Introduction to the environment & version control

Marcin Lewandowski

October 7, 2025

The goal of Class 1

1: Get you familiar with Git:

- A technology to track changes to a project (version control)
- Facilitates multiple people collaboration.
- Or multiple computers work.

The goal of Class 1

1: Get you familiar with Git:

- A technology to track changes to a project (version control)
- Facilitates multiple people collaboration.
- Or multiple computers work.

We will:

1. Use GitHub to post class materials
2. Expect you to post your homework/projects on GitHub

Thus working knowledge of Git is essential for this class and beyond!

The very basics of Git & GitHub

But before we begin...

Go to `https://github.com/` and create an account.

Preparing directory

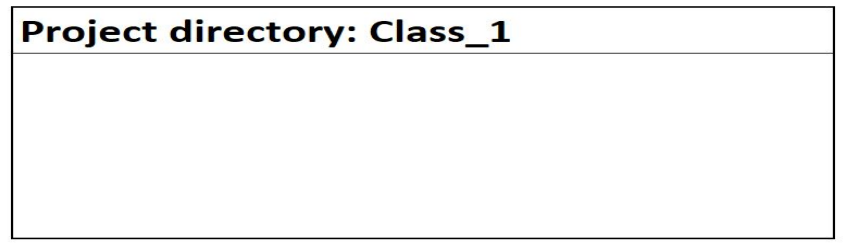
Before we dive into Git let us first create our project directory.
Simply create a Class_1 folder.

Preparing directory

Before we dive into Git let us first create our project directory.

Simply create a Class_1 folder.

This scheme will now depict the state of our directory:



Initializing a Local Repository

- A *Local repository* - a repository that is stored on a computer.
- A *Remote repository* - a repository that is hosted on a hosting service.

A local repository is represented by a hidden directory called .git in a project directory.

.git contains the data on the changes that made to the files in a project.

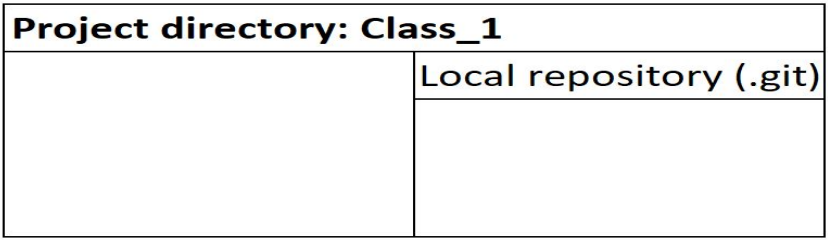
Initializing a Local Repository

- A *Local repository* - a repository that is stored on a computer.
- A *Remote repository* - a repository that is hosted on a hosting service.

A local repository is represented by a hidden directory called .git in a project directory.

.git contains the data on the changes that made to the files in a project.

Let us initialize a local repository



The key concepts of Git

There are four important areas to be aware of when you are working with Git:

- Local repository – this one you already know
- Working directory
- Staging area
- Commit history

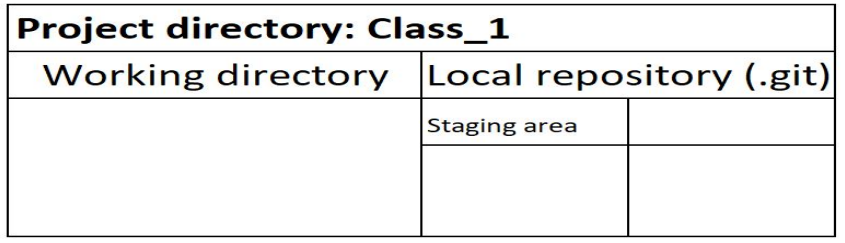
The working directory

- Working directory is like a current workbench
- It is populated by files related one version of a project.
- It is where you add, edit, and delete files and directories.

Project directory: Class_1	
Working directory	Local repository (.git)

The staging area

- The staging area is similar to a rough draft space.
- You can add and remove files. Prepare what to include in the next save.
- The staging area is represented by a file in the .git directory called index.



The commit history

But what is a commit?

- A commit in Git is basically one version of a project!
- Every commit has an unique commit hash (sometimes called a commit ID).

The commit history

But what is a commit?

- A commit in Git is basically one version of a project!
- Every commit has an unique commit hash (sometimes called a commit ID).

