Git and GitHub

**V**ersion **C**ontrol **S**ystem in the context of managing configuration files and maintaining source code of programs and scripts

**VCS** (**Ex.** Git tool) will let us keep the track of the changes that we make to our **scripts/program-code** or **configuration files** and **any other kind of documents** that need to be tracked.

**VCS** keeps the historical copies of project. This will let you see **the progress of the changes over time**, what the project was look like before and easily go back to that previous if the latest changes were wrong and even **helps you understand why the change was made** and **who made the changes.**

**Technical Terms:**

**Commit:** A command to make edits to multiple files and treat that collection of edits as a single change.

**Commit files:** A stage where the changes made to files are safely stored in a snapshot in the Git directory

**Commit message:** A summary and description with contextual information on the parts of the code or configuration of the commit change.

***git commit*** - command that stores the current contents of the index in a new commit along with the commit message. It captures a snapshot of the project's currently staged changes.

**Diff:** A command to find the differences between two files

**DNS zone file:** A configuration file that specifies the mappings between IP addresses and host names in your network

**Git:** A free open source version control system available for installation on Unix based platforms, Windows and macOS

**Git directory:** A database for a Git project that stores the changes and the change history

**Git log:** A log that displays commit messages

**Git staging area:** A file maintained by Git that contains all the information about what files and changes are going to go into the next commit

**Modified files:** A stage where changes have been made to a file, but they have not been stored or committed

**Patch:** A command that can detect that there were changes made to the file and will do its best to apply the changes

**Repository:** An organization system of files that contain separate software projects

**Source Control Management (SCM):** A tool similar to VCS to store source code

**Stage files:** A stage where the changes to files are ready to be committed

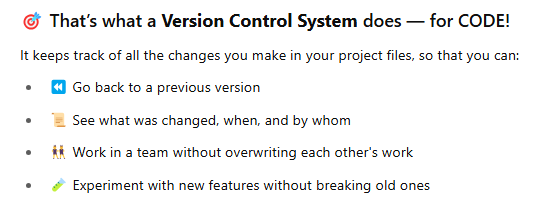
**staging area** - where files are gathered and prepared for git before committing them.

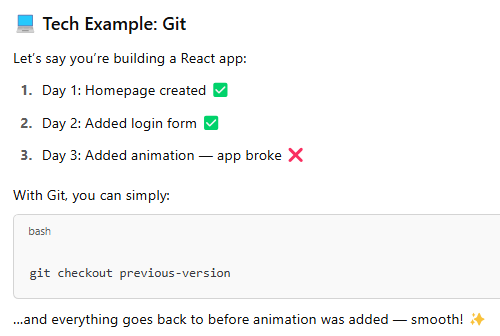
**Tracked:** A file’s changes are recorded

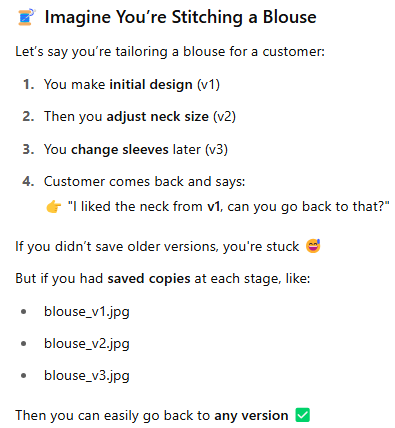
**Untracked:** A file’s changes are not recorded

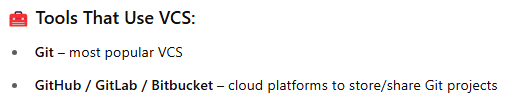
**Version control systems (VCS):** A tool to safely test code before releasing it, allow multiple people collaborate on the same coding projects together, and stores the history of that code and configuration.

how to use Git to keep a history of your code changes:



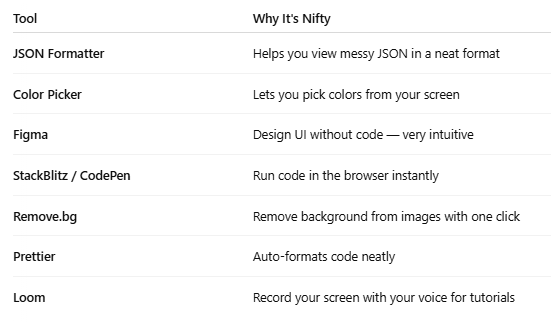




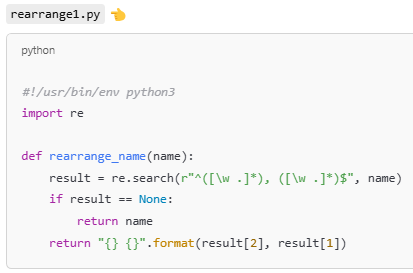
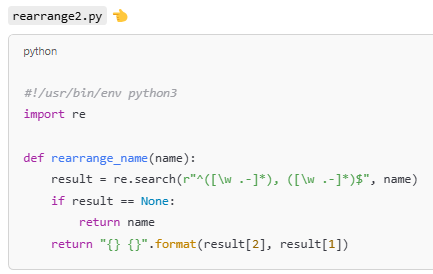


**Diffing Files:**

Imagine you have two copies of code and you wanted to see what the difference was between them.

There are some smart/nifty tools that will make our work faster: 

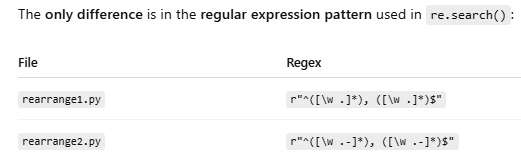
Here we use *diff* command line tool to take 2 files or 2 directories to show the difference between them.

**Example-1:** Assume you have 2 python files:

**NOTE: ** This is a **shebang line**. It tells the system to run this script using Python 3.

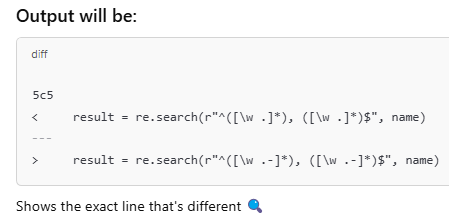
**Command used to output the code content of a file:**

user@ubuntu:~$ cat rearrange1.py  - file lo rasina code ni output ga isthundi.

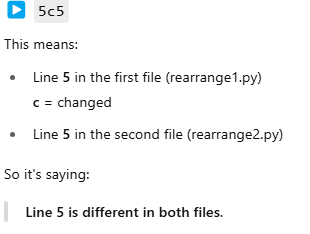
****

**Command to run:**

****



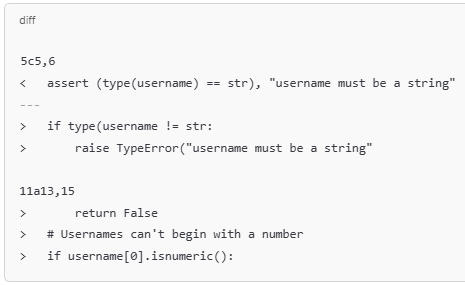
Output Explanation:



**Example-2**: Above *rearrange.py* file has only one line changed, Let take other example where couple of changes are made to code.

**Assume the output we got after running the command:**

**d**iff validations1.py validations2.py



Output explanation:

* 5c5,6 – Line 5 changed to line 5 and 6
* assert (type(username) == str), "username must be a string" – line that got changed.

This is a **quick one-line check** using assert. If username is not a string, it throws an AssertionError.

* 11a13,15 - New lines added after line 11

**NOTE:** To have better understanding of changes made in file use **-u** (unified formatter**)** flag in ***diff***command i.e.,

**d**iff -u validations1.py validations2.py

**Popular file comparison tools:**

1. ***diff* :** Compares two files **line by line** and shows the **differences**. Shows which lines changed, added, or deleted

**command:** diff file1.py file2.py

1. ***wdiff*: word diff -** Compares two files but instead of lines, it shows **word-by-word** differences.
2. ***meld* :** A **graphical tool** to compare and merge files or folders. Side-by-side color-coded comparison
3. ***KDiff3* :** Another **graphical tool** for comparing and merging **up to 3 files**. Supports **3-way merge** — useful for resolving merge conflicts in Git.
4. ***Vimdiff* :** Used inside the **Vim editor** (Terminal-based) to compare files side-by-side.

