$ git gud

LEARN GIT CLI BASICS 🧑‍💻

TODO

> What is Git?

> Bash CLI 101

> Creating a repository

> Working in a team

TODO

> **What is Git?** 👈

> Bash CLI 101

> Creating a repository

> Working in a team

> Best practices

WHAT IS GIT?

Git is a free and open source distributed Version Control System (VCS) designed to handle everything from small to very large projects with speed and efficiency.

Ok, but what does it actually mean?

I like to think about it as a very elaborate save system.
Let my real life version explain what I mean!

Best practices

TODO

> What is Git? ✅

> **Bash CLI 101** 👈

> Creating a repository

> Working in a team

> Best practices

BASH 101

`ls` - display content of the current directory

`mkdir <directory>` - create a new directory

`cd <directory>` - go to a directory

`touch <file>` - create a file

`cat <file>` - display file contents

`rm <file>` - remove a file

ADDING SSH KEY TO YOUR GITHUB ACCOUNT

This part might be a bit confusing at this moment and you might find it much more clearer after the workshop.

All need to know now is that wit an SSH key you don't have to authorize yourself everytime you want to push changes to your remote.

Now open your terminal and type in:

```
$ ssh-keygen -t ed25519
```

After successful completion, go to `~/.ssh` and check what files have been created.

ADDING SSH KEY TO YOUR GITHUB ACCOUNT

`ssh-keygen` generates two keys: private and public.
The former is just `id_ed25519` and the latter is `id_ed25519.pub`.

Display contents of public key file and copy all of it.

Nextly open GitHub and follow instruction of my real life version.

TODO

> What is Git? ✅

> Bash CLI 101 ✅

> **Creating a repository** 👈

> Working in a team

> Best practices

CREATING A GIT REPOSITORY

Let's create a new directory called `repo-test` and then inside this directory create a new file `README.md`.

Next, run this command to initialize it as a Git repository:

```
$ git init
```

With this command you can turn any directory into a Git repository. It's all thanks to the magical `.git` directory.

What are you talking about?
I can't see anything like that!

.git DIRECTORY

It is so important, that it is hidden by default.
This is the heart of your git repository.
Without it, no git commands would work.

Type the following and check again:

```
$ ls -al
```

Can you see it now?

.GITIGNORE

`.gitignore` is a special file in which you can put files which you want to, like the name suggests, ignore.

Anything listed in the file will be omitted when commiting new changes.

You can ignore single files, whole directories and its contents.
You can even use regular expressions to ignore series of files which names have some kind of pattern.

CREATING YOUR FIRST COMMIT

The command I end up using the most is:

```
$ git status
```

It prints out a concise description of the current status of your repository and its contents.

Now you can add some files to be included in the next commit using:

```
$ git add <file>
```

It's also possible to add whole directories, which will add all of its contents.

CREATING YOUR FIRST COMMIT

Now that you have all the desired files added, it's finally time to create a commit.
To do so you use the following command:

```
$ git commit
```

It will open a text editor (most likely *nano*) in which you can type a description of the changes you made.

If your commit message is short you can skip opening the editor and set the commit message directly in the command:

```
$ git commit -m <message>
```

CHECKING YOUR COMMIT HISTORY

Sometimes after coming back to the project after a short break you may not remember what was the last thing that has been done. Checking your commit history is a good way to catch up to speed and get to know what were the most recent changes.

You can view your commit history by typing in:

```
$ git log
```

PUSHING YOUR CHANGES TO A REMOTE

Pushing changes is basically uploading them to a remote server. You do that simply by typing:

```
$ git push
```

`Hmm, it doesn't seem to work...`

Ah, of course. We haven't set up a `remote` yet, that's why Git doesn't know where to push the changes.

A `remote`, which as name suggests, is a remote repository corresponding to the local one.

PUSHING YOUR CHANGES TO A REMOTE

To add a new `remote` to the repository use the following command:

```
$ git add remote origin <repo-url>
```

`repo-url` is the URL of the repository you're gonna create on GitHub - right now!

All done? Let's try with the push one more time.

TODO

> What is Git? ✅

> Bash CLI 101 ✅

> Creating a repository ✅

> **Working in a team** 👈

> Best practices

TIME FOR A BREAK 🕚

Form groups of 2-4.

After the break we'll be covering teamwork with Git.

