Day II - Part II - GitLab exercise

December 9, 2020

1 GitLab exercise

The final part of the workshop focuses on implementing a software package for calculating different distance measures. The exercise uses the git repository server at GWDG "GitLab".

This part of the workshop covers:

- Creating GitLab repositories
- Adding a Open Source License
- Packaging and testing software
- Package documentation
- Creating merge requests
- Continuous integration (automated building and testing).

Here we use gitlab since it does not require any additional registration procedure. For the sake of completeness, here is a (non-comprehensive) list of alternative git hosting services:

- github (https://github.com): probably the best know git service, free of charge for public repositories
- bitbucket (https://bitbucket.com): From the Atlassian family, let's you have a limited number of private repositories for free.
- gitlab (https://gitlab.com): Github's main contender. Easy to deploy as institutional service.

Here, we use the gitlab installation at GWDG. Everybody with a `*.mpg.*` email address automatically also has a gitlab account at GWDG. You can create as many public, private, or internal repositories as you like. Advanced features like Continuous Integration are part of the service, no Third Parties involved.

There is also a gitlab instance at MPI EvolBio, but the version is a bit outdated and it is missing many of the features that we will use in the workshop.

2 Log into GitLab

Navigate to https://gitlab.gwdg.de and log in with your institutional email address and password.

3 Add a ssh key to your gitlab profile

The folks at GWDG are a bit crazy about security. Therefore, you can only push (upload commits) after configuring your account for *passwordless* authentication using ssh keys.

3.0.1 Create ssh keypair.

If you already have a ssh keypair on your local computer, skip this step. Otherwise, follow these instructions:

- Run the following command in a shell (git shell on windows)

```
$ ssh-keygen
```

Simply accept all default settings by pressing [Enter] on each prompt. For simplicity, do not enter a key passphrase.

3.0.2 Upload public key to gitlab.

** This step is mandatory unless your public key is already uploaded to gitlab. **

First paste your public key to the command line:

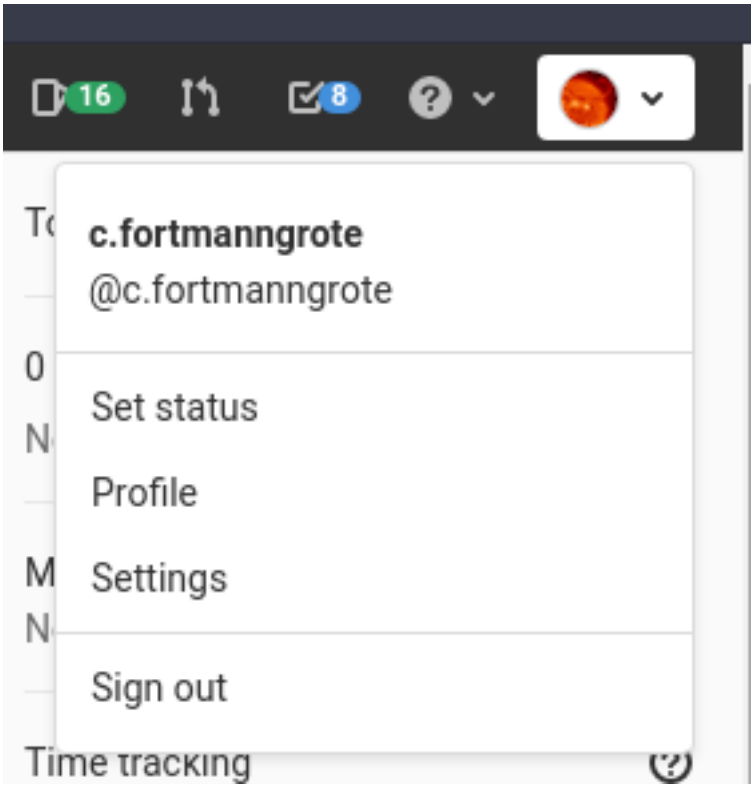
```
$ cat ~/.ssh/id_rsa.pub
```

Then use your mouse to copy the entire public key. It start with `ssh-rsa` and ends with your username and hostname of your computer. E.g., here is the begin and end of my public key:

```
ssh-rsa AAAA...inxol6NQ== grotec@micropop037
```

NOTE Disclosing *public* ssh keys is no security issue. But **NEVER EVER** disclose your *private* key. It's in the same directory and has the same filename except for the extension.

