



*Faculty of Mechanical Engineering
and Robotics*

*Department of Robotics and
Mechatronics*



Python for Machine Learning and Data Science
Course for Mechatronic Engineering

Additional instruction:

GIT
and
version control and cooperation

You will learn: How to work with GIT, colab integration, collaboration in GIT.

Course supervisor:
Ziemowit Dworakowski, zdw@agh.edu.pl

Instruction author:
Mateusz Heesch

Introduction

This laboratory class will cover usage of GIT in colab, with a focus on collaboration (hence some tasks will be performed in groups). Though technically version control, code sharing, and co-development can be done using other tools, GIT is universally accepted as the “go-to” option for that – getting familiar with it will be useful not only for streamlining project assignments in this course, but it’s safe to say you’ll also use it in any programming (or programming-adjacent) jobs you might hold after graduating. This instruction will mostly cover bare basics, you’re encouraged to refer to extensive documentation and troubleshooting available online whenever you run into more complex scenarios.

Let’s start with introducing some basic relevant terms:

- **Repository** itself is sort of a container for code (oversimplified, but let’s stick to that for now) - the parallel to it that you’re likely the most familiar with is a project folder with code that you keep locally on your machine. Bear in mind the repository will usually have at least 2 instances – **remote** (the one hosted on whatever git service you might be using), and **local** (data kept locally on one or more computers)
- **Branch** is (as intuition might suggest) is a “path” the code is taking, if we were to imagine it as a road. The main code is kept and updated on “master” branch (also sometimes referred to as “main”), however the code development might *branch* out to several features (especially with multiple people working on it). On creation, new branches will snapshot the state of whatever branch they’re created from, and can then be updated independently – making it considerably easier to go back to “last working version” of code
- **Commit** is bits and pieces of code and files wrapped into a single package, usually accompanied by a message detailing the purpose of this package
- **Push** is what we do with a locally created commit, do put it on the remote repository
- **Pull** is what we do to update local code to latest version on remote repository
- **Pull request** is a request to **merge** changes from one branch to another (e.g., after finishing an experimental feature on a side-branch, you decide to incorporate it in your main code, and request to merge that branch with master/main). Often accompanied by **peer review**.
- **Peer review** is a review process during collaboration, where changes introduced in a pull request are reviewed by other collaborators to ensure the new additions are of adequate quality (and don’t break anything)
- **Merge conflicts** occur when a piece of code that a pull request attempts to change, has already changed on target branch (as compared to the source branch origin point) - these conflicts need to be resolved before merge, as it is not obvious which version of this code should be kept
- **Clone** is what we do to create a local version of target remote repository

Creating git repository on Github

Though Github is not the only option for free repository hosting (besides self-hosting), during this course we'll be using it instead of the alternatives due to the in-built integration with google colab, as well as keeping things simple and consistent between teams.

Task 1: Create a personal git repository following instructions in this section

- 1) Naturally, first you'll have to create an account on <https://github.com/>
- 2) Then you'll create a new repository using the website UI in top left corner