TODO

> What is Git? ✅

> Bash CLI 101 ✅

> Creating a repository ✅

> **Working in a team** 👈

> Best practices

TEAMWORK AND GIT

Git is a great tool for a single developer.
But is even a **greater** one for developers working in a team.

Imagine working on a big project, which has hundreds of files, each at least 500 lines of code long. How do you handle that?

By using a version control system, for example Git!

CLONING A GIT REPOSITORY

Let's start by cloning this repo: git-gud-py

```
$ git clone <repo-url>
```

NOTE: *When cloning a repository it will create the directory with the exact name as the repository and put all the contents there. No need to create an additional directory for it.*

You can now see you have a local copy of the repository.

You don't have to perform any of the previous steps to start making commits, because `git clone` takes care of initialization and setting up the remote since you gave it all the required information.

But...

REMOTE REPOSITORY PERMISSIONS

You need proper permissions to push the commits to the remote.

You're free to clone any public repository but you cannot publish any changes unless you are part of the collaborators for this repository.

Try to make any change, commit and push it.

`Failed...`

In that case we need to remove the data that link the files in the repository to my remote and, link it to the one that you have access to.

Any ideas?

QUICK EXERCISE

> Remove `.git` directory.
> Initialize it as your own repository.
> Check if there is a commit history.
> Create and add a new remote.
> Make and push a new commit.

ADDING TEAM MEMBERS

To add other people to your git repository do the following:

> Open your repository on GitHub.
> Go to the `Settings` tab.
> On the sidebar select `Collaborators and team`.
> Click `Add people` button and type their name/username/email.

After few minutes, they should receive an email with the invitation from you.

Once they've accepted, you can edit their permissions.

Now everyone clone the repository.

ABOUT THE CLONED PROJECT

The repository you just cloned contains a very simple Python project.

> `main.py` is an entry point,
> `person.py` stores the definition of a Person class.

To run it just enter the following in the project directory:

```
$ python main.py
```

Completing definition of methods will be your next exercise to which will get soon.

WORKING WITH BRANCHES

Branching means you diverge from the main line of development and continue to do work without messing with that main line.

To create a new branch:

```
$ git branch <new-branch>
```

...then to change to the branch:

```
$ git checkout <new-branch>
```

Now each person in the group may create a new branch with corresponding name:

-----------------> **name | age | country | job** <-----------------

EXERCISE 2

What you need to do:

> Create a new branch.

> Complete the method of your choice.

> Call it in the `introduction()`.

> Create a new Person object in `main.py` and make it introduce itself.

> Commit changes on your new branch.

> Push to the remote.

> Mege all your branches together.

MERGING CHANGES

When you completed all the changes on your branch, you want to incorporate them into the branch you diverged form (usually `main`).

You can do that by switching to the desired branch and then typing:

```
$ git merge <branch-to-merge>
```

So let's say you were working on a branch `name`, you switch to `main` and then call `git merge name` to merge changes from `name` branch to `main`.

PULL REQUESTS

But when working in a team you want to keep track of changes made by everyone and prevent some spaghetti code to be part of the main branch.

`Pull Requests` are a way to let your team know that you're done with your change and want somebody to review it.

If everything is in order you are free to merge your changes.

Let real life me show how you can create a Pull Request.

MERGING CHANGES - CONFLICTS

Occasionally when making changes in parallel you and another developer will introduce different changes in the same place.
In that case Git is unsure what to do and informs you that there are `Conflicts`, which you need to resolve yourself.

If that is the case, then the list of files with conflicts will be displayed when you try to merge.

Now you need to go to each file and edit them according to your need.

Back to my real life version to show how it can be done.

TODO

> What is Git? ✅

> Bash CLI 101 ✅

> Creating a repository ✅

> Working in a team ✅

> **Best practices** 👈

BEST PRACTICES

> One change per commit.

> Descriptive commit messages.

> Name your branches after features.*

> Avoid pushing commits directly to `main`.

> NEVER force push to `main`.

This depends on the workflow of your team, but I find feature based branches to be the most common.

TODO

> What is Git? ✅

> Bash CLI 101 ✅

> Creating a repository ✅

> Working in a team ✅

> Best practices ✅

That is all!

Thank you for participating!

Now you're thinking with ~~portals~~ Git 🤖