Now navigate to your gitlab profile by clicking on your avatar icon in the top right corner of your gitlab page. Then select `Settings`.

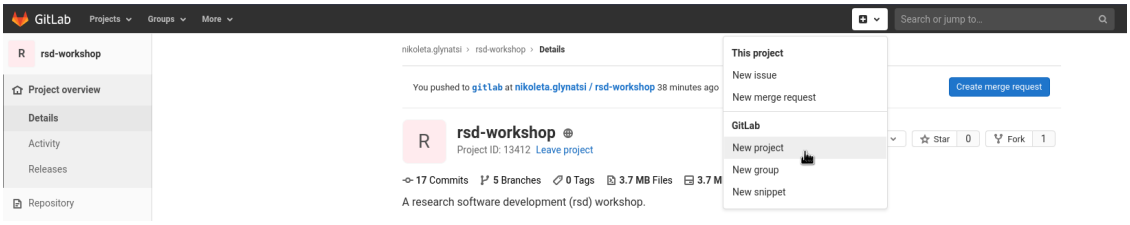


On your profile setting page, go to `SSH Keys` in the left navigation panel, paste your public ssh key into the text box. Click `Add key` to submit the new key.

4 Create a repository (aka Project)

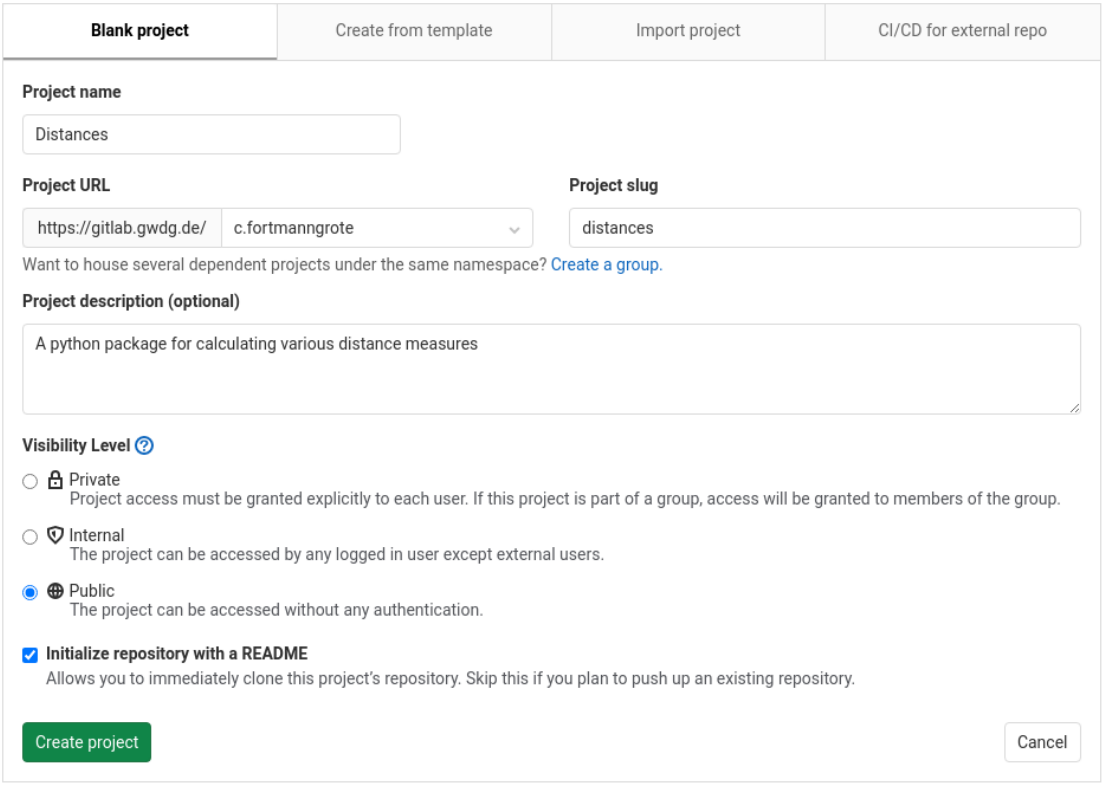
A software repository, or "repo" for short, is a storage location for software packages. A repository is managed through a version control system. Here, we use git.

