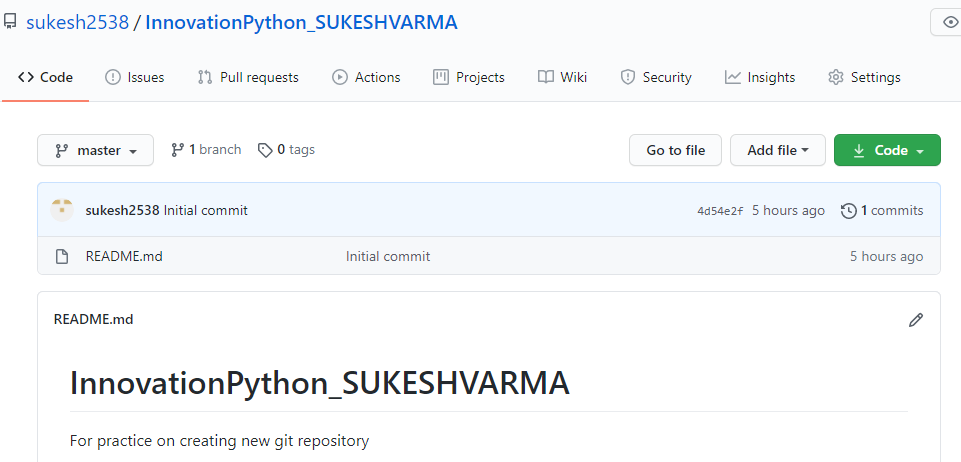
**GIT TASK** **BY ‘SUKESHVARMA’**

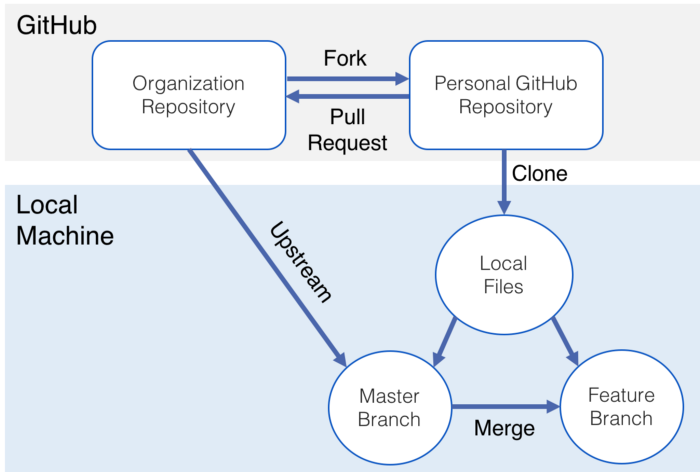
1. **Make a repository on GitHub with the name** **“InnovationPython\_yourname”**



1. **Read about the difference between Git and GitHub?**

Git is a version control system that lets you manage and keep track of your source code history. GitHub is a cloud-based hosting service that lets you manage Git repositories. If you have open-source projects that use Git, then GitHub is designed to help you better manage them.

1. **Read about Git Workflow.**



**Step 1: Set up a Github Organization-**This will be the core code of your project. It is the central working repository on Github that everyone will draw from.

**Step 2: Fork Organization Repository to Your Personal GitHub-**Next fork the central organization repository to your personal GitHub account. This will be your own copy of the project files.

**Step 3: Clone the Repository to Your Local Machine-**Next, create a local copy of these files on your personal machine so you can modify/add to them. You do this by navigating to the directory where you wish to store the project and typing the following into the command line (excluding the ‘<’ and ‘>’ symbols): **git clone<url to the project repo on your personal Github profile>** (\*Be sure the url ends with .git)

This ‘clones’ your GitHub repo to your personal computer. Afterwards, navigate into the cloned directory your just created. Now type: **git branch.**

You should be on the master branch — your main branch. This should be the branch that always has functional code.

**Step 4: Create a Branch for your Working Files**

But what if you want to update the project? Where should you make your changes if the master branch can only have clean, functional code? Let’s make a new branch to create new features, test things out and essentially be the working file. You create a new branch (and move into it) by typing: **git checkout –b <branch name>**

The -b is important, that is what is creating the new branch. I will be calling this branch that contains our working files ‘feature’ from here on out, but you can call it whatever you like.

**Step 5: Set Remote Repository to the GitHub Organization**

As you continue to work on your project, you will periodically want to incorporate changes from your other team members. Since your GitHub organization is where your functional code is stored, you will want to pull from there. In order to set up this link between your local directory on your own machine and the GitHub organization repository, you will need to set up a ‘remote’ repository (aka a repo that is remote and not located on your local machine).