**Example-3:** How you can explain remote team-mate about bug you found by *.diff* file using *diff* command.

If your teammate encounters a bug after changing the old file and creating a new one, you can use this command:

diff old\_file.py new\_file.py > change.diff

This command will compare the old file and new file, and save only the changes into a file called ***change.diff***.

Now, You can send this ***change.diff*** file to another teammate, so they can see what was changed and fix the bug easily.

1. **Assume a python file which has bug:**

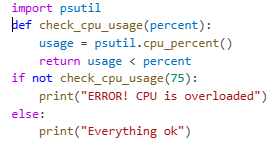
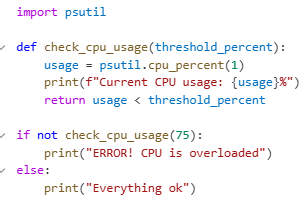


Fig: **CPU\_usage.py** - code that checks the CPU usage

**Code explanation:**

1. uses psutil.cpu\_percent() to get the current CPU usage.
2. If the usage is **higher** than a threshold (in this case, 75%), it prints an error.
3. Without passing interval to psutil.cpu\_percent(), it just gives the immediate value (which is often 0), not a sampled average.
4. Add a time interval i.e., usage = psutil.cpu\_percent(1) to get a more accurate CPU usage
5. Make a copy of **CPU\_usage.py** and fix a bug in it and save with **CPU\_usage\_fixed.py** .

****

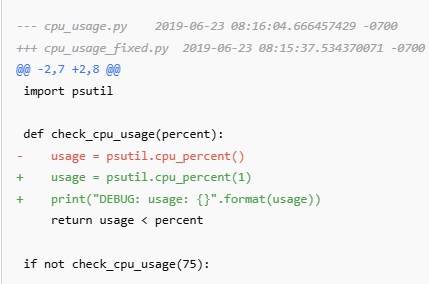
**Fig: CPU\_usage\_fixed.py**

1. **Run the command which compares the files and save those changed lines to *diff file:***

Diff -u CPU\_usage.py CPU\_usage\_fixed.py > CPU\_usage.diff

1. **Use *cat* Ubuntu command to see the file content:**

cat CPU\_usage.diff

Expected Output: 

* + --- and +++ lines: Show the names and timestamps of the old and new files.
  + @@ -2,7 +2,8 @@: Shows the line range that was changed.
  + - (minus): Lines that were removed.
  + + (plus): Lines that were added.

1. Now, use **patch command** to read the code differences (like what was removed and added) contain in cpu\_usage.diff by using **<** (input redirection operator) symbol and then updates cpu\_usage.py by applying those changes.

patch cpu\_usage.py < cpu\_usage.diff

* + **Patch**: A Unix/Linux command used to apply changes (called patches) to files.
  + **cpu\_usage.py**: The original file that you want to apply the patch to.
  + **< cpu\_usage.diff**: This uses the < symbol to redirect input. It takes the contents of the file on the **right** (cpu\_usage.diff) and sends it as **input** to the command on the **left** (patch cpu\_usage.py).

1. Use Linux *cat* Command to see the final content in cpu\_usage.py:

Cat cpu\_usage.py

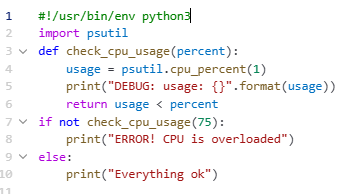


Fig: above ***cat*** command output

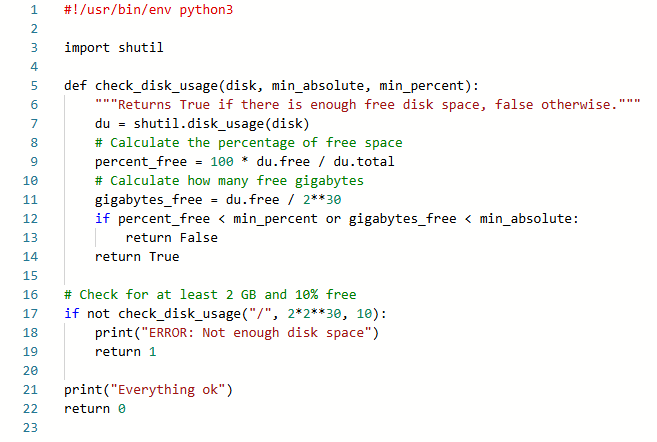
**Example-4:** Another example of applying *diff -u* and *patch* command on 2 python files:

**Step-1:** Always keep copy of original file

****

Above ***cp*** command copies disk\_usage.py to a new file called disk\_usage\_original.py and disk\_usage\_fixed.py.

**Step-2:**



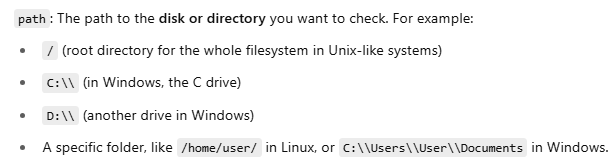
**Fig**: **disk\_usage\_fixed.py has few bugs**

**Code Explanation:**

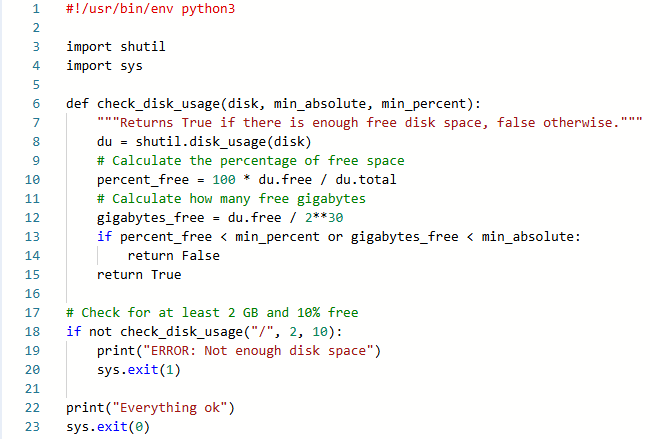
* + shutil module, which provides disk usage information (and other file operations).
  + if not check\_disk\_usage("/", 2\*2\*\*30, 10) - This checks disk usage on the root directory **/**.
  + **Shutil** has a method **disk\_usage ()** - It returns a named tuple with **three values in bytes.**

**shutil.disk\_usage(path to disk/directory in string)** → (total, used, free)

* du.total → total disk space (in bytes),
* du.used → used disk space (in bytes),
* du.free → free disk space (in bytes)



* + Python throws a **SyntaxError** if return is used outside of a function.
  + You already converted du.free to **gigabytes** in the function. But you're passing 2 \* 2\*\*30, which is in **bytes**, not GB. So this causes a **unit mismatch** and incorrect results.
  + sys.exit(1) - Proper way to exit a script with an error code

**Step-3: Fig:** Bug free code of **disk\_usage\_fixed.py**. Assume bugs found after running above code using “./disk\_usage\_fixed.py “ command and are fixed.