So what is commit history?

- The commit history is where all of your commits exist.

The commit history

But what is a commit?

- A commit in Git is basically one version of a project!
- Every commit has an unique commit hash (sometimes called a commit ID).

So what is commit history?

- The commit history is where all of your commits exist.
- Every time you make a commit it is saved in commit history

The commit history

But what is a commit?

- A commit in Git is basically one version of a project!
- Every commit has an unique commit hash (sometimes called a commit ID).

So what is commit history?

- The commit history is where all of your commits exist.
- Every time you make a commit it is saved in commit history
- It is represented by the objects directory inside the .git directory

The commit history

But what is a commit?

- A commit in Git is basically one version of a project!
- Every commit has an unique commit hash (sometimes called a commit ID).

So what is commit history?

- The commit history is where all of your commits exist.
- Every time you make a commit it is saved in commit history
- It is represented by the objects directory inside the .git directory

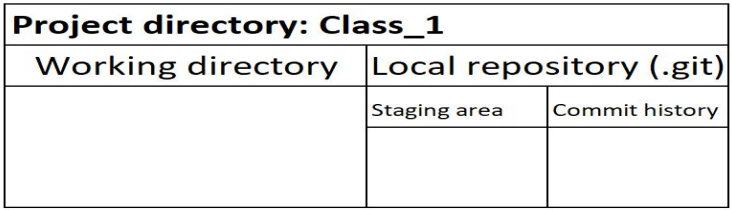
The commit history

But what is a commit?

- A commit in Git is basically one version of a project!
- Every commit has an unique commit hash (sometimes called a commit ID).

So what is commit history?

- The commit history is where all of your commits exist.
- Every time you make a commit it is saved in commit history
- It is represented by the objects directory inside the .git directory

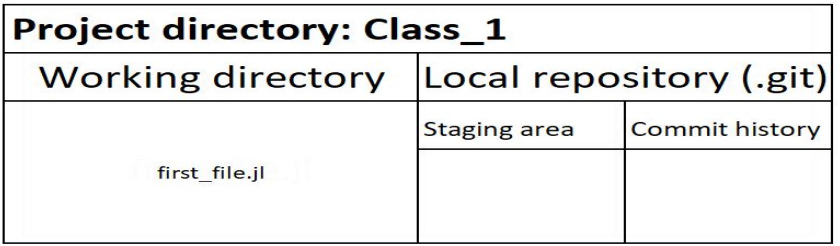


Creating a new file

Let us create first_file.jl in our Class_1 folder.

Creating a new file

Let us create first_file.jl in our Class_1 folder.



Creating a new file

Let us create first_file.jl in our Class_1 folder.

<table>
<tr><th colspan="3">Project directory: Class_1</th></tr>
<tr><th>Working directory</th><th colspan="2">Local repository (.git)</th></tr>
<tr><td rowspan="2">first_file.jl</td><td>Staging area</td><td>Commit history</td></tr>
<tr><td></td><td></td></tr>
</table>

- Since first_file.jl is in Class_1 project directory, it is in the working directory
- BUT first_file.jl is not yet in your repository, it is an untracked file.
- Untracked file is not version controlled by Git
- Once added to the staging area and committed it becomes a tracked file

Making a commit 1

Committing is important because it allows you to back up your work and avoid losing it.
To commit you first add a file into the staging area

<table>
<tr><th colspan="3">Project directory: Class_1</th></tr>
<tr><th>Working directory</th><th colspan="2">Local repository (.git)</th></tr>
<tr><td rowspan="2">first_file.jl</td><td>Staging area</td><td>Commit history</td></tr>
<tr><td>first_file.jl</td><td></td></tr>
</table>

- The first_file.jl is now both in working directory & in the staging area!

