

# Git & Modularization

Bálint Aradi

Course: Scientific Programming / Wissenschaftliches Programmieren (Python)

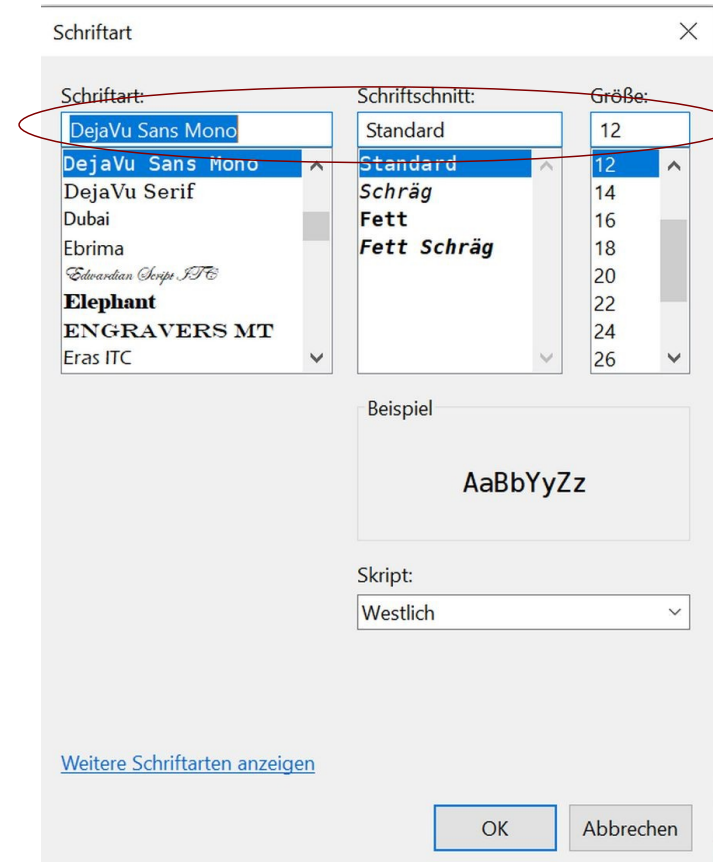


<https://www.bccms.uni-bremen.de/people/b-aradi/wissen-progr/python/2023>

- Installing the prerequisites
- First steps with the version control system Git
- Creating modules