**Step-4:** use ‘*diff -u’* command to compare files and save those comparisons to *.diff* file

diff -u disk\_usage\_original.py disk\_usage\_fixed.py > disk\_usage.diff

**Step-5**: Use *cat* command to check what contain in disk\_usage.diff file.

cat disk\_usage.diff

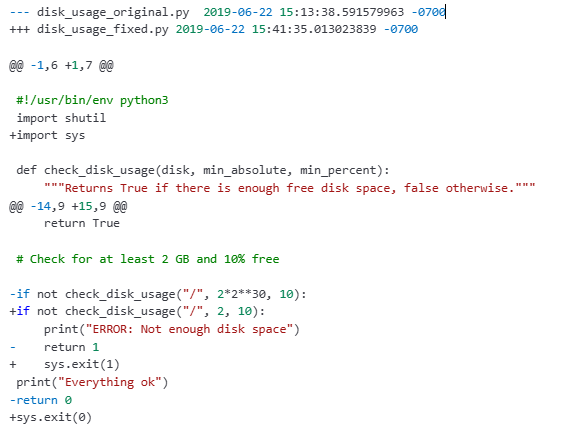


Fig: *cat* command output

Step-6: *patch* command to apply the comparisons to original file

patch disk\_usage.py < disk\_usage.diff – **o/p:** patching file disk\_usage.py

**Step-7:** Finally run the original patched file.

./disk\_usage.py

**Example-5:** performing *diff command* on two **.text** files

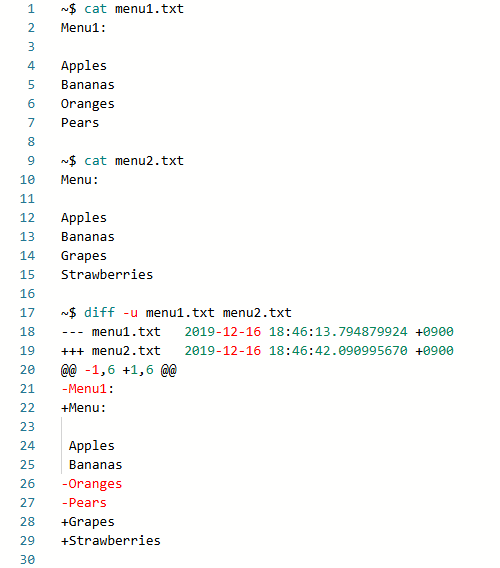


Fig: Using “ ***diff -u*** “command to compare two files, line by line, and have the differing lines compared side-by-side in the same output.

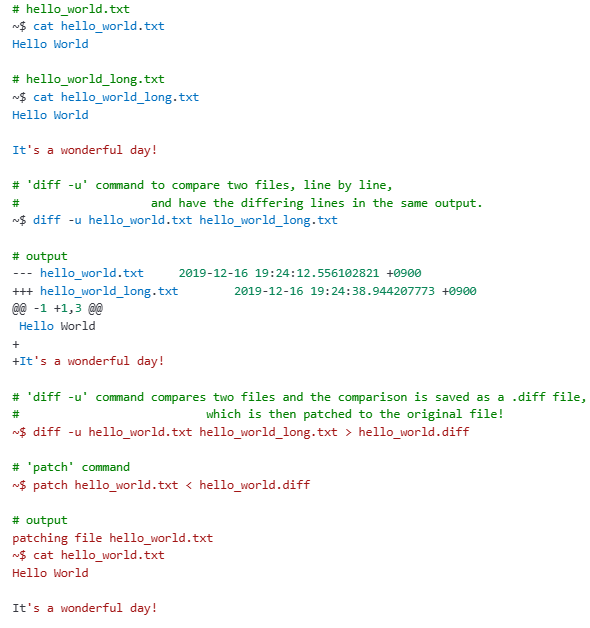


Fig: compares two files and the comparison is saved as a .diff file, which is then patched to the original file! The ***patch*** command is useful for applying file differences.

**Check the following links for other interesting patch and diff commands:**

* + <http://man7.org/linux/man-pages/man1/diff.1.html>
  + <http://man7.org/linux/man-pages/man1/patch.1.html>

**VCS in practice:**

Till We have seen, how we can directly/manually use tools ( like *diff* )to extract differences between versions of files and apply those changes back to the original file.

Now, instead of directly using *diff* tool let use **Version Control System**( which keeps the track of changes we make to our files)

Term ‘***Commit’*** – used to make edits to multiple files and treat that collection of edits as single change

**VCS usage in simple Example:**

1. Let say you store **DNS Zone file** for your company in VCS.
2. **DNS Zone file** is a **configuration file** that specifies mapping between IP addresses and host names in your network.
3. Best practice, when you update the **Zone file** content always use good explanatory message that why/when you are adding new Ip address & host names to the **Zone file** while **committing** it to VCS
4. If anything breaks after you add a new entry to file, you rely on the VCS to revert back the old file before changes are made, this makes the problem solving easier.

**Git – Popular version control system**

1. Every person contributing to a repository has full copy of the repository on their own development machines.
2. Collaborators can **share** and **pull in** changes that other have made.
3. Git can work as **standalone program**, **as a server** and **as a client** .

**Git** ane tool ki **centralized server** compulsory kaadu.

Ante, mana system lone Git **independent ga** work cheyyagaladu — without depending on any main server.

* Git ni **standalone program** laga use cheyyachu.(local ga files track cheyyadam, commits cheyyadam etc).You can use Git on a single machine without having n/w connection.
* Git ni **server** laga kuda setup cheyyachu to host repositories (inkokar ki code share cheyyadam kosam).
* Git ni **client** laga kuda use cheyyachu to access the repository from local/another machine (server nunchi code pull cheyyadam, push cheyyadam kosam).

**Chocolatey** - a **package manager** for **Windows**.

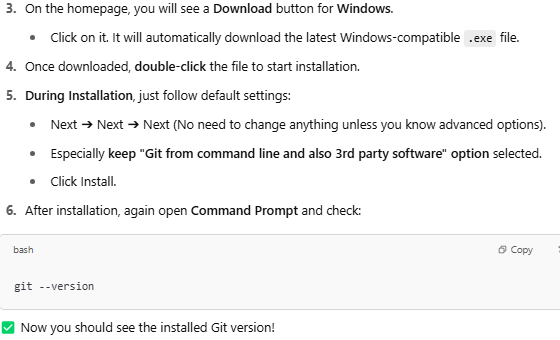
* It helps you **install**, **update**, **configure**, and **uninstall** software **easily** using **simple commands** (just like apt in Linux or brew in Mac).
* 👉 Instead of manually searching and downloading **.exe** files from websites,  
  you can install software with **just one command**!
* Chocolatey has a huge **repository** (library) of software packages.

**Downloading GIT:**

**Git:** git is VCS/tool used to track changes in source/project code. Git can be used via GUI tools or command-line

**Git bash:** It is a command-line application, with Bash shell-environment/terminal to use Git commands and partial Unix/Linux commands on windows.

1. Open windows command line
2. Check whether system already installed git using command 👉 **git - -version**
3. If not installed you can either
   1. install from official git website 👉 <https://git-scm.com/>