Making a commit 1

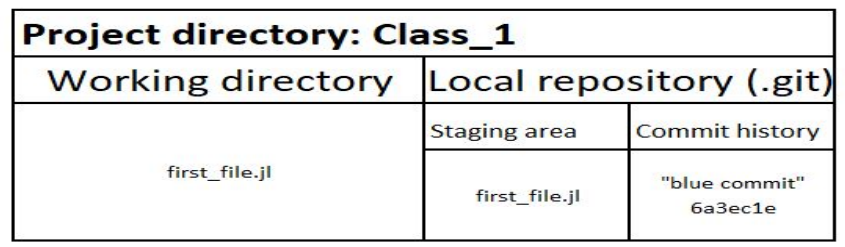
Committing is important because it allows you to back up your work and avoid losing it.
To commit you first add a file into the staging area

<table>
<tr><th colspan="3">Project directory: Class_1</th></tr>
<tr><th>Working directory</th><th colspan="2">Local repository (.git)</th></tr>
<tr><td rowspan="2">first_file.jl</td><td>Staging area</td><td>Commit history</td></tr>
<tr><td>first_file.jl</td><td></td></tr>
</table>

- The first_file.jl is now both in working directory & in the staging area!
- Adding files to the staging are does not move them, it *copies* them.

Making a commit 2

To make a commit means to save a version of a project!
It is good practice to add short message to your commit



Remote repositories

Are useful because:

- You can back-up your project online.
- Access a Git project from multiple computers.
- Collaborate with others.

Remote repositories

Are useful because:

- You can back-up your project online.
- Access a Git project from multiple computers.
- Collaborate with others.

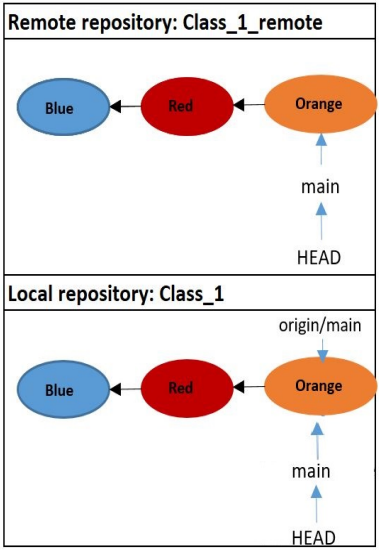
Local repositories and remote ones will not update each other automatically!

Remote repositories

Are useful because:

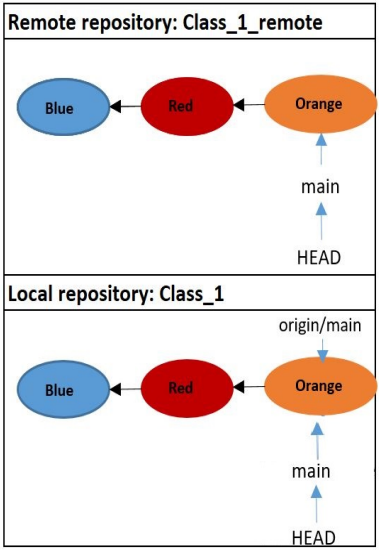
- You can back-up your project online.
- Access a Git project from multiple computers.
- Collaborate with others.

Local repositories and remote ones will not update each other automatically!



Remote repositories

- .git/refs contents changed; local repository now has a "remote-tracking branch"
- Origin is the default shortname Git associates with a remote repository
- Now let's edit our first_file.jl in the remote repository and pull the changes!



Concept check!

In-class excercise:

1. Create a new file second_file.jl in our Class_1 folder, write a line of code.
2. Add the file to the staging area
3. Commit it with a proper message
4. Push those changes to the remote repository
5. Edit the second_file.jl in the remote repository
6. Pull changes from the remote repository

On Three-Way Merges

- Three-way merges can deal with the merge conflicts.

On Three-Way Merges

- Three-way merges can deal with the merge conflicts.
- Conflicts arise if two branches have changes to the same parts of a file.

On Three-Way Merges

- Three-way merges can deal with the merge conflicts.
- Conflicts arise if two branches have changes to the same parts of a file.
- Thus its development histories diverge, it is not possible to get from one to the other through parental links.

On Three-Way Merges

- Three-way merges can deal with the merge conflicts.
- Conflicts arise if two branches have changes to the same parts of a file.
- Thus its development histories diverge, it is not possible to get from one to the other through parental links.

On Three-Way Merges

- Three-way merges can deal with the merge conflicts.
- Conflicts arise if two branches have changes to the same parts of a file.
- Thus its development histories diverge, it is not possible to get from one to the other through parental links.

Let's create a merge conflict with the remote repository:

1. Edit the first_file.jl in the remote repository

On Three-Way Merges

- Three-way merges can deal with the merge conflicts.
- Conflicts arise if two branches have changes to the same parts of a file.
- Thus its development histories diverge, it is not possible to get from one to the other through parental links.

Let's create a merge conflict with the remote repository:

1. Edit the first_file.jl in the remote repository
2. Pull changes from the remote repository

On Three-Way Merges

- Three-way merges can deal with the merge conflicts.
- Conflicts arise if two branches have changes to the same parts of a file.
- Thus its development histories diverge, it is not possible to get from one to the other through parental links.

Let's create a merge conflict with the remote repository:

1. Edit the first_file.jl in the remote repository
2. Pull changes from the remote repository
3. Integrate the changes in the local branch of the local repository (merge)

On Three-Way Merges

- Three-way merges can deal with the merge conflicts.
- Conflicts arise if two branches have changes to the same parts of a file.
- Thus its development histories diverge, it is not possible to get from one to the other through parental links.

Let's create a merge conflict with the remote repository:

1. Edit the first_file.jl in the remote repository
2. Pull changes from the remote repository
3. Integrate the changes in the local branch of the local repository (merge)
4. Push those changes to the remote repository

VS code has an extremely useful editor to deal with merge conflicts

You can simultaneously compare:

- Current version

VS code has an extremely useful editor to deal with merge conflicts

You can simultaneously compare:

- Current version
- Incoming change

VS code has an extremely useful editor to deal with merge conflicts

You can simultaneously compare:

- Current version
- Incoming change
- Base (i.e. last common commit)

VS code has an extremely useful editor to deal with merge conflicts

You can simultaneously compare:

- Current version
- Incoming change
- Base (i.e. last common commit)
- The resulting file

VS code has an extremely useful editor to deal with merge conflicts

You can simultaneously compare:

- Current version
- Incoming change
- Base (i.e. last common commit)
- The resulting file

VS code has an extremely useful editor to deal with merge conflicts

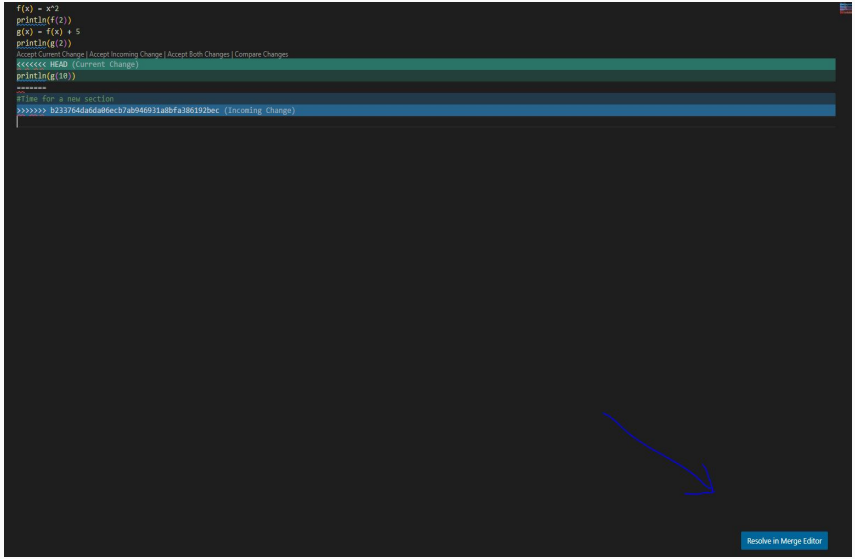
You can simultaneously compare:

- Current version
- Incoming change
- Base (i.e. last common commit)
- The resulting file