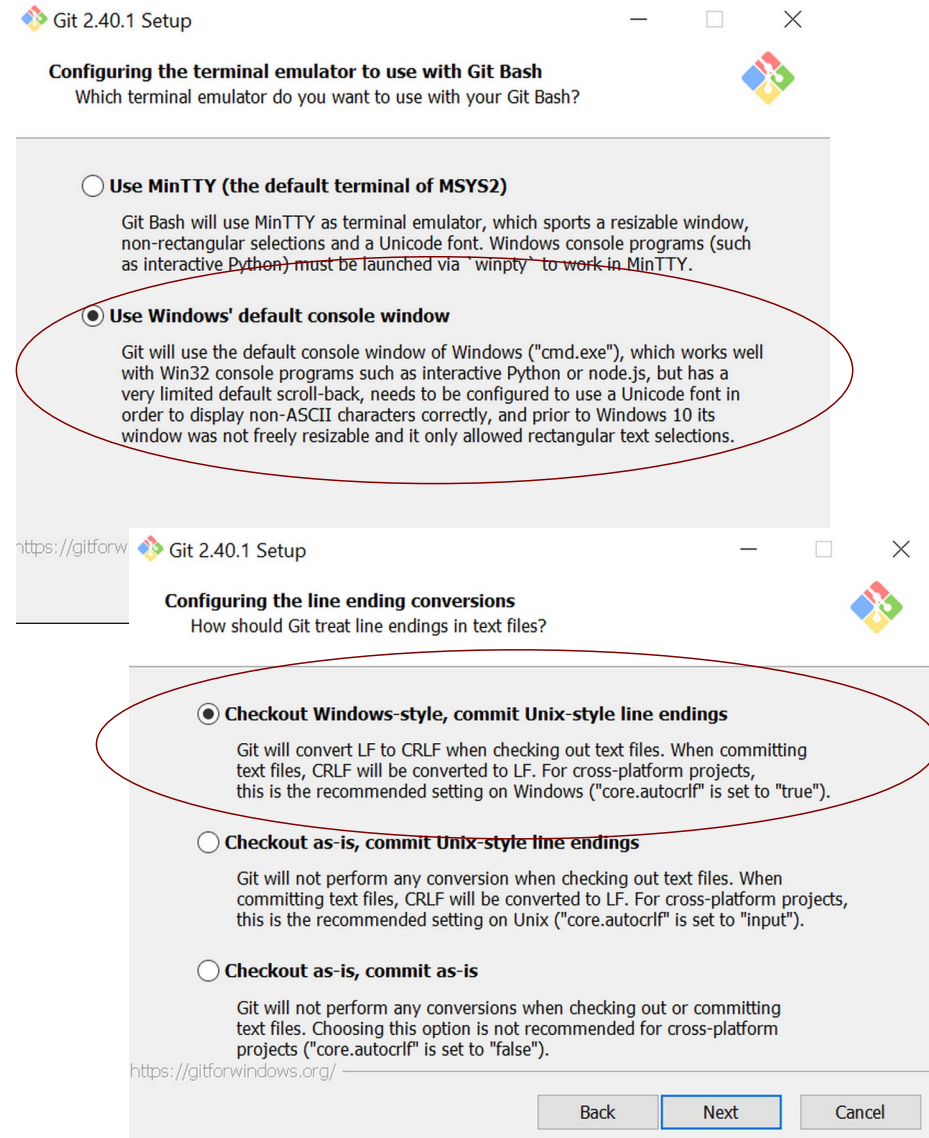
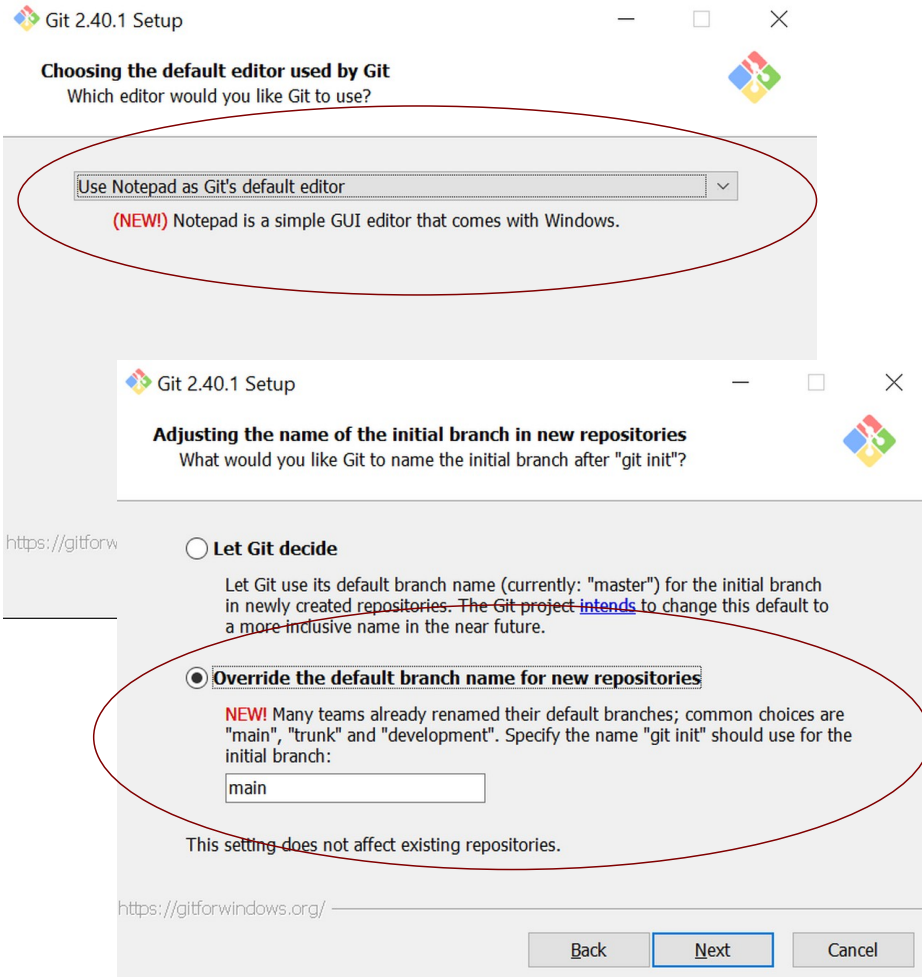
# Prerequisites - simple text editor (Windows)