To create a repository on gitlab click on "New" located at the left side of the home page.



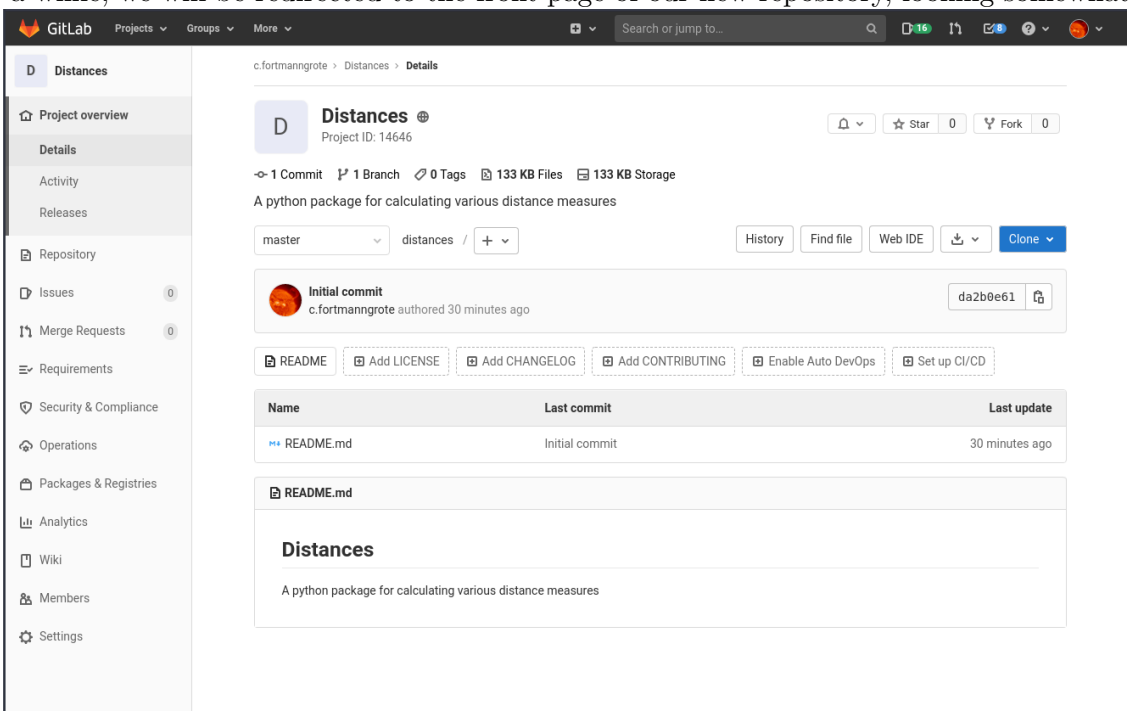
Here we are going to create the project's repository.

- **Repository name**: distances (this will automatically fill in the project slug, the last part of the repository URL.
- **Description**: A python package for calculating various distance measures.
- **Private**. We are going to create a public repository, so outsiders can use and contribute to our amazing code.
- **Initialize repository with a README**. GitLab allows us to initialize a repository with a README file. A README file contains information about other files in a directory or archive of computer software.