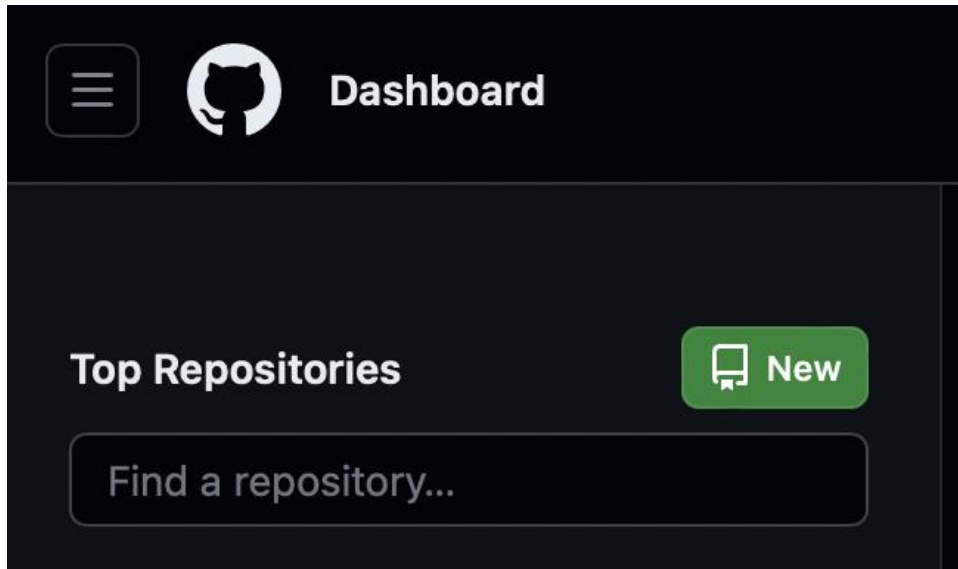


Figure 1: Github repositories dashboard

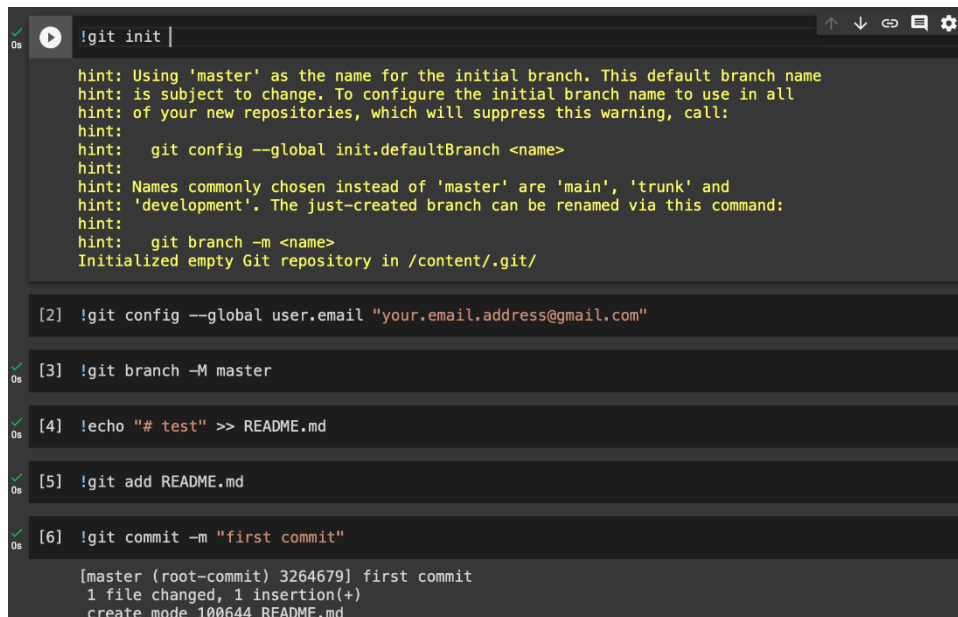
- 3) Fill in repository name, keep the repository public (overall more convenient to use, the course projects aren't exactly top-secret material)
- 4) Now that your repository is created, it's time to populate it. Several ways of doing that will get automatically displayed, however it'll be slightly more complicated in our case (which is sticking exclusively to colab, to guarantee that you can actually go through it regardless of what computer you're on, and what permissions you've got). We'll be utilizing this proposed instruction, albeit with some changes:

...or create a new repository on the command line

```
echo "# testclass" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:hesz94/testclass.git
git push -u origin main
```

Figure 2: Proposed repository creation from command line

5) Open a colab notebook and run following cells (replacing the email address with your own)



```
!git init |
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /content/.git/

[2] !git config --global user.email "your.email.address@gmail.com"

[3] !git branch -M master

[4] !echo "# test" >> README.md

[5] !git add README.md

[6] !git commit -m "first commit"

[master (root-commit) 3264679] first commit
1 file changed, 1 insertion(+)
create mode 100644 README.md
```

Figure 3: Creating repository via colab part 1 (some commands might take up to few minutes to execute, if you're not getting an error, wait patiently)

To quickly unpack what we just did, we:

- created an empty git repository on the colab runtime
- configured colab runtime git user
- switched the current branch to "master"
- created a README.md file with "# test" as its content
- added the created readme file to next commit
- created the commit with "first commit" message

On an unrelated note – as you may have noticed, you can run bash commands in colab by putting a "!" in front of them.