- We will use **Notepad** to enter git commit messages
- Configure Notepad to use a **mono-spaced** font (e.g. DejaVu Sans Mono)



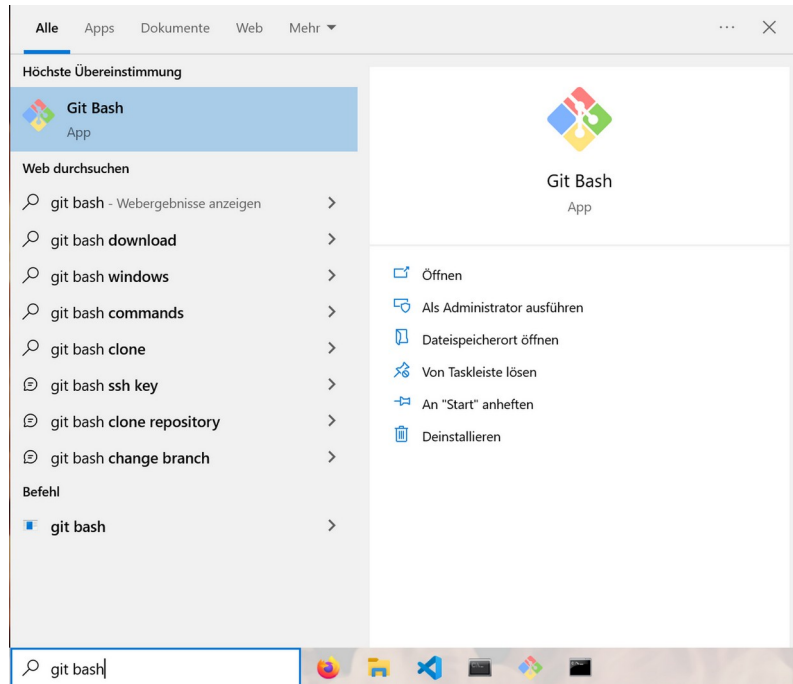
# Prerequisites - Git (Windows)

- Download and install [Git for Windows](#)
- Take default options apart of following ones:



# Prerequisites - Git (Windows)

- We will use the Git Bash application (part of Git for Windows) to communicate with Git



- You can even configure Git Bash to access Conda by creating a `~/.bashrc` file with following content:

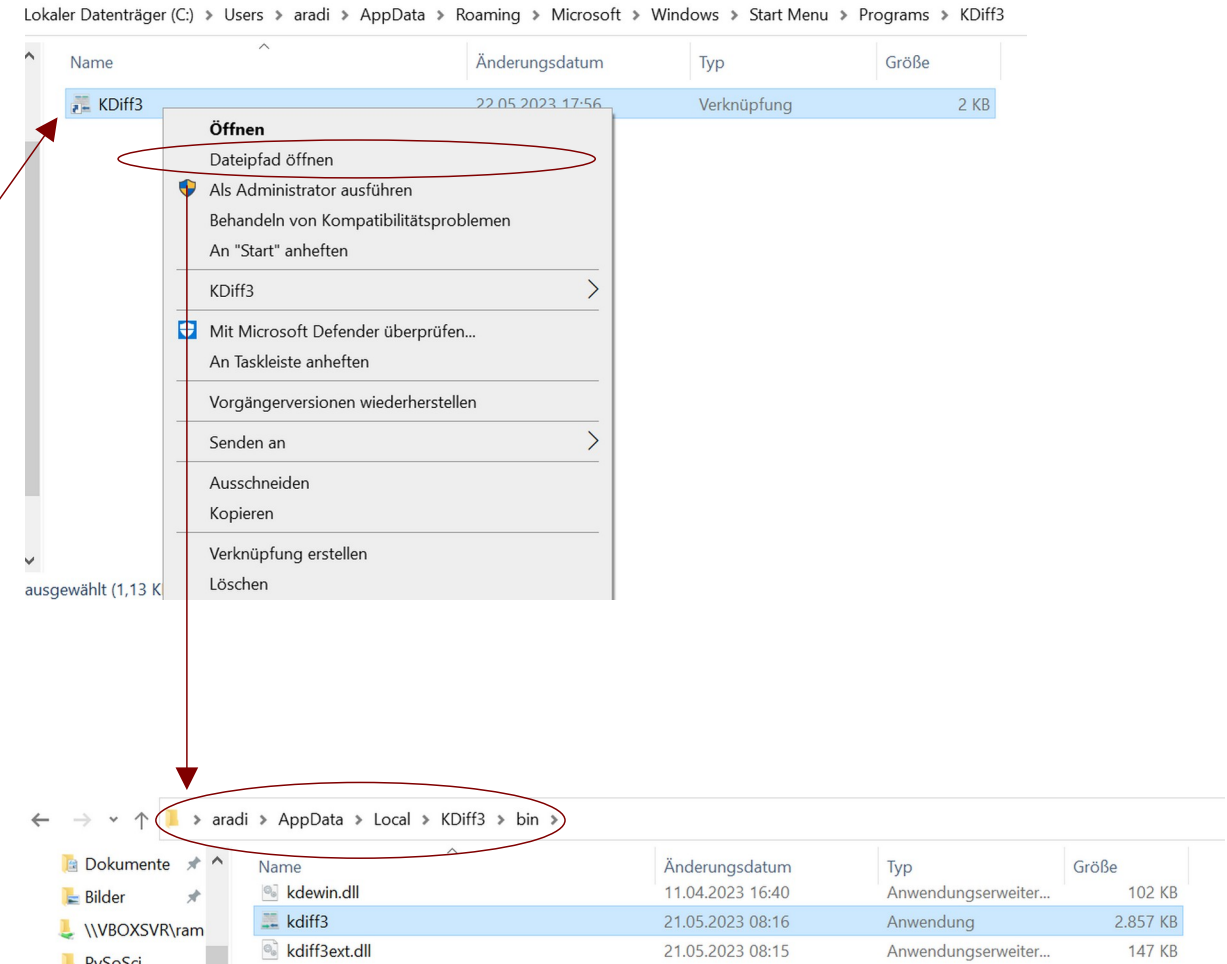
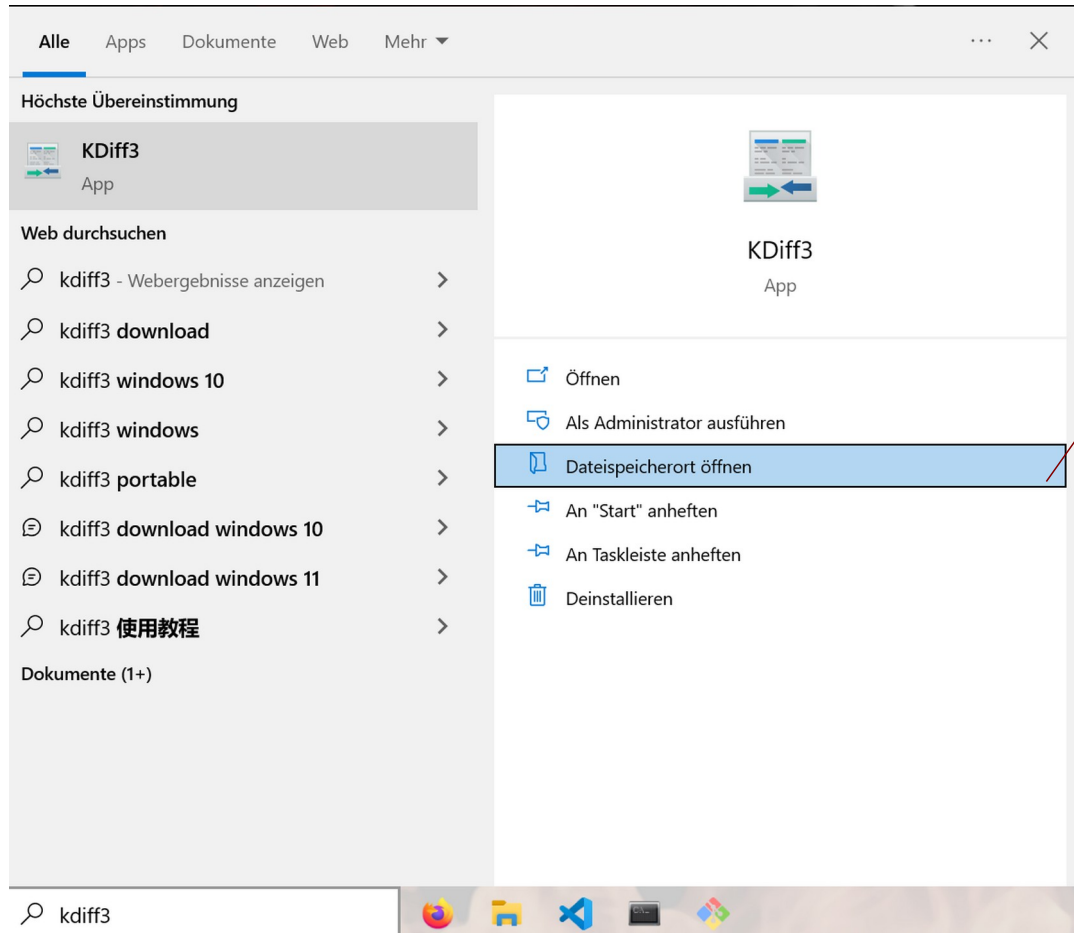
`~/.bashrc`

```
source ~/miniconda3/etc/profile.d/conda.sh
```