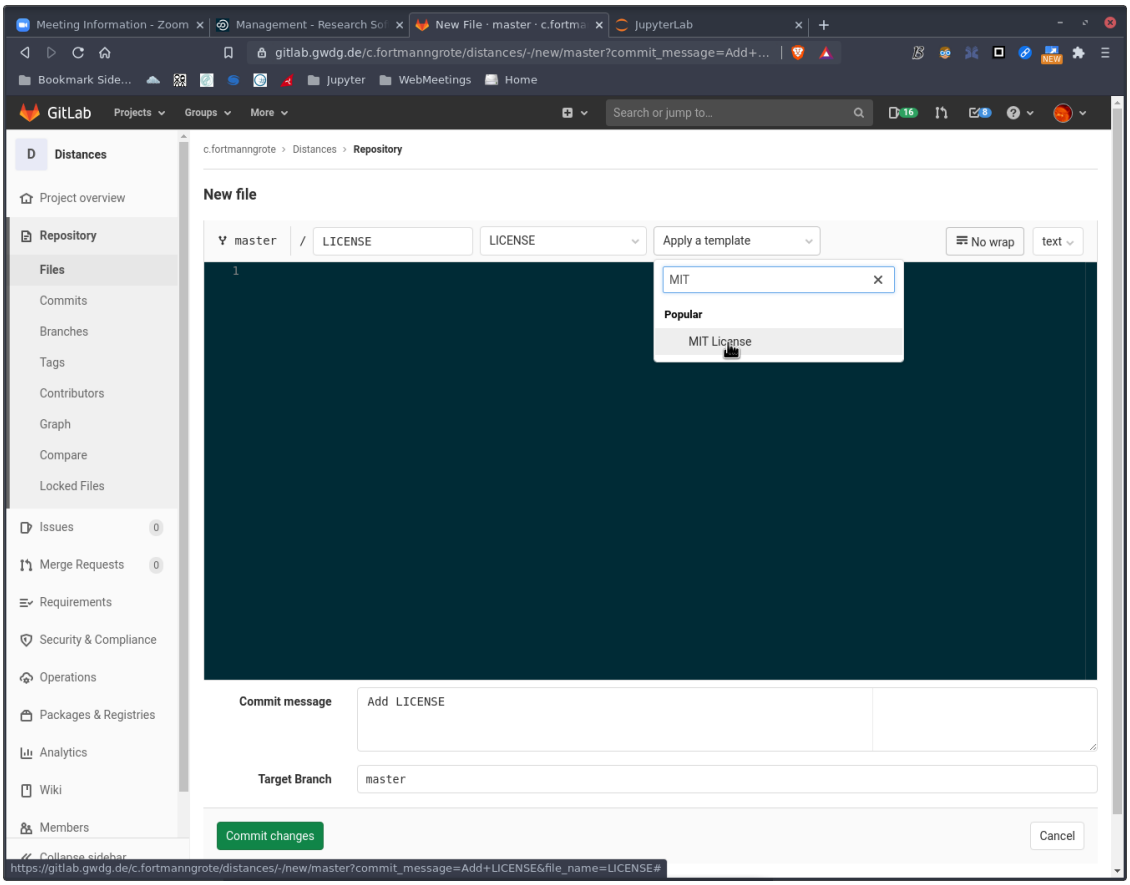
We click `Create project` to finalize the creation of our repository.

After a while, we will be redirected to the front page of our new repository, looking somewhat like this:



To complete the setup, we will add a license file. Click on `Add LICENSE`. Since this is a public

repository, we will add an Open Source License. Here, we choose the MIT License. Click on `Apply a template` and type "MIT" into the search field. Then click on "MIT License".