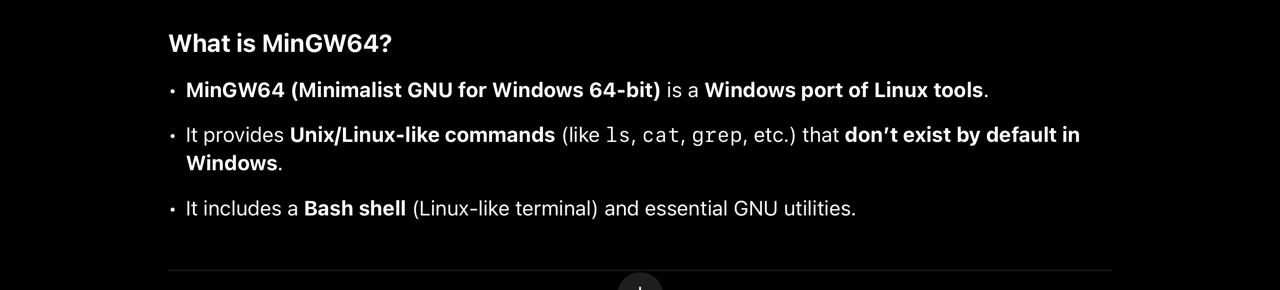


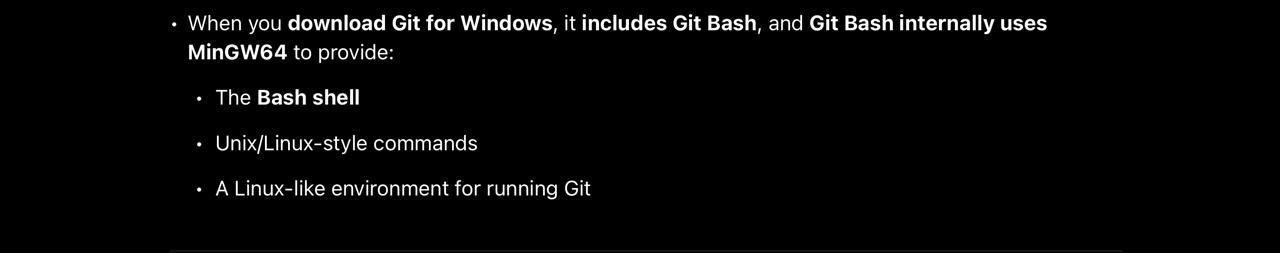
* 1. open👉 [**https://gitforwindows.org/**](https://gitforwindows.org/)
     1. Click on **Download** button which install **Git software package** including bunch of Unix-like utilities and other tools.
     2. Keep go will all the default configurations.
     3. As per ‘**Coursera’** tutorial the default text editor changed to **Atom** - an open source text editor which comes with built-in support for **Git** and **GitHub**.
     4. At Adjusting path environment i.e., Choosing How would you like to Git from what command line (i.e., **Git bash**, **Command prompt**, **PowerShell**, **as well as any 3rd party software**). Go with ‘**Recommended’** option.
     5. At Choosing HTTPS transport backend i.e., Choosing which SSL/TLS library would you like Git to use for HTTP connections? Choose default **OpenSSL library** as we will only be interacting with **GitHub**.

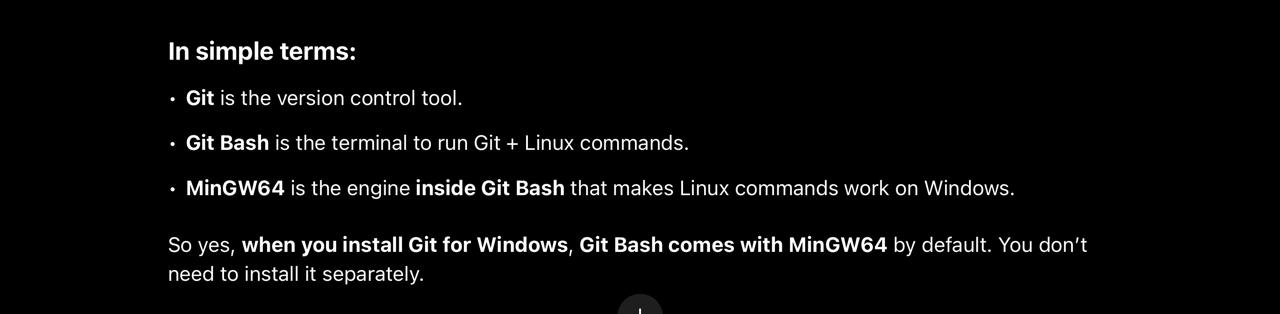
**NOTE:** Choose **native windows secure channel** library if you need to interact with your company’s internal systems

* + 1. At configuring the Line ending conversions – The characters used to indicate the end of line are different between windows, Linux, mac OS. Go with default.
    2. **At configuring terminal emulator to use with Git Bash - Git Bash** can run **some** basic Linux/Unix commands, but **not all** **real Linux system commands**.

**Git Bash -** **provides a small set of Linux-like commands**, mainly through **MinGW and bash.exe**







1. After installing Git, you get **Git Bash** installed along with a **GUI** to interact with Git.
   1. It’s a special terminal similar to Linux style where we can run all the **Git commands**.
   2. You can find it by searching **Git Bash** in Start menu.
   3. You can use **Git Bash** or **Command Prompt** — both will support Git commands!

**Start Git with Basic configuration:**

1. We need to tell git who we are? Which helps Git to understand who made changes to file.

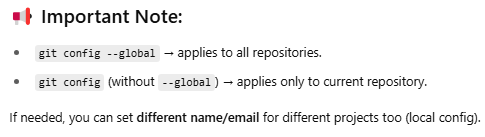
git config --global user.email [me@example.com](mailto:me@example.com)

git config --global user.name "My name"

* + **git config** → command is used to set the values to identify who made changes to Git repositories.
  + **--global** → apply this setting for **entire system** (for **all** Git projects on your local computer).
  + **user.email** → setting the **email address** you want Git to associate with your commits.

**User.name** your name that will appear in your Git commits.

👉 **Meaning:** Whenever you commit code, Git will **tag** that commit with this **email**.



1. Command to creates a **new folder.**



**mkdir** → "Make Directory" command.

Creates a **new folder** called checks.

1. Command to movie inside **“Checks“** folder.



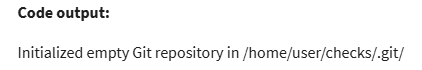
1. Command to initialize **New Empty Git repository** (You start **from scratch** — no code.) in the current folder **“Checks”**



It creates a hidden **.git** folder inside **checks/**.

From now on, Git will **start tracking changes** inside this folder.

Common when **starting new projects**.

****

1. **To copy an existing Git repository** from a remote (like GitHub) to your computer.

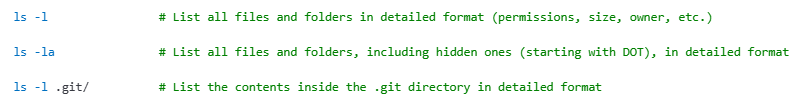


You **download** an existing project — with all code + Git history.

➡️ This will **create a folder**, **download all files**, and **setup Git** in one command.

Common when **working on existing projects**.

1. ***Linux list* commands.**



1. **Git directory** and **working tree**?
   1. **git directory** contains all the changes and their history and the **working tree** contains current versions of the files.
   2. **git directory** acts as a database for all changes tracked in Git and the working tree act as sandbox where we can edit current versions of the files.
2. **Copy a project files to current “checks” folder to make it as a our working tree.**

**cp ../disk\_usage.py .**

**NOTE:** disk\_usage.py is our earlier python program

1. Use “***ls -l”*** command to list all files and folders.
2. **Always Remember:** Now we must make Git to track our project files saved in current folder “checks” using following command.

git add file or .

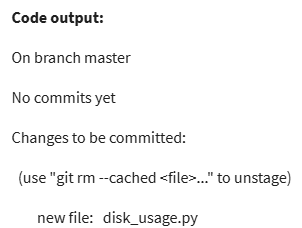


git add disk\_usage.py

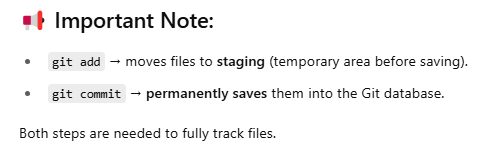
**NOTE:** **Staging area – A** Temporary place to collect files for next commit. think of it like a **shopping cart.**  First you **select** items (files you changed). After you're ready, you **checkout** (commit) everything together.