Now is the time where normally we'd set the repository target (to our previously created repository on github) and pushed the commit, however since we're working in an environment without a terminal, we need to sort out authentication first. ***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***

- 6) Since authentication via account password is no longer an option, we'll need to use "personal access token" (refer to <https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens> if you want to learn more about them)
 - a. On Github, go to profile (top right corner) -> settings -> developer settings -> personal access tokens -> tokens (classic)
 - b. Click "generate new token (classic)"
 - c. Fill in token description, tick "write:packages" (see Figure 4). Lastly, confirm token creation, and save the created token somewhere discreet

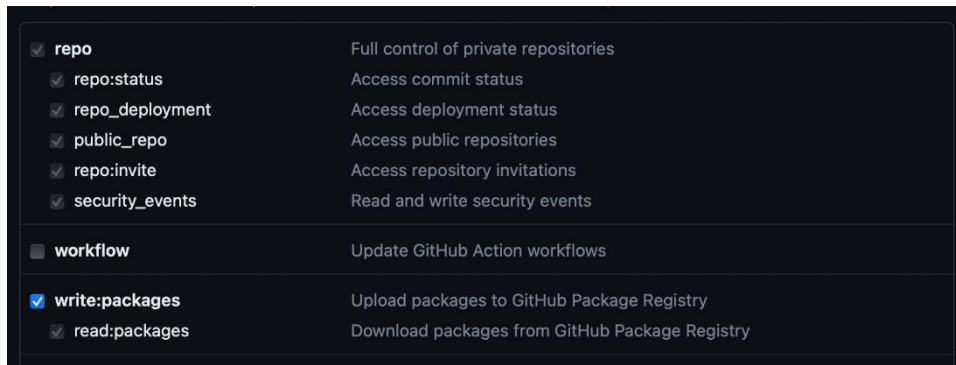


Figure 4: Personal access token permissions

7) Now that we have an authentication method, we can push our first commit, as follows:

```
[12] !git remote add origin git@github.com:your_username/your_repo_name.git
[13] !git push https://your_token@github.com/your_username/your_repo_name.git
```

Figure 5: Creating repository via colab part 2 – replace “your_username”, “your_repo_name” and “your_token” with appropriate text, the token being the one we just generated

8) After pushing, navigate to your repository page on github.com, it should look something like Figure 6

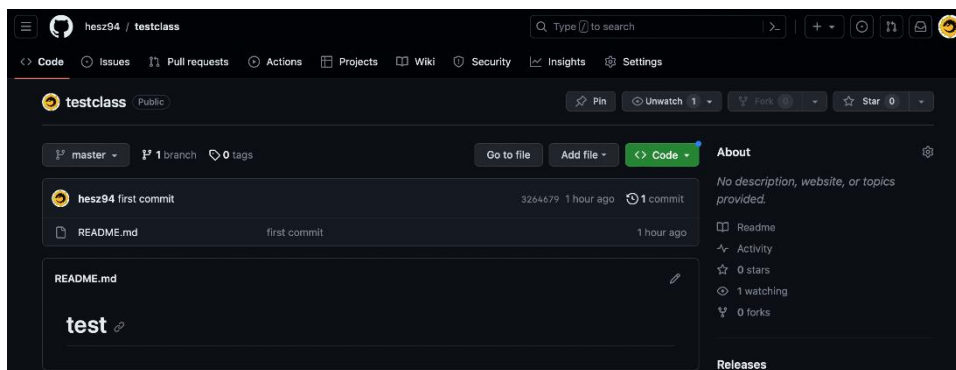


Figure 6: Sample repository dashboard

9) Now that the first commit (and branch) is there, you can use the website GUI to do most things – create new branches, create and resolve pull requests, review commits, etc.