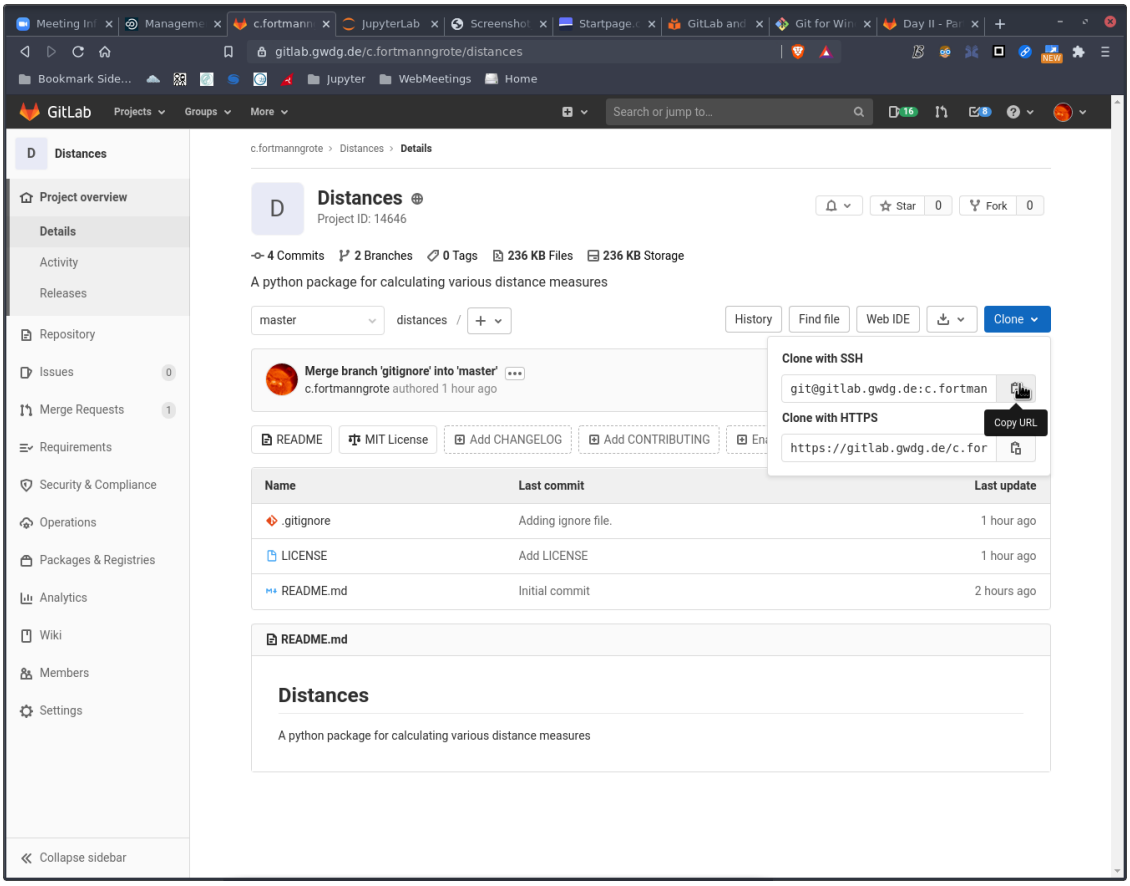


Finally click on `Commit changes`. Congratulations! You have just pushed your first commit to the repository.

5 Clone the repository

When we create a repository on GitLab, the new project exists as a *remote* repository. It serves as the central *hub* where all changes to the code base will be collected and where prospective users go to download the code. While GitLab allows you to edit the code online, developers typically work on a local copy (i.e. on their laptop/desktop computer) of the repository (a *clone*) and synchronize changes between the local repo (or multiple local copies on different computers) and the remote repo using `git`.

To clone a GitLab repository we need to copy its address. On the front page of your newly created repo, click on `Clone` and then on the `Copy` icon next to the **first** web address, the one starting with "git".



Then run the following command in a terminal/command prompt:

```
$ git clone <the_address_we_just_copied>
```

Hint: On most terminals you can right-click and paste the copied text into the command line.

The `git clone` command will create a new directory named like the repository, "distances" right under your current working directory.

Q: Remember how to detect the current working directory?

Make sure that you run this command while on the location you want to clone the repository.

For example if I wanted to clone the repository on my Deskop I would run the following commands:

```
$ cd Desktop
$ git clone https://gitlab.gwdg.de/<GITLAB_USERNAME>/distances.git
```

Now it's time to start developing. First, change into the repository's directory:

```
$ cd distances
```

Before we start implementing our package, we will add one more configuration file to our repository, the `.gitignore` file. This file lists all files and filename patterns that we do **not** want to be monitored by git. For example, we may not want to include files generated when running our code. Running python code typically results in the creation of various "compiled" files with extensions

like .pyc or .pyo (depending on the operating system). To exclude such files from being monitored by git, we run the following command:

```
$ echo "*.pyc" > .gitignore
$ echo "*.pyo" >> .gitignore
```

Q: What's the meaning of ">" and "»" in above commands? Why not using ">" again in the second command?