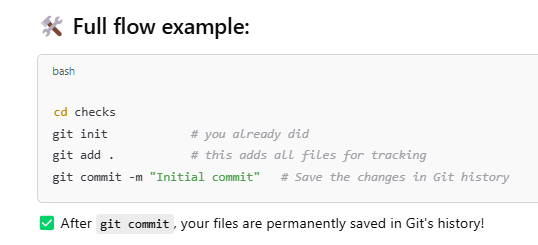
1. Command to see the changes that are staged for *commit.*

git status



1. Command to commit



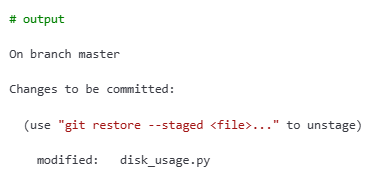
**NOTE**:

**Any Git project** will consist of **3 sections** – **Git directory**, **Working tree**, and **the staging area.**

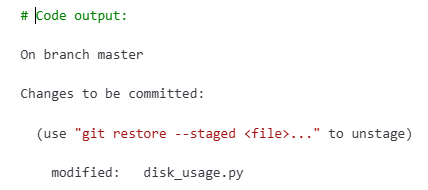
1. **Git directory** contains the history of all files and changes.
2. **Working tree** contains the current state of project code, including any changes that we have made.
3. **Staging area** contains the changes that have been marked to be included in the next commit.
4. **Remember: Git does observe changes made to files even if you don’t add them to the staging area.**
   1. **How Git Observes Changes:**
      * Git always watches the working directory (your project folder).
      * When you modify a tracked file (a file Git already knows about), Git detects it immediately — even if you don’t run *git add*.
      * But until you do *git add*, the changes are not staged for commit.

**Steps demonstrate how git Tracking files / observing changes made to file in working tree:**

1. user@ubuntu:~$ **cd checks**
2. user@ubuntu:~/checks$ **git status**
3. user@ubuntu:~/checks$ **ls -l**
4. user@ubuntu:~/checks$ **atom disk\_usage.py** - command *atom* opens file in Atom editor / you can directly open file from working directory in VS code editor.
5. Now made small changes to code in **disk\_usage.py Ex**: change content in print().
6. Save the changes to disk\_usage.py file
7. user@ubuntu:~/checks$ **git status**



1. user@ubuntu:~/checks$ **git add disk\_usage.py**
2. user@ubuntu:~/checks$ **git status**

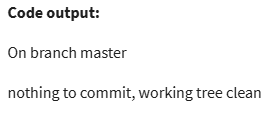


1. user@ubuntu:~/checks$ git commit -m 'Add periods to the end of sentences.'

**NOTE:**

* + By this we committed our stage changes. This creates a new snapshot and stores them in Database that lives in Git directory.
  + If we made *git commit* with empty message then commit will be aborted.

1. user@ubuntu:~/checks$ **git status**



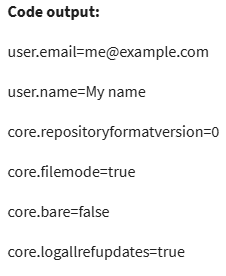
**Git workflow with basic example:**

1. mkdir scripts
2. cd scripts
3. git init - **Output:** Initialized empty Git repository in /home/user/scripts/.git/

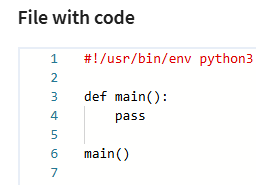
**NOTE:** Git initializes a new empty repository in “Scripts” folder. A hidden folder called “.git/” is created inside your “Scripts” folder i.e., **D:\scripts\.git\. Now** your “**Scripts**” folder is officially a Git repository (local repo).

1. git config -l - git config = View/set Git configuration settings.

- -l = List all the current configuration values



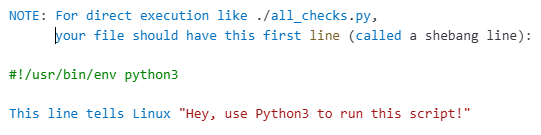
1. Create a **all\_checks.py** python file inside “**Scripts**” folder.



1. Command to run python file from Git terminal.

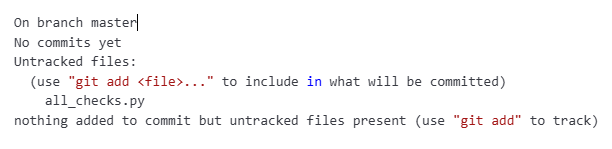
chmod +x all\_checks.py - this allows to run **.py** file successfully without

denying the permissions.

chmod - Change file **permissions** (command)  
+x - Add execute permission. 

1. git status - Here we just created .py file in “**scripts**” repo which

still be untracked by Git. We see this by running *git status* command.



1. git add all\_checks.py - This *git add* command makes “**Scripts**” repo untracked

files to tracked and add those stage area by Git.

**NOTE:** It will also change a file in the modified state to stage state(files that are

ready to commit).

1. git commit - This opens a default editor which asks to provide commit message.

Ex. Add a Created an empty python file **all\_checks.py**

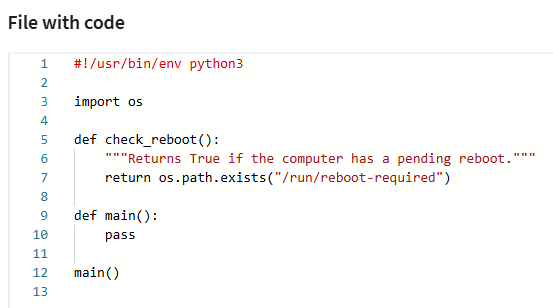
This also takes the snapshot of the project code stored in local Git repo.

Every time we commit changes, it takes another snapshot and stored in

Git repo.

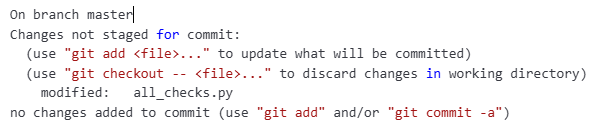
**NOTE:** If we made *git commit* with empty message then commit will be aborted.

1. Changes made to python file

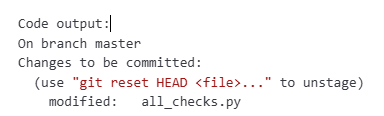


1. git status - By this we are checking what are staged after changes made to .py file.

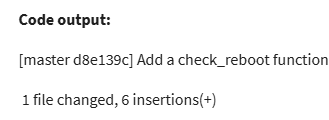
We see changes are not staged.



1. git add all\_checks.py  - By this we stage our changes made to .py file.
2. git status

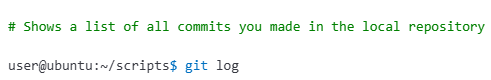


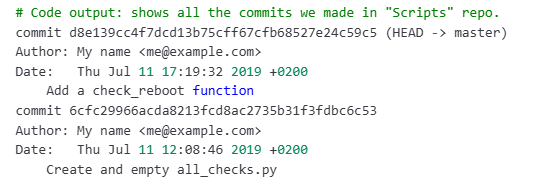
git commit -m 'Add a check reboot function'



1. **Writing meaningful commit messages:**

A **Commit message** should look like - A short description of the change (up to 50 characters), followed by a blank line, then one or more paragraphs giving more details of the change. Each line in the detailed paragraph should be less than 72 characters





**Here what does HEAD - > master means:**

**Git uses** the terms

* + Head which points to latest commit in the currently checked-out snapshot of your project.
  + Master is the **default branch** Git creates when you initialize a new repo.

Which means master ane branch lo manam project ni develop chesthu untam.

**HEAD - > master** means “you are currently on the master branch, and HEAD is pointing to its latest commit”

**Qwiklabs:**

An online platform powered by google. It gives you **temporary cloud computers** (Linux, Windows, Google Cloud servers) where you can practice **hands-on labs**.

Qwiklabs lo **real cloud environments** untayi means you are working with **actual tools** in the **real cloud**, just like you'd do in a job.

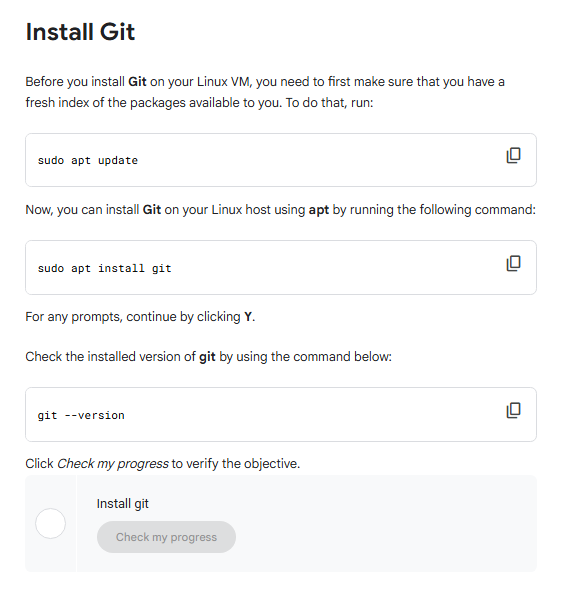
**Git Practice Scenario:**

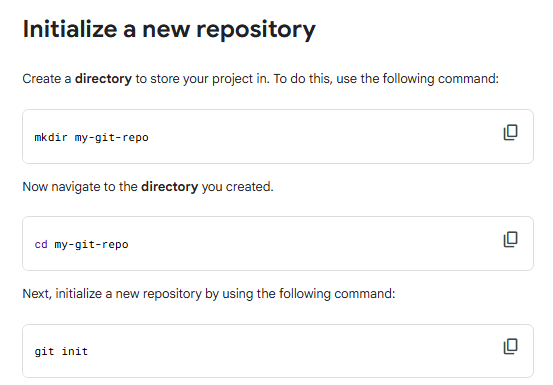
you are a project lead in an IT company. You and your team are working on a huge project, which consists of multiple functionalities and modules. This project is evolving over time and so your team is expecting a lot of code revisions. In this lab, you'll learn how to use a distributed version control system called Git. You'll also discover how to connect to a VM instance, install Git, and configure your Git user information. Next, you'll create a local Git repository, add a file to the repository, and do some basic operations like adding a file, editing files, and making commits.

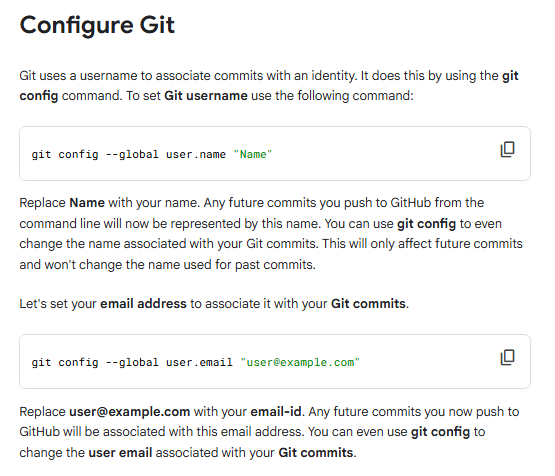
**What you'll do**

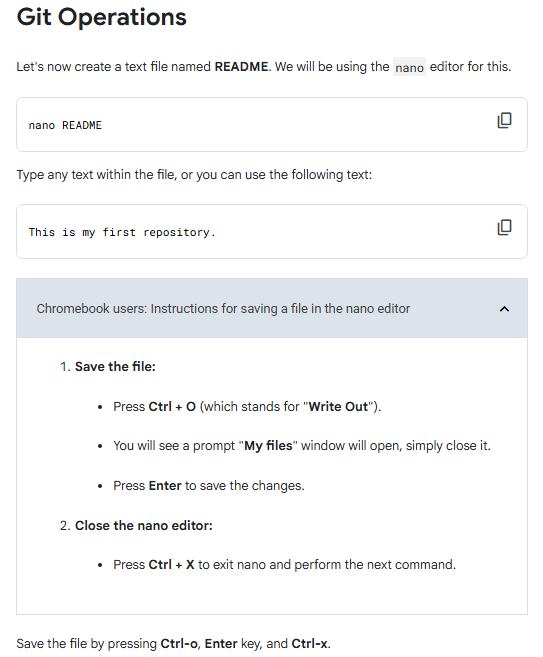
* Create a git repository.
* Add files to this repository
* Edit the files
* Commit the changes to the repository.

**NOTE:** For compatibility **Coursera** Recomamded us to use Google chrome/Mozilla Firefox browser while accessing the labs





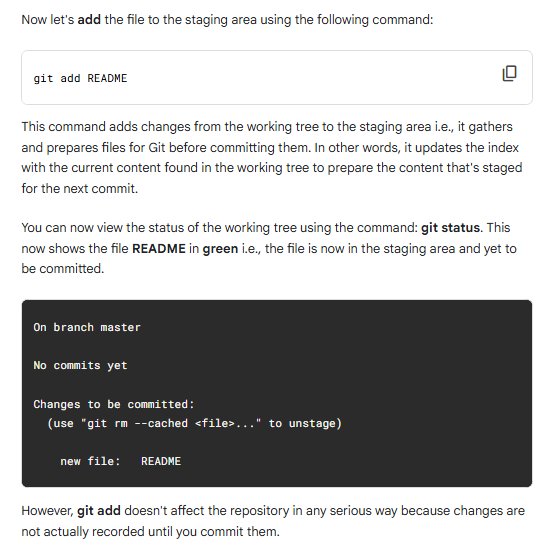
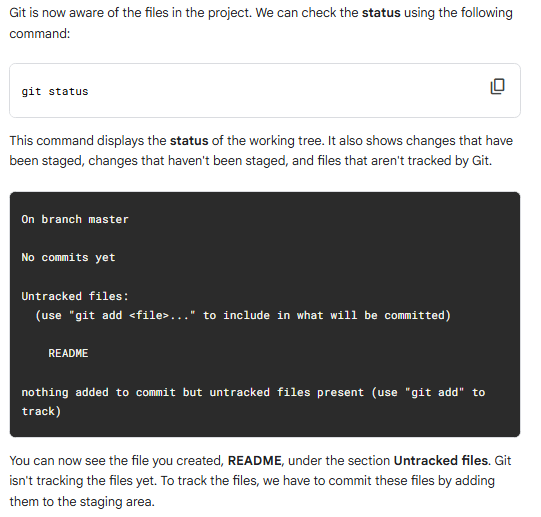


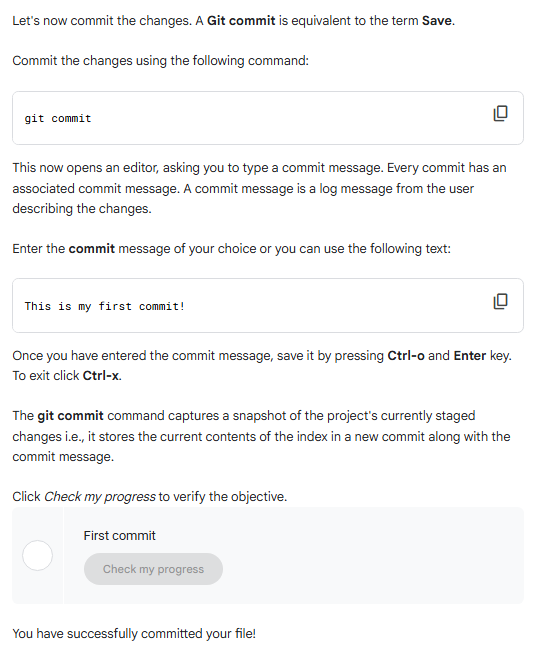


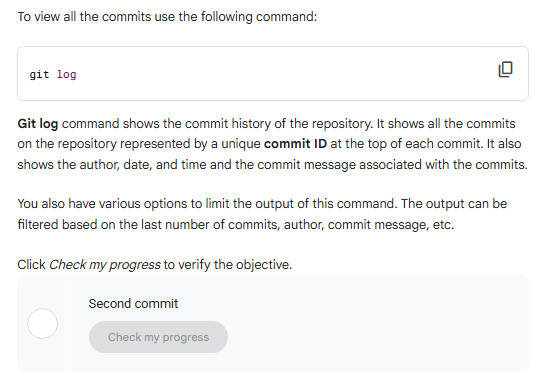
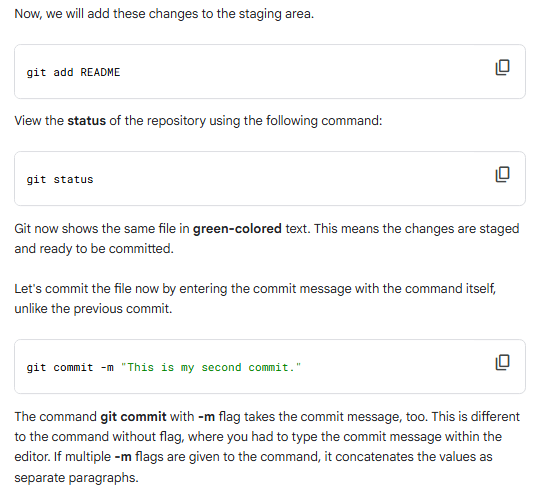
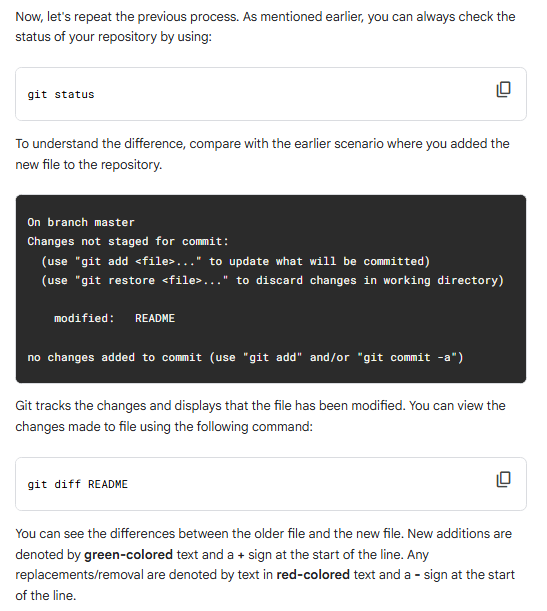
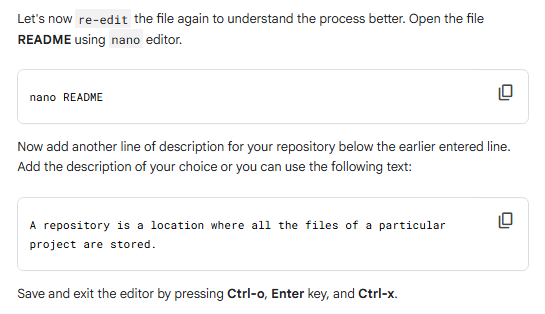
**NOTE:**

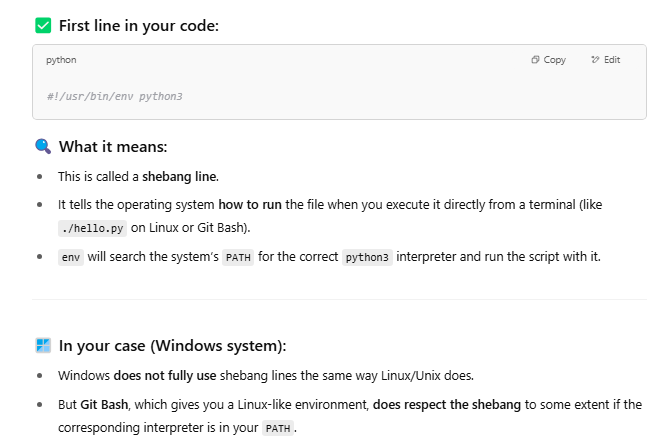
**If you exit before saving file, then** *nano* will ask **“Save modified buffer? (Answering 'No' will DISCARD changes.)"**

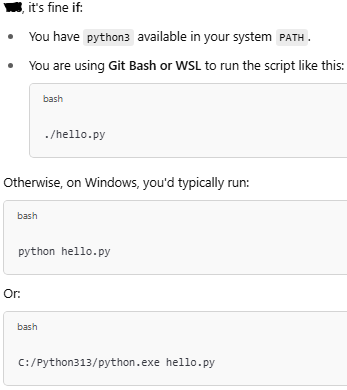
1. Press **Y** → to save your changes.
2. After that, it will take you to prompt: **“File Name to Write: README”**
3. Then press **Enter** to confirm the filename.
4. Now *nano* will save the file and exit.

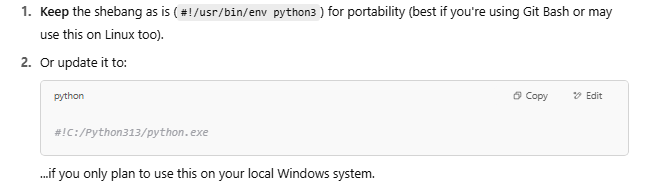
 











**How to skip the staging area:**

**Use ‘*git commit -a*’** command **- to automatically stages all tracked, modified files** before committing. This saves you from having to manually run git add for each modified file to get tracked.

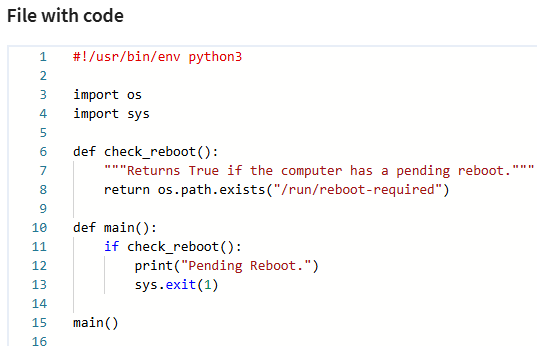
**NOTE:**  Command **only works** with already tracked files. It will **not** stage new (untracked) files — for those, you must use git add first.

**Practice Example:** Run the following commands in Git Bash.

1. **cd scripts**
2. **atom all\_checks.py** - opens python file in ‘atom’ text editor

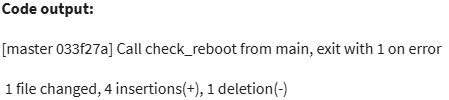
**NOTE:** ***code all\_checks.py*** - opens python file in VS Code if it is the Git Bash

default editor.

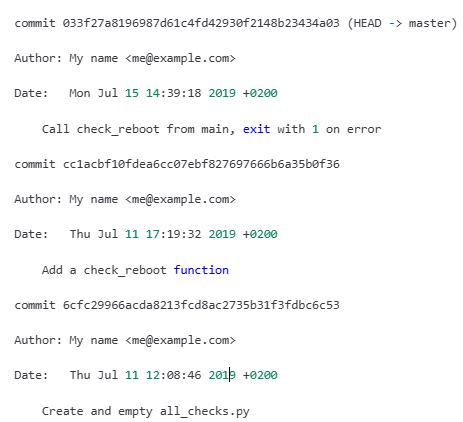


**Fig:** *all\_checks.py*

1. git commit -a -m "Call check\_reboot from main, exit with 1 on error"



1. git log



**Fig:** *git log* command output.

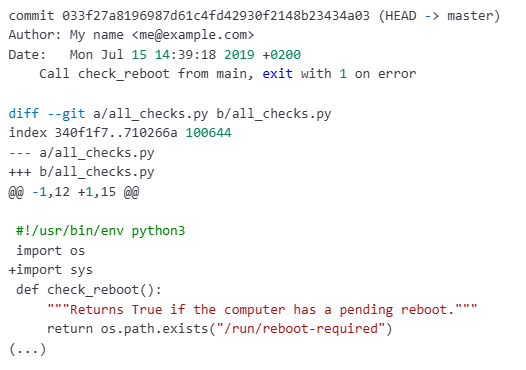
**Getting more information using ‘*git log*’ command:**

**Run the following commands:**

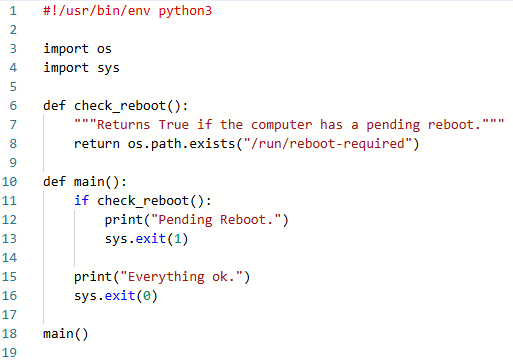
1. *git log*➡️ shows the list of commits we made on the current Git repository. By default, it prints **commit messages**, **the author**, **the date of the change**.
2. Usegit log --stat command➡️ lists all commits along with which files were changed and how many lines were added or removed
3. Run *git log* ➡️ *where you see e*ach commit has unique identifier we can use that ID to see.

