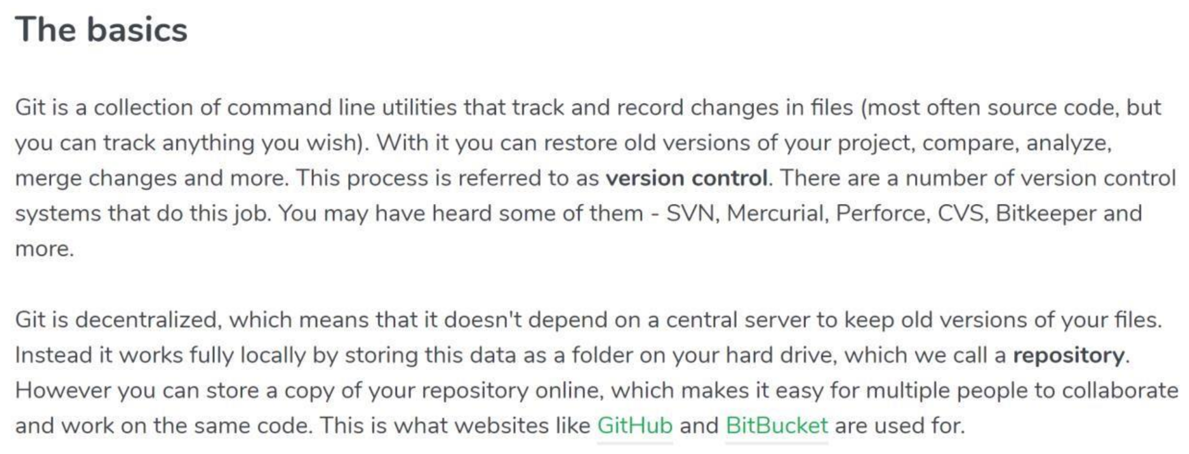
**Developers Institute**

**Python Course**

# GIT AND GITHUB INTRODUCTION

1) Install Git: https://git-scm.com/downloads
2) Read below the explanations on Git and Github (information comes from tutorialzine.com) and do the following **BOLD** exercises