10) Next, let’s link your colab and github – go to <https://colab.research.google.com/> and click “github” on the initial pop-up (you may need to enable additional pop-ups), and follow instructions on-screen to link your accounts

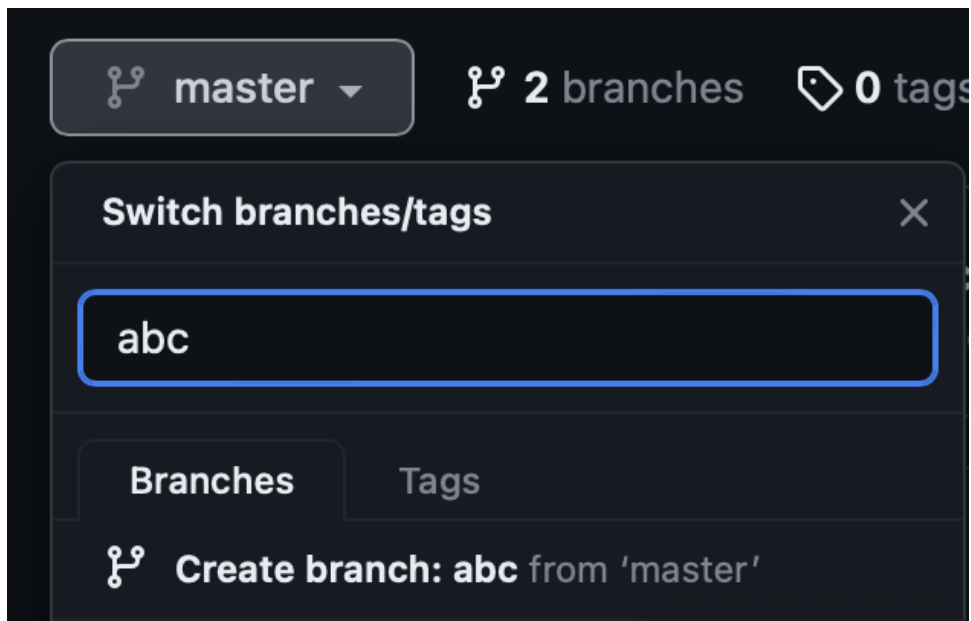


Figure 7: Creating new branch – the menu opens after clicking on a triangle in the “master” button.

11) From now on:

- a. if you want to commit your colab notebook to your github repository, simply go to File -> Save copy to Github, select the desired repository, branch, and filename, and commit
- b. whenever you start up a new notebook, you'll have the option of selecting one uploaded to your github

Task 2: Working with branches pt.1

- 1) Open an empty colab notebook, fill the first cell with code which creates a list with some content
- 2) Push the notebook to your master branch under “lab2task2.ipynb” name
- 3) Create two new branches via github GUI (see Figure 7), name them “for_variant” and “while_variant”
- 4) Fill the second cell with code that displays the content of the list using a for loop – push it to the “for_variant” branch
- 5) Fill the second cell with code that displays the content of the list using a for loop – push it to the “while_variant” branch
- 6) Close the notebook, and re-open both versions utilizing the github integration (start the way you did in point 10 of this section tutorial)
- 7) Pick one of these branches, and merge it to master via pull request (See section 3.b)

Collaboration with git

a. Adding collaborator

You're the sole owner of your newly created repository – to allow others to contribute to it, you have to add them as collaborators in the repository settings tab (see Figure) -> Access -> Collaborators -> Add people

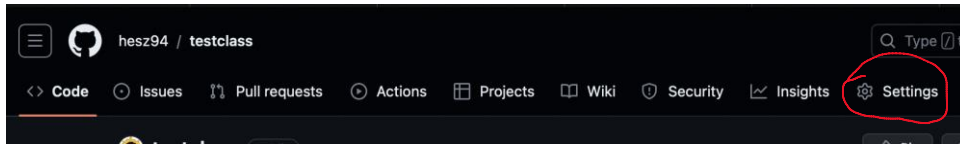


Figure 8: Repository settings tab location

b. Merging changes

Though solo projects (should) involve merging changes too, it's certainly more common among projects where multiple people work on different features in parallel, eventually putting the changes they introduced together to form the final result.

To merge changes you have to create a pull request in the “pull requests” tab (see Figure), which will summarize the changes between source (what you're merging) and target (where you're merging) branch. If everything is okay, you will get a green-light for merging and closing the pull request, however sometimes things will not be as simple (see 3.c)

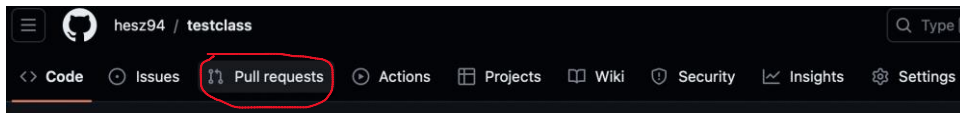


Figure 9: Pull requests tab location

c. Merge conflicts

Merge conflicts occur when some code that was changed on the source branch, has also already changed on target branch since the source branches creation – in such situation, it is not obvious which variant should be used, and this piece will be marked as conflicting. You can resolve merge conflicts on github web GUI editor, for details refer to:

<https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-on-github>

d. Reviews

Having more eyes on a pull request before merging it gives better odds at avoiding errors, as well as keeping the code clean and functional, though it does introduce additional friction to the process. This process is formalized as pull request reviews. After a pull request is created, but before it's merged, all collaborators can view the list of changes introduced by this potential merge, and comment on them – either giving approval, or requesting changes. To limit haphazard merges that might break more things

than they fix, it is possible to set a requirement of having at least one persons approval before having the ability to merge.

Though it may seem like a waste of time at this stage, code reviews (at least to some degree) bring many benefits, and beyond the obvious “cleaner/better code”, it also ensures higher familiarity with the codebase amongst the collaborators. After all, to give your teammates code a thorough review, you first have to read and understand what changes they’re proposing in the first place.

Task 3: Working with branches pt.2 (team task)

1. Create a team repository in the same manner as the individual ones
2. Give team members read/write access
3. Create a “lab2task3” branch
4. Push a notebook with following cells to task3 branch:
 - a. Cells 1, 2, 3, 4 containing an empty function definition (function with a given name, no arguments, and only “return” in body)
 - b. Cell 5 calling these 4 functions one after another
5. Create an individual branch for each team member from lab2task3 branch
6. Every member should edit one of these empty functions (on their computer, coordinate with each other on who does which one) to print their name. Note that you’re only defining a function that will print that, don’t call the function. Push the changes to your personal branch, make sure the notebook name stays the same.
7. Merge all these individual branches to lab2task3
8. Open the latest notebook from lab2task3 and execute all cells, this should result in 5th cell printing all team members names

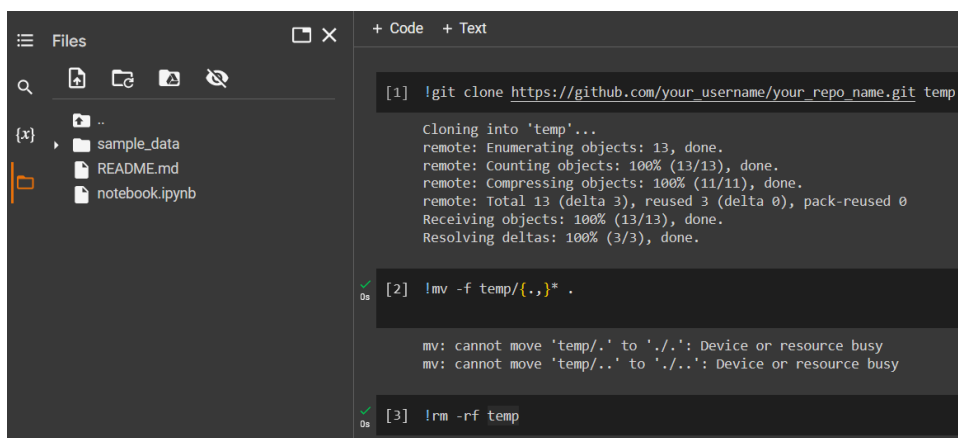
Task 4: Merge conflicts

1. Create a new notebook
2. In first cell define an empty function called “celebrate” that takes no arguments, and an if block with always – false condition, inside which this empty function is called (e.g. if False)
3. Push the notebook to master branch
4. Create new branch
5. Fix the condition to an always – true one (e.g. if True), push the fixed condition to master branch
6. Switch to branch created in subtask 4 (Close the notebook, open it through git interface from a new branch), edit the “celebrate” function to print “woooo!” or another celebratory message
7. Merge the new branch to master, resolve the conflict in such a way, that after running the cell the celebratory message is printed

Inclusion of non-notebook files in repository and handling it within colab.