This must match the path where Miniconda/Conda had been installed into

# Prerequisites - diff-viewer (Windows)

- Download and install a suitable diff-viewer for Windows ([KDiff3](#) or [Meld](#))
- We will need the location, where the executable can be found

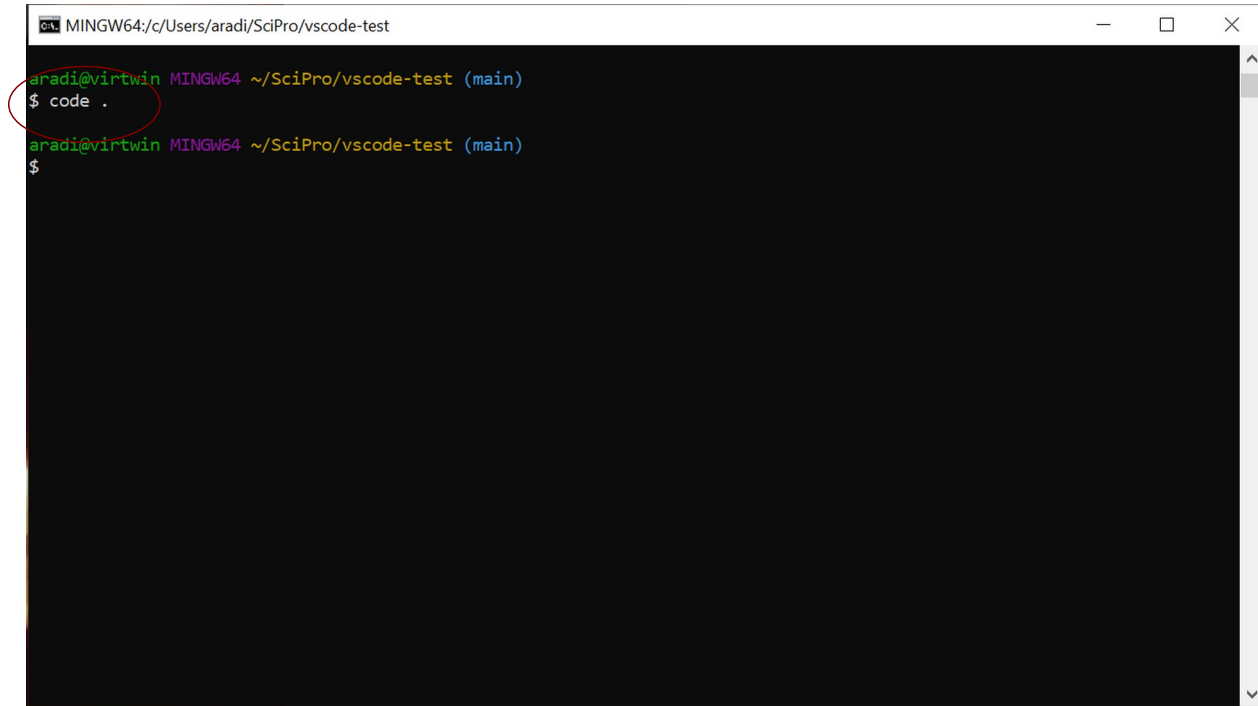