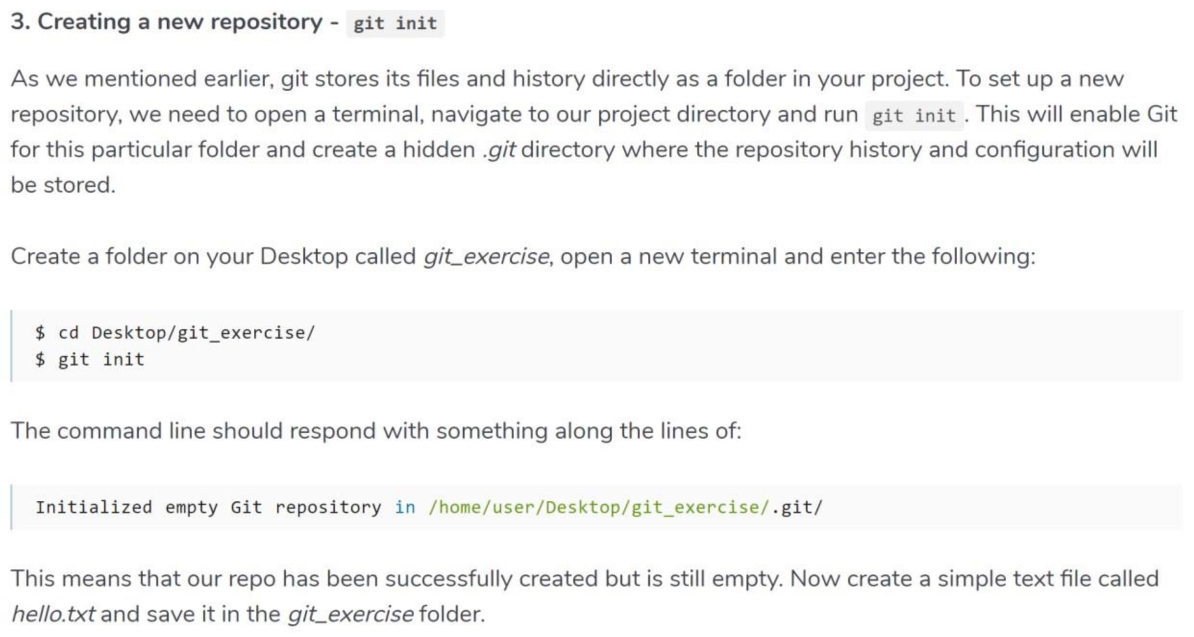
---



## The basics

Git is a collection of command line utilities that track and record changes in files (most often source code, but you can track anything you wish). With it you can restore old versions of your project, compare, analyze, merge changes and more. This process is referred to as **version control**. There are a number of version control systems that do this job. You may have heard some of them - SVN, Mercurial, Perforce, CVS, Bitkeeper and more.

Git is decentralized, which means that it doesn't depend on a central server to keep old versions of your files. Instead it works fully locally by storing this data as a folder on your hard drive, which we call a **repository**. However you can store a copy of your repository online, which makes it easy for multiple people to collaborate and work on the same code. This is what websites like GitHub and BitBucket are used for.



### 3. Creating a new repository - `git init`

As we mentioned earlier, git stores its files and history directly as a folder in your project. To set up a new repository, we need to open a terminal, navigate to our project directory and run `git init`. This will enable Git for this particular folder and create a hidden *.git* directory where the repository history and configuration will be stored.

Create a folder on your Desktop called *git_exercise*, open a new terminal and enter the following:

```
$ cd Desktop/git_exercise/
$ git init
```

The command line should respond with something along the lines of:

```
Initialized empty Git repository in /home/user/Desktop/git_exercise/.git/
```

This means that our repo has been successfully created but is still empty. Now create a simple text file called *hello.txt* and save it in the *git_exercise* folder.

### 4. Checking the status - `git status`

Git status is another must-know command that returns information about the current state of the repository: is everything up to date, what's new, what's changed, and so on. Running `git status` in our newly created repo should return the following:

```
$ git status

On branch master

Initial commit

Untracked files:
  (use "git add ..." to include in what will be committed)

    hello.txt
```

The returned message states that *hello.txt* is untracked. This means that the file is new and Git doesn't know yet if it should keep track of the changes happening to that file or just ignore it. To acknowledge the new file, we need to stage it.

### 5. Staging - `git add`

Git has the concept of a *"staging area"*. You can think of this like a blank canvas, which holds the changes which you would like to commit. It starts out empty, but you can add files to it (or even single lines and parts of files) with the `git add` command, and finally commit everything (create a snapshot) with `git commit`.

In our case we have only one file so let's add that:

```
$ git add hello.txt
```

If we want to add everything in the directory, we can use:

```
$ git add .
```

Checking the status again should return a different response from before.

```
$ git status

On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached ..." to unstage)

    new file:   hello.txt
```

Our file is ready to be commited. The status message also tells us what has changed about the files in the staging area - in this case its *new file*, but it can be *modified* or *deleted*, depending on what has happened to a file since the last `git add`.

### 6. Commiting - `git commit`

A commit represents the state of our repository at a given point in time. It's like a snapshot, which we can go back to and see how thing were when we took it.

To create a new commit we need to have at least one change added to the staging area (we just did that with `git add`) and run the following:

```
$ git commit -m "Initial commit."
```

This will create a new commit with all the changes from the staging area (adding hello.txt). The `-m "Initial commmit"` part is a custom user-written description that summarizes the changes done in that commit. It is considered a good practice to commit often and always write meaningful commit messages.