This creates the `.gitignore` file in the repository root directory.

Q: List all files in the repository root directory. Can you confirm that the `.gitignore` file was created?

To put the `.gitignore` file under version control, we add and commit it to our repo:

```
$ git add .gitignore
$ git commit .gitignore -m "Add gitignore file."
```

6 Branches

A freshly created project has only one branch, `master`. It is good practise to not work on the master branch directly. This way, we can make sure that the tip of the master branch always reflects the last stable (i.e. tested) point in the history of the code. Instead we create different branches for working on different parts of a project.

So let's create a branch called `implement-distances-package`.

```
$ git branch implement-distances-package
$ git checkout implement-distances-package
```

The repository's structure right now is as follows:

```
|--- .gitingore
|--- LICENSE
|--- README.md
```

Spend some time now to familiarize yourself with the structure and the current files of the repository.

Qs: - List all files (including hidden files). - Look at the commit history (log). - What is the current status of your repo? - Which branches exist? - Which branch is checked out? - List the content of the hidden directory `.git`.

7 Implement the distance function

We are now ready to write a package that calculates distances.

1. Inside our repository root directory, we will create a(nother) folder called `distances`. This can be done with the command:

```
$ mkdir distances
```

2. In the folder we just created we are going to add a file called `euclidean.py`. This file will contain the code needed to calculate the euclidean distance of two vectors.

Alter the file `euclidean.py` to contain the following lines of code:

```
import math

def euclidean_distance(u, v):
    """
    Computes the Euclidean distance between two vectors `u` and `v`.

    The Euclidean distance between `u` and `v`, is defined as:

    \sqrt{(u_1 - v_1) ^ 2 + ... + (u_n - v_n) ^ 2}

    Parameters
    ----------
    u : list
        Input vector.
    v : list
        Input vector.

    Returns
    -------
    euclidean : double
        The Euclidean distance between vectors `u` and `v`.
    """
    distance = 0

    for u_i, v_i in zip(u, v):
        distance += (u_i - v_i) ** 2

    return math.sqrt(distance)
```

3. Now that the function is implemented we need to commit the change.

It is always good practice, to check the status of your repo before making any changes.

Q: Confirm that the new file is currently untracked.

Now add it to the repo:

```
$ git add distances/euclidean.py
$ git commit
```

You can use the following commit message:

```
implement euclidean distance
```

The structure of the repository should now be the following:

```
|--- distances
    |--- euclidean.py
|--- .gitingore
|--- LICENSE
|--- README.md
```

8 Test the function

Now that we have written the function it is time to use it. Creating a test for an implemented function is a great way to:

1. Demonstrate it's usage
2. Test it's implementation

Lets's create a file `test_euclidean.py` at the root of the repository such as the structure now is:

```
|--- distances
    |--- euclidean.py
|--- .gitingore
|--- LICENSE
|--- README.md
|--- test_euclidean.py
```

Open `test_euclidean.py` with your editor and alter it so that it looks like:

```
import distances

u = (2, -1)
v = (-2, 2)

print(distances.euclidean_distance(u, v))
```

Trying to run this file using the command:

```
$ python test_euclidean.py
```

should return the following error:

```
(base) ~/Desktop/distances(implement-distances-package ✗) python test_euclidean.py
Traceback (most recent call last):
  File "test_euclidean.py", line 6, in <module>
    assert distances.euclidean_distance(u, v) == 5
AttributeError: module 'distances' has no attribute 'euclidean_distance'
```

In order to let python know that files under the module `distances` we need to creat a file called `__init__.py` which will be under `distances` and will include the following line:

```
from .euclidean import euclidean_distance
```

Now run the command:

```
$ python test_euclidean.py
```

again.

Now let's alter the code to include an `assert` command:

```
import distances

u = (2, -1)
v = (-2, 2)

assert distances.euclidean_distance(u, v) == 5
```

and run `python test_euclidean.py`.