Naturally, you might want to include non-notebook (.ipynb) files in your repository as well – in particular .py files where you could put all written convenience functions, and keep them out of notebooks for easier maintenance. In general – besides the inconvenience of colab, .py files will be much easier to work with and monitor changes on when it comes to github. To work with repository in this way in colab, we need to start from loading it differently, as the colab-github integration only allows for loading up notebooks.

For that purpose, launch a new unrelated notebook, and in the first cell simply clone your repository, which will download the remote repository to current colab runtime – letting you view and interact with files via the colab file explorer (see Figure 10). Sadly, the colab-github integration leaves a bit to be desired, so we'll have to do some extra steps to get things working.



```
[1] !git clone https://github.com/your_username/your_repo_name.git temp

Cloning into 'temp'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 13 (delta 3), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (13/13), done.
Resolving deltas: 100% (3/3), done.

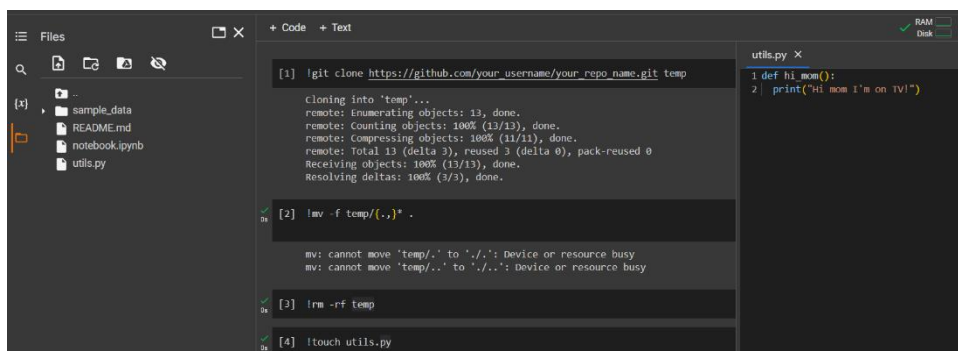
[2] !mv -f temp/{,}* .

mv: cannot move 'temp/.' to './.': Device or resource busy
mv: cannot move 'temp/..' to './.': Device or resource busy

[3] !rm -rf temp
```

Figure 10: Cloning repository into temp directory, moving its contents into current workdir, removing temp directory (as before, remember to replace “your_username” and “your_repo_name”)

Now, we'll create a .py file in this notebook as well, making sure to place it in the actual repository.



```
[1] !git clone https://github.com/your_username/your_repo_name.git temp

Cloning into 'temp'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 13 (delta 3), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (13/13), done.
Resolving deltas: 100% (3/3), done.

[2] !mv -f temp/{,}* .

mv: cannot move 'temp/.' to './.': Device or resource busy
mv: cannot move 'temp/..' to './.': Device or resource busy

[3] !rm -rf temp

[4] !touch utils.py
```

utils.py X

```
1 def hi_mon():
2 | print("Hi mom I'm on TV!")
```

Figure 11: Creating and editing files in colab

Next, to edit the .py file, we simply double click the text file, and an edit window will pop out on the right.

Lastly, after saving the file, we can push it to the repository in the same manner in which we pushed the README.md file in initial push. Remember to change the current branch if need be (“git branch branchname”, by default after cloning we start in “master”).

Now that this utils.py file is created, all notebooks in this directory can import its functions, as they would from a library (see Figure 12)

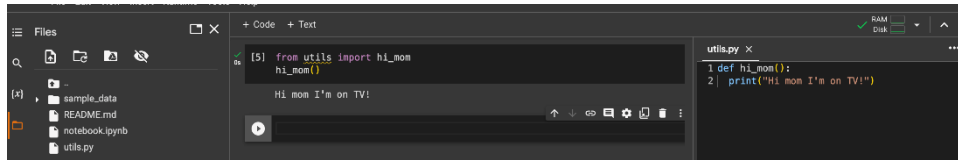


Figure 12: Using function imported from local .py file

Task 5: Using .py files

1. Create (and push to repository) a .py file with a utility function that checks whether a triangle with side lengths a, b, c (input parameters) can be constructed, returning True / False based on that condition
2. Import that function in a notebook
3. Create 3 arrays of 100 random numbers (or a 2-dimensional 100x3 matrix), and check how many of these 3s can be used to construct a valid triangle, using the imported function
4. Push both notebook and .py file to your repository