stages files automatically](https://git-scm.com/docs/git-commit#Documentation/git-commit.txt---all) |
| git log -p | [Produces patch text](https://git-scm.com/docs/git-log#_generating_patch_text_with_p) |
| git show | [Shows various objects](https://git-scm.com/docs/git-show) |
| git diff | [Is similar to the Linux `diff` command, and can show the differences in various commits](https://git-scm.com/docs/git-diff) |
| git diff --staged | [An alias to --cached, this will show all staged files compared to the named commit](https://git-scm.com/docs/git-diff) |
| git add -p | [Allows a user to interactively review patches to add to the current commit](https://git-scm.com/docs/git-add) |
| git mv | [Similar to the Linux `mv` command, this moves a file](https://git-scm.com/docs/git-mv) |
| git rm | [Similar to the Linux `rm` command, this deletes, or removes a file](https://git-scm.com/docs/git-rm) |

There are many useful git cheatsheets online as well. Please take some time to research and study a few, such as [this one](https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf).

.gitignore files

.gitignore files are used to tell the git tool to intentionally ignore some files in a given Git repository. For example, this can be useful for configuration files or metadata files that a user may not want to check into the master branch. Check out more at: <https://git-scm.com/docs/gitignore>.

A few common examples of file patterns to exclude can be found [here](https://gist.github.com/octocat/9257657).