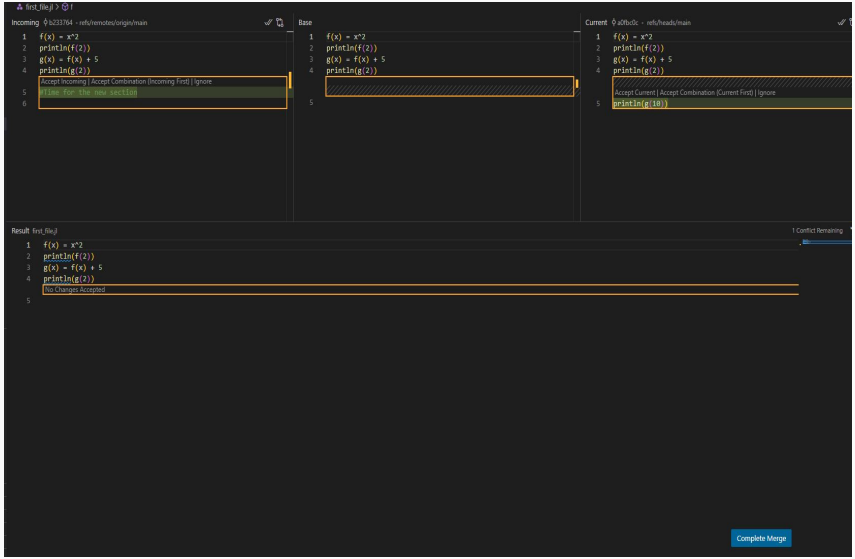
Simply click on the "Resolve in Merge Editor" button.
Then:

1. Track the number of remaining conflicts
2. Accept either current, incoming or keep the way code was written in base!

Go into Merge Editor



Conflict resolution view



Concept check!

Learn how to deal with the merge conflicts!

1. Edit second_file.jl in the remote repository.
2. Edit the second_file.jl (in the same place) locally, commit, push.
3. Integrate the changes in the local branch of the local repository (merge).
4. Push the commit with resolved conflicts to the remote repository.

Cloning the remote repository

Cloning is an essential part of collaborative process!

- In home you can clone the repository you created, make some changes, and push it.

Cloning the remote repository

Cloning is an essential part of collaborative process!

- In home you can clone the repository you created, make some changes, and push it.

Here we want you to clone repository with the class materials.

Do not make any changes to your local class materials repository

It will create conflict as we update the materials (it will not be an issue for us, it will be an issue for you)

Possible workflow:

1. Clone official "Class materials" repository.
2. Pull materials at the beginning of each class.
3. Create your own repository.
4. Copy the relevant scripts into your own "Class_materials" folder.
   - You will play and experiment with scripts in this copied files in your folder
5. Commit and push your repository after every class
   - This way you can access your scripts & materials from any computer!

Do not make any changes to your local class materials repository

It will create conflict as we update the materials (it will not be an issue for us, it will be an issue for you)

Possible workflow:

1. Clone official "Class materials" repository.
2. Pull materials at the beginning of each class.
3. Create your own repository.
4. Copy the relevant scripts into your own "Class_materials" folder.
   - You will play and experiment with scripts in this copied files in your folder
5. Commit and push your repository after every class
   - This way you can access your scripts & materials from any computer!

Concept check: implement those steps right now!

Homework rules

You have to subimt it via Github!

1. Team-up in groups of 4/5 students
2. Create a repository for your group
3. Share the link with us via email!
4. Submit the homework in the repository

Additional resources:

1. There are a ton of online tutorials: e.g. `https://swcarpentry.github.io/git-novice/`
2. This class very closely follows the "Learning Git" book by Anna Skoulikari
3. For the resources in VS code specifically, see those short clips:
   - `https://www.youtube.com/watch?v=i_23KUAEtUM`
   - `https://www.youtube.com/watch?v=HosPml1qkrg`
4. And the entire playlist!

Setting up work environment: packages in Julia

Navigating in the terminal

Sometimes we need to navigate ourselves in the terminal.
We will need only two commands:

- pwd() - print working directory
- cd("path") - change directory
  - cd("..") - go one folder up
  - cd("materials") - go into the folder materials
  - cd("materials /class1") - go to the folder materials then to the folder class1
  - you can also just paste the whole path

The concept of a package

Some functionalities are not available in basic Julia environment.

The concept of a package

Some functionalities are not available in basic Julia environment.
Packages are of two types:

1. Standard libraries are shipped with Julia (e.g. Linear Algebra library)
   - Here we just write "using LinearAlgebra" to have functionalities available
2. Other require external installation (e.g. Plots)

The concept of a package

Some functionalities are not available in basic Julia environment.
Packages are of two types:

1. Standard libraries are shipped with Julia (e.g. Linear Algebra library)
   - Here we just write "using LinearAlgebra" to have functionalities available
2. Other require external installation (e.g. Plots)

To manage the latter we will use Julia fantastic Package manager.

Julia Package Manager

- Traditional package managers install **global** set of packages.