Do this by typing: **git remote add upstream<url to the organization on GitHub>**

(\*Be sure the url ends with .git)

‘Upstream’ is the conventional name chosen for a remote, but it can be anything you like.

**Step 6: Get Coding!**

Add those new features, tests, and tweaks (you’re making these changes on the feature branch, right?). Once you get to a place where you feel good about the work you’ve done, let’s incorporate it into the project’s code base.

Before we start this process, be sure to add and commit the files you’ve been working on.

**Step 7: Pull the Most Recent Files from the Organization Repo**

All the steps from here on out are to ensure a clean incorporation of your code into the project’s code base. The goal is to isolate any potential code conflicts by adding safeguards to ensure the project’s code is not harmed by anyone’s contribution.

You’ve been working in the feature branch, and you’ve added and committed your changes.

As you’ve been working away, it is safe to assume your team members have incorporated their changes into the project code. Before you add your files/changes to the overall project, you want to make sure your changes don’t conflict with anyone else’s code. To do this, you need to pull down the most recent project files to your local machine so you can make the most up to date comparison.

The GitHub organization repo is where the latest, functioning code base is stored, so we should pull from there. We already set up a link to this repo in step 5, so now is our chance to utilize it!

First, move to the master branch by typing: **git checkout master**

Our master branch is what holds clean, functional code on our local machine, so this is where we pull the most recent changes to. We’re essentially bringing our master branch up to date with what is in the organization repo. We pull the changes from the organization into our master branch by typing: **git pull upstream master**

We are *pulling*changes from our *upstream’s*(which we set up earlier as the organization repo) *master*branch.

**Step 8: Merge the Master Branch Into the Feature Branch**

Now that our master branch is up to date, we want to incorporate all those changes into our feature branch. Let’s go back to our feature branch by typing: **git checkout features**

Now we want to merge all of the code in the master branch into our feature branch. Once we do this, our feature branch contains essentially the entire working code base plus our added features. We make this merge by typing: **git merge master**

This means *merge*the *master*branch into the branch we are currently in (the feature branch).

If there are any conflicts between the code we’ve created in the feature branch and project overall, we will see them now *in the feature branch*.

**Step 9: Push Your Code to your GitHub Repo**

After you’ve merged the master branch into your feature branch (and potentially sorted out any merge conflicts), you’re ready to push your changes up to your GitHub repo. While you’re still in the feature branch, type: **git push origin feature**

This means you are *pushing*your code to the *origin*repository’s (the repository on your GitHub account, since that is where all your cloned files currently on your local machine originated from) *feature* branch.

**Step 10: Make a Pull Request to the Organization Repo**

This is the final step! Now that your changes are in your GitHub repo’s feature branch, make a pull request from that branch to the organization’s repo.

This ‘pull’ is different from the one we made in step 7 to get the most recent code from the organization repo. A ‘pull request’ on GitHub is simply you asking a repository to pull in your changes. Once you make a pull request, your teammates can review the changes you’ve made before they are officially ‘merged’ into the organization repository. Once they are merged in, your other team members will have access to the code you’ve added by pulling it down to their local machines like you did in step 7.

1. **How many types of version control systems are there?**

There are **two types** of version control**: centralized** and **distributed.**

## Centralized version control:

With centralized version control systems, you have a single “central” copy of your project on a server and commit your changes to this central copy. You pull the files that you need, but you never have a full copy of your project locally. Some of the most common version control systems are centralized, including Subversion (SVN) and Perforce.

## Distributed version control:

With distributed version control systems (DVCS), you don't rely on a central server to store all the versions of a project’s files. Instead, you clone a copy of a repository locally so that you have the full history of the project. Two common distributed version control systems are Git and Mercurial.

While you don't have to have a central repository for your files, you may want one "central" place to keep your code so that you can share and collaborate on your project with others. That's where Bitbucket comes in. Keep a copy of your code in a repository on Bitbucket so that you and your teammates can use Git or Mercurial locally and to push and pull code.

1. **Explain Branching concept in Git.**

A **branch** in **Git** is simply a lightweight movable pointer to one of these commits. The default **branch** name in **Git** is master . As you start making commits, you're given a master **branch** that points to the last commit you made.

1. **Explain Forking Workflow in Git**

The **Forking Workflow** is a bit different than other workflows. Because it gives every developer their own server-side repository. The Forking Workflows are more commonly seen in open-source projects.

The main advantage of the Forking Workflow is that contributions can be done without the need for everybody to push to a single central repository. Developers push to their **own server-side repositories**, and only the **project maintainer**can push to the official repository. This allows the maintainer to accept commits from **any developer** without giving them write access to the official codebase.