Copy the commit\_ID and,

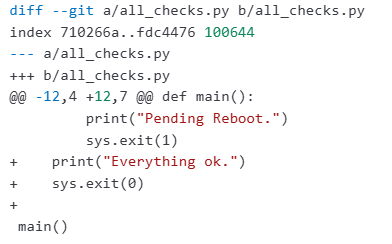
1. Use *git show commit\_ID ­*command➡️ Instead of logging all commits with associated patches, we can use *git show commit\_ID* to see specific commit message with associated patch.
2. *git log -p* ➡️ Use *-p* patch flag tells Git to show all commits along with exact code that changed \_to in the files along with commit messages.



1. *code all\_checks.py* ➡️ open python file in VS Code editor and do the following changes

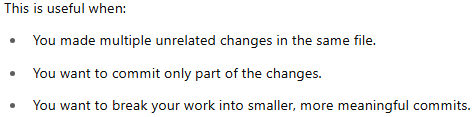


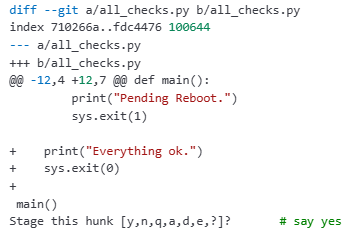
1. Run *git diff or git diff file\_name* ➡️ display the exact code that we changed in ***all\_checks.py*** file



**NOTE:** *git diff* shows only changes that are not yet staged

1. Run *git add -p* ➡️ allows you to **interactively review and stage changes hunk by hunk** (a hunk is a contiguous block of changes in a file), rather than staging the whole file.

**

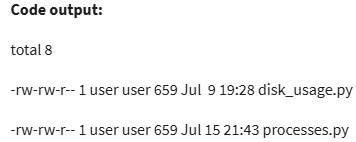
**

*Fig:* When you run *git add -p*, Git shows each hunk like this and ask whether to stage this hunk?

1. Run *git diff --staged* (to see changes that are staged but not committed) instead *git diff* (as it shows changes that are not staged)
2. git commit -m 'Add a message when everything is ok'

**Deleting and renaming files:**

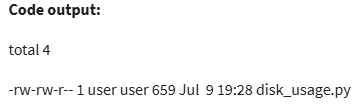
1. Open repository in Git Bash ➡️ *cd checks/*
2. ls -l ➡️ see what is inside the directory



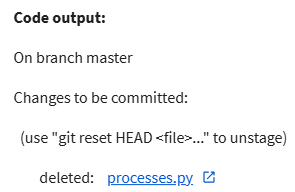
1. git rm process.py ➡️ command to remove file from repository



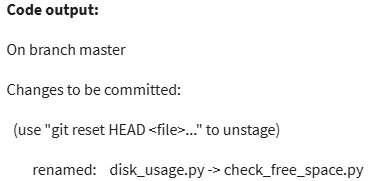
1. ls -l



1. git status

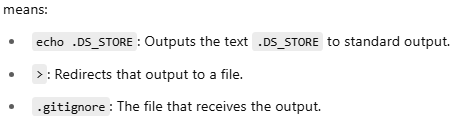


1. git commit -m 'Delete unneeded processes file'
2. git mv disk\_usage.py check\_free\_space.py ➡️ used to rename the file /moving the file from one repo to another repo.
3. git status



**NOTE:** Here the change is stage, but not committed.

1. git commit -m 'New name for disk\_usage.py'
2. echo .DS\_STORE > .gitignore ➡️

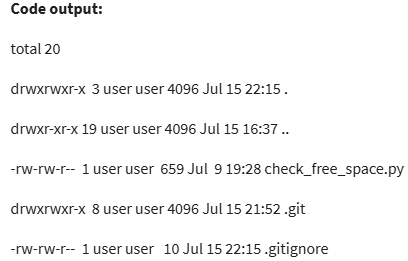


***.DS\_Store*** is a hidden file created by macOS in directories to store folder view settings. You typically don’t want it tracked in Git repositories.

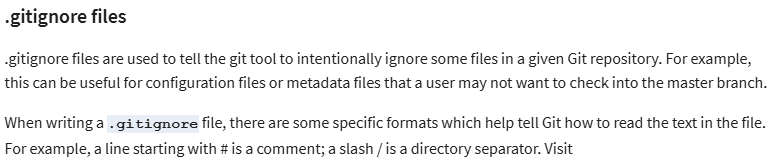
**Important:** If a *.gitignore* file already exists, this command will **overwrite it**, potentially deleting other ignore rules.

To **append** instead of overwrite, **use:** *echo .DS\_STORE >> .gitignore*

1. ls -la ➡️ Linux command used to list all along with **hidden DOT files** in a directory.



1. git add .gitignore ➡️ .gitignore also needs to tracked like all files in repo.



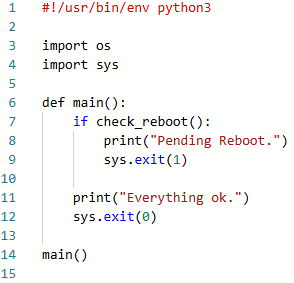
**To see more examples** ➡️ https://git-scm.com/docs/gitignore

1. git commit -m 'Add a gitignore file, ignoring .DS\_STORE files'

**How to Revert our changes before committing:** Assume you have made bunch of changes to file that you don’t want to keep them hence you want to revert back to earlier committed state.

1. cd scripts
2. atom all\_checks.py

**NOTE:** Code afterRemoved **check\_reboot()** method from above python file.



1. *./all\_checks.py*➡️ run python file form Git Bash.

It raise **NameError:** name 'check\_reboot' is not defined

1. git status

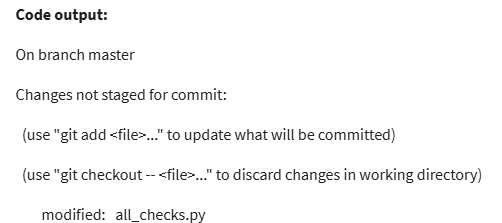
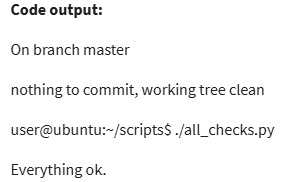


Fig: We see file as modified and changes not staged

1. git checkout all\_checks.py ➡️ command helps to check out the original file from the latest stored snapshot.
2. git status



1. ./all\_checks.py



1. ./all\_checks.py > output.txt ➡️ Creating a temporary file to store the output of

python File.

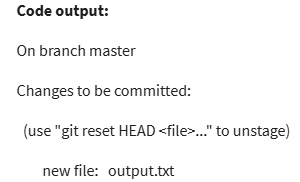
1. *git add \** ➡️ Stage all currently untracked or modified files in your working directory so

that they are ready to be committed.

***NOTE:*** *\** ➡️ Tells Git to stage all files and folders in the current directory expect the

hidden ones(like .gitignore, .env etc)

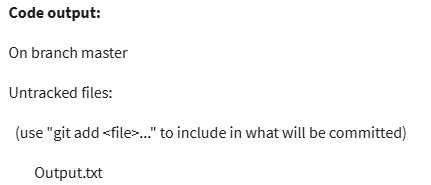
1. *git status*



1. *git reset HEAD output.txt* ➡️ used to **unstage** a file that was previously added to the staging area using *git add* command.

**NOTE:** *Git reset -p* ➡️ Used to interactively unstage parts(hunks) of files, rather than the whole file

1. *git status*



1. git commit -m "it should be os.path.exists"