Fantastic. Now, let's commit this change:

```
$ git add test_euclidean.py
$ git add distances/__init__.py
```

with the following commit:

```
add test for the euclidean distance
```

9 Pytest

Currently the command:

```
$ python test_euclidean.py
```

does not give any feedback. In our case that is because the `assert` command is `True` and thus there is nothing to report.

Several packages already exist that can make tests output more useful. An example of such a Python package is `pytest`. `pytest` has been installed on your computers with Anaconda.

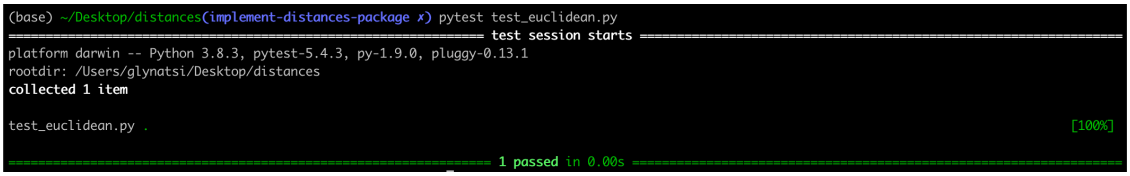
Alter the code in `test_euclidean.py` to:

```
import distances

def test_euclidean():
    u = (2, -1)
    v = (-2, 2)

    assert distances.euclidean_distance(u, v) == 5
```

and now use the following command to run the tests:

```
$ pytest test_euclidean.py
```

This should return something like:



Now commit the changes with a message: `use the library pytest to run tests`.

The commands are:

```
$ git add test_euclidean.py
$ git commit
```

10 Document the package

When you are developing a software package, you want to make available for other people to use. Thus, you need to document your project.

This include, letting people know how to install your project, its purpose and functionality, its license and how to test it.

All these details can be included on a projects `README.md`.

The current `README.md` of our project looks like this:

11 distances

A package for calculating different distance measures.

Alter the file to include details of your project.

An example of a `README.md`:

12 distances

A package for calculating different distance measures. # Installation The project can be cloned locally using the following command:

```
$ git clone <path>
```

13 Usage

Currently the following distance measures are implemented in the package: - Euclidean distance. # Tests

The package is being tested using `pytest`. To run the test suite use the command:

```
$ pytest test_euclidean
```

14 License

The package is under the MIT license.

Once you are done altering the `README.md` remember to commit your changes.

15 Open a pull request

Our project is now ready. We have implemented a package that can calculate the euclidean distance and we want to share it with the world.

Before we update our copy on GitHub run the command:

```
$ git status
```

We can see that there are a few files that have not been committed, but also that we don't reconsiged these files. You don't need to worry about these files. Let's add them to our `.gitignore`.

So our `.gitingore` should include the lines:

```
__pycache__/
distances/__pycache__/
```

Add the changes, commit and run `git status` again.

Now that everything has been committed we are ready to update the copy of our project on GitHub.

To do that you need to run the following command:

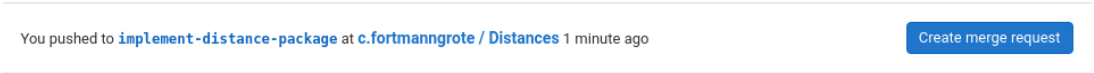
```
$ git push -u origin implement-distances-package
```

and then you should something like this:

```
$> git push -u origin implement-distance-package  [implement-distance-package]
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 609 bytes | 609.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote:
remote: To create a merge request for implement-distance-package, visit:
remote:   https://gitlab.gwdg.de/c.fortmanngrote/distances/-/merge_requests/new?
merge_request%5Bsource_branch%5D=implement-distance-package
remote:
To gitlab.gwdg.de:c.fortmanngrote/distances
 * [new branch]      implement-distance-package -> implement-distance-package
Branch 'implement-distance-package' set up to track remote branch 'implement-dis
tance-package' from 'origin'.
```

Once you have run the command open your repository on GitLab.