---

## Git Exercise

1. Create an account on Github
2. Create a folder on your computer called **intro_git**
3. Run `git init` to initialize a local repository.
4. Check the status
4. Add two files:
   - **hello_world.py** that prints "Hello World"
   - **hello_you.py** that prints "The weather is beautiful today"

   You can add these files one by one, or at the same time
5. Commit with the name "first commit"
6. Your repository is only local, to deploy it to a server, we will use Github

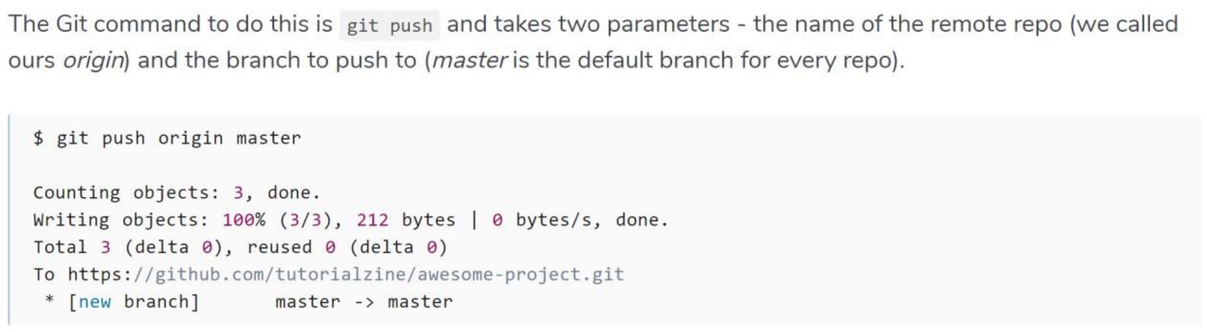
## Github Exercise

5. In your Github account, create a new repository called **intro_DI_github**
6. Follow the instructions and notice that our work is in the Github Repository! Yeahhh 😊

   *Explanation:*

   **git remote add origin URL** : links our local repository with the one on GitHub
   **git push origin master** : transfers our local commits to the server. This is done every time we want to update the remote repository.

---



The Git command to do this is `git push` and takes two parameters - the name of the remote repo (we called ours *origin*) and the branch to push to (*master* is the default branch for every repo).

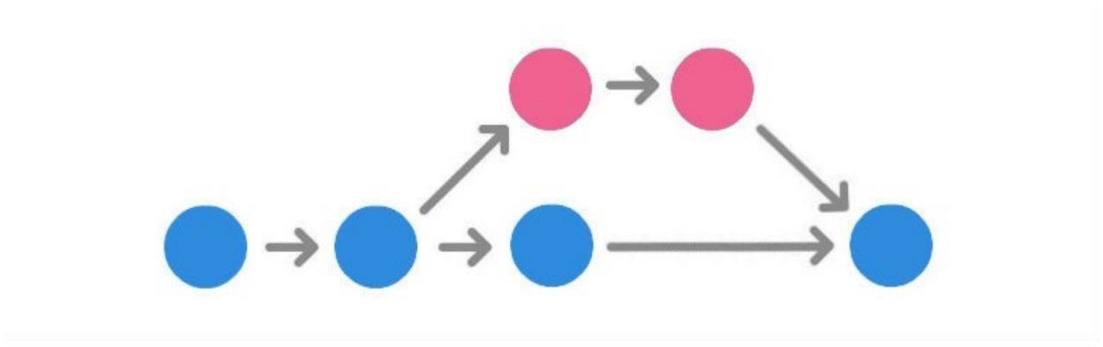
```
$ git push origin master

Counting objects: 3, done.
Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/tutorialzine/awesome-project.git
 * [new branch]      master -> master
```

## Making Changes

1. In the file **hello_you.py**, change "beautiful" by "rainy"
2. Add, Commit and push
3. Look what happened in your Github Repo

---



## Branches

When developing a new feature, it is considered a good practice to work on a copy of the original project, called a *branch*. Branches have their own history and isolate their changes from one another, until you decide to merge them back together. This is done for a couple of reasons:

- An already working, stable version of the code won't be broken.
- Many features can be safely developed at once by different people.
- Developers can work on their own branch, without the risk of their codebase changing due to someone else's work.
- When unsure what's best, multiple versions of the same feature can be developed on separate branches and then compared.

### 1. Creating new branches - `git branch`

The default branch of every repository is called ***master***. To create additional branches use the `git branch <name>` command:

```
$ git branch amazing_new_feature
```

This just creates the new branch, which at this point is exactly the same as our *master*.

### 2. Switching branches - `git checkout`

Now, when we run `git branch`, we will see there are two options available:

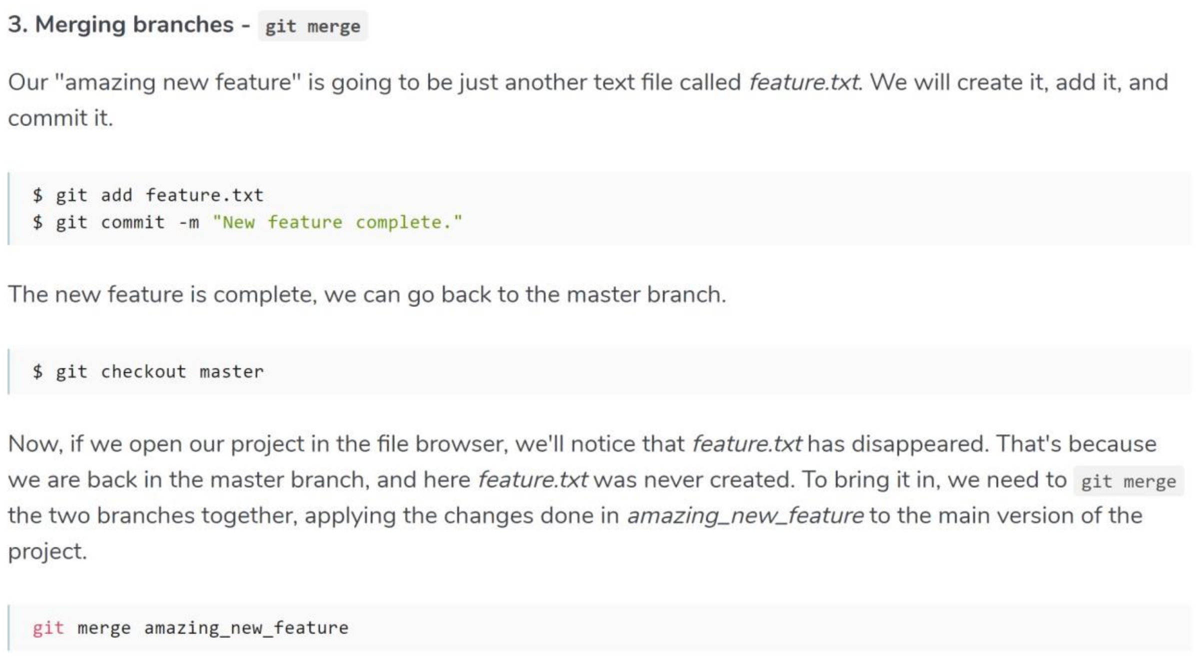
```
$ git branch
  amazing_new_feature
* master
```

Master is the current branch and is marked with an asterisk. However, we want to work on our new amazing features, so we need to switch to the other branch. This is done with the `git checkout` command, expecting one parameter - the branch to switch to.

```
$ git checkout amazing_new_feature
```

## Exercise on Branches

1. Create a new branch called "test"
2. Checkout in the branch *test*
3. In the file **hello_you.py**, print "I love coding and testing"
4. Add, Commit and push to the branch *test*
5. Look what happened in your Github Repo in the branch *test*
6. Look what happened in your Github Repo in the branch *master.* Nothing 😊



### 3. Merging branches - `git merge`

Our "amazing new feature" is going to be just another text file called *feature.txt*. We will create it, add it, and commit it.

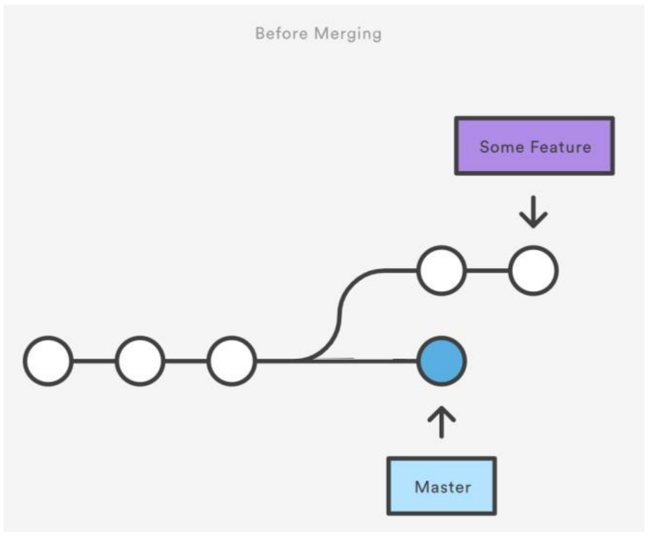
```
$ git add feature.txt
$ git commit -m "New feature complete."
```

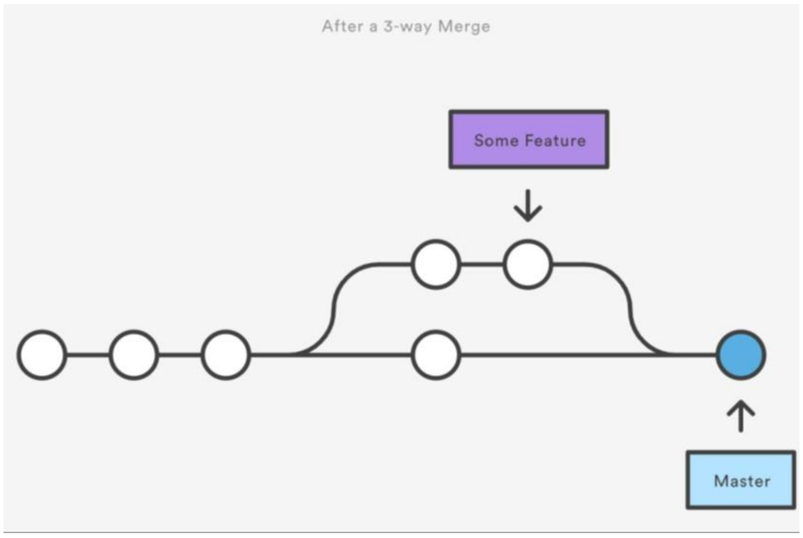
The new feature is complete, we can go back to the master branch.

```
$ git checkout master
```

Now, if we open our project in the file browser, we'll notice that *feature.txt* has disappeared. That's because we are back in the master branch, and here *feature.txt* was never created. To bring it in, we need to `git merge` the two branches together, applying the changes done in *amazing_new_feature* to the main version of the project.

```
git merge amazing_new_feature
```





**Exercise on Merging Branches**

1. Go back to the master branch and merge the branch *test* in it
2. Push the changes into the *master* branch
3. Look what happened in your Github Repo in the branch *master*

The *master* branch is now up to date with the test branch.
BUT they are a few situations where merging is not the best way…:
If the Github repository doesn't belong to us, we don't have the permission to **push** If we work in a team, then if everyone merges at the same time they will be a lot of conflicting lines.
This is why we are going to learn about **Pull Request**

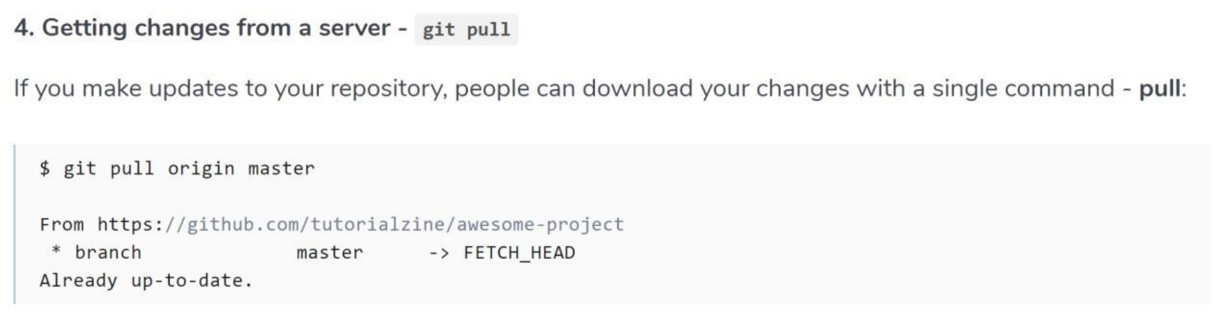
**Exercise on Pull Request**

1. Create another branch called *test2* and checkout
2. In the file **hello_world.py**, print "I am learning how to do a pull request"
3. Add, Commit and push to the branch *test2*
4. Go to Github and click on the button "New Pull Request" next to the branch dropdown menu.
   Set the base branch to the **master** because we want to merge the new branch into the master branch.
   Then select the branch **test2** as the branch which will be merged into the master branch.
   It will compare the two branches. If there are no conflicting lines between the two codes, it will show a positive message "Able to merge". Then click on the button "Create pull request"

   After a Pull Request is made, the participant of the team and the owner of the repository will be able to see the changes.

   When everything is well, and the team or you, are satisfied, you can merge the two branches by clicking on "Merge pull request"

   Next step: Making sure that your local code is updated from the changes made to the master branch.



4. Getting changes from a server - `git pull`

If you make updates to your repository, people can download your changes with a single command - **pull**:

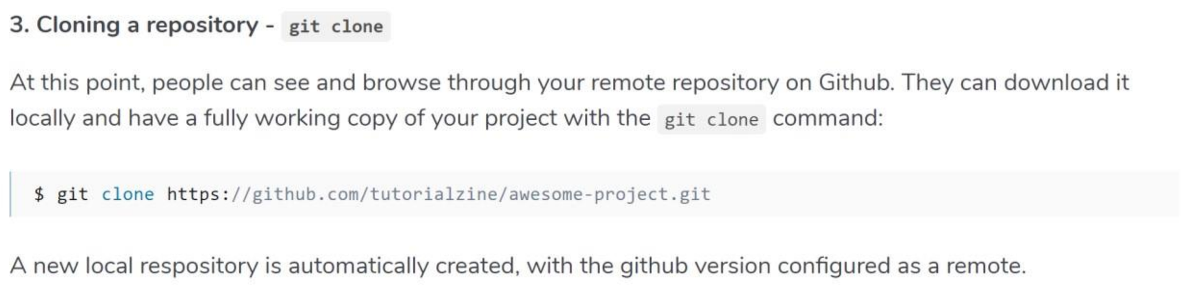
```
$ git pull origin master

From https://github.com/tutorialzine/awesome-project
 * branch            master     -> FETCH_HEAD
Already up-to-date.
```

**Exercise on Pull Request Git**

1. Make sure that you are on the local *master* branch
2. Then write **git pull origin master**
3. All the changes are now in your local repository! Great 😊

## How to fetch a repository from Github ?



3. Cloning a repository - `git clone`

At this point, people can see and browse through your remote repository on Github. They can download it locally and have a fully working copy of your project with the `git clone` command:

```
$ git clone https://github.com/tutorialzine/awesome-project.git
```

A new local respository is automatically created, with the github version configured as a remote.