Julia Package Manager

- Traditional package managers install **global** set of packages.
- Julia uses **local** environments: independent sets of packages **local** to a project!

Julia Package Manager

- Traditional package managers install **global** set of packages.
- Julia uses **local** environments: independent sets of packages **local** to a project!
  - You can use newer version of a library in one project and older in the other.

Julia Package Manager

- Traditional package managers install **global** set of packages.
- Julia uses **local** environments: independent sets of packages **local** to a project!
  - You can use newer version of a library in one project and older in the other.
- This reduces the "dependency hell"!

Julia Package Manager

- Traditional package managers install **global** set of packages.
- Julia uses **local** environments: independent sets of packages **local** to a project!
  - You can use newer version of a library in one project and older in the other.
- This reduces the "dependency hell"!
- Allows one to easily come back to the projects created long time ago!

Julia Package Manager

- Traditional package managers install **global** set of packages.
- Julia uses **local** environments: independent sets of packages **local** to a project!
  - You can use newer version of a library in one project and older in the other.
- This reduces the "dependency hell"!
- Allows one to easily come back to the projects created long time ago!
- This quality of Julia greatly enhances research reproducibility.

Julia Package Manager

- Traditional package managers install **global** set of packages.
- Julia uses **local** environments: independent sets of packages **local** to a project!
  - You can use newer version of a library in one project and older in the other.
- This reduces the "dependency hell"!
- Allows one to easily come back to the projects created long time ago!
- This quality of Julia greatly enhances research reproducibility.

Julia Package Manager

- Traditional package managers install **global** set of packages.
- Julia uses **local** environments: independent sets of packages **local** to a project!
  - You can use newer version of a library in one project and older in the other.
- This reduces the "dependency hell"!
- Allows one to easily come back to the projects created long time ago!
- This quality of Julia greatly enhances research reproducibility.

The relevant information about libraries and its dependencies is stored in the:

- Project.toml
- Manifest.toml

Activating local environment

1. Change the working directory of Julia to the relevant folder (use "cd")
2. In Julia REPL type "]" to get to the package manager mode.
3. The prompt will change showing name of the active environment (usually the global one).
4. Activate the project environment by using the "activate ." command (it will activate the environment in the current folder).

- If you want to add new package write "add XYZ" (e.g. Plots)
- If you cloned the repo and want to download all the required packages use the "instantiate" command.

So what is happening when we add a package?

- The package manager downloads the package.

So what is happening when we add a package?

- The package manager downloads the package.
  - The general registry: `https://github.com/JuliaRegistries/General`

So what is happening when we add a package?

- The package manager downloads the package.
  - The general registry: `https://github.com/JuliaRegistries/General`
- It also downloads all the dependencies of the package (registered in Manifest.toml).

So what is happening when we add a package?

- The package manager downloads the package.
  - The general registry: `https://github.com/JuliaRegistries/General`
- It also downloads all the dependencies of the package (registered in Manifest.toml).
- The "environment dependencies" are registered in the the Project.toml file.

So what is happening when we add a package?

- The package manager downloads the package.
  - The general registry: `https://github.com/JuliaRegistries/General`
- It also downloads all the dependencies of the package (registered in Manifest.toml).
- The "environment dependencies" are registered in the the Project.toml file.

So what is happening when we add a package?

- The package manager downloads the package.
  - The general registry: `https://github.com/JuliaRegistries/General`
- It also downloads all the dependencies of the package (registered in Manifest.toml).
- The "environment dependencies" are registered in the the Project.toml file.

This way we can easily reproduce the environment of the project

So what is happening when we add a package?

- The package manager downloads the package.
  - The general registry: `https://github.com/JuliaRegistries/General`
- It also downloads all the dependencies of the package (registered in Manifest.toml).
- The "environment dependencies" are registered in the the Project.toml file.

This way we can easily reproduce the environment of the project

Let's test "instantiate" command with the "Example" package.
`https://github.com/JuliaLang/Example.jl/blob/master/src/Example.jl`

Some other useful commands:

- status
  - Displays where the file managing the dependencies in this environment is located (Project.toml).
  - Displays which packages are installed in this environment.
- remove XYZ removes XYZ package from the environment.
- help in package manager mode gives you all the commands with short description
- "Backspace" KEY will return you from package manager to Julia REPL.

Let us now experiment with package manager in Julia!