There you should see the following on the top of your page:

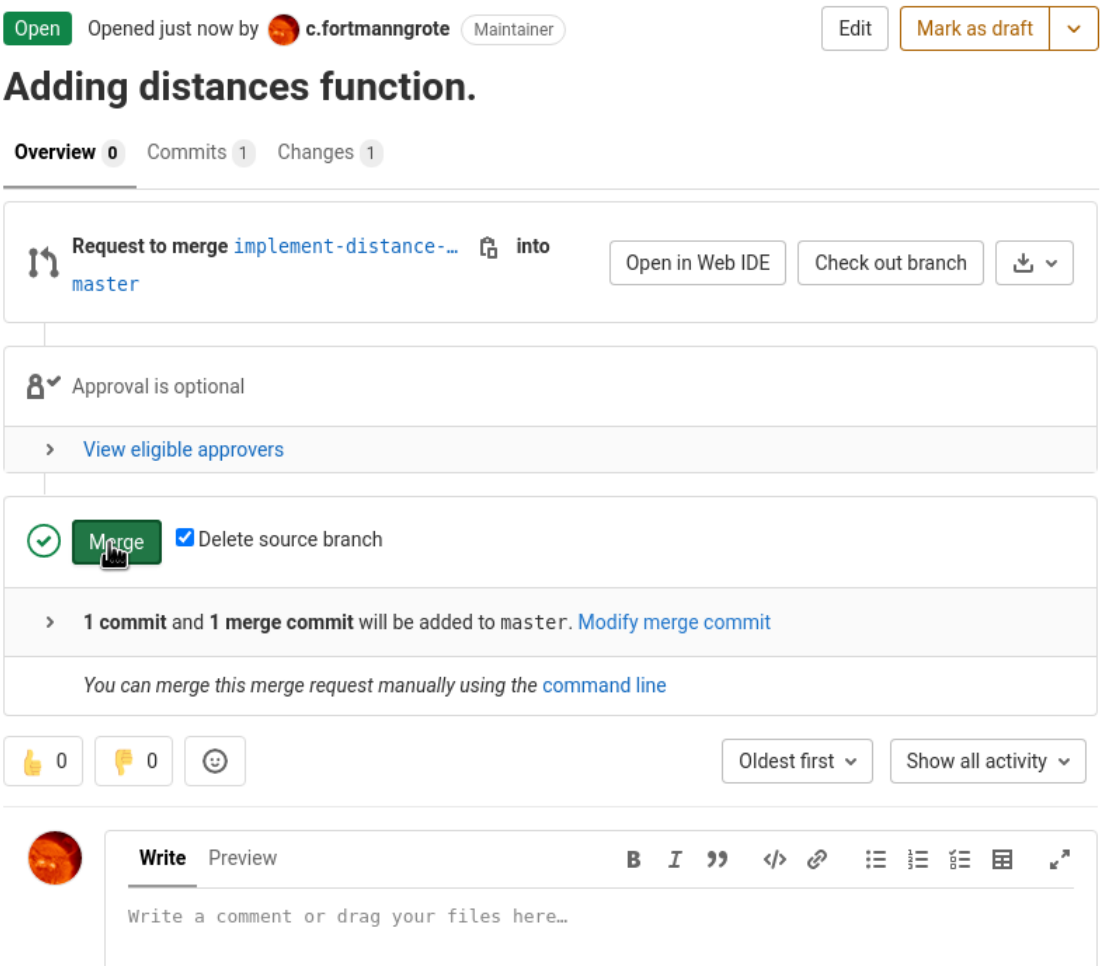


Click on `Create merge request`.

GitLab then transfers you to the `Merge request` page. Here you can review the changes you have made and `request` for your code to become part of the projects main branch.

Take sometime to familiarize yourself with the `Merge request` page and then click on `Submit merge request`.

The page will then refresh and you can again review the changes. Once all issues (if any) are resolved, you can finally accept the merge request by clicking the `Merge` button. All changes from the `implement-distances-package` branch will then be incorporated into the master branch. Optionally, the old branch is deleted.



Congratulations you have created and merged your first merge request!

NOTE: On github, the corresponding functionality is called “Pull request”.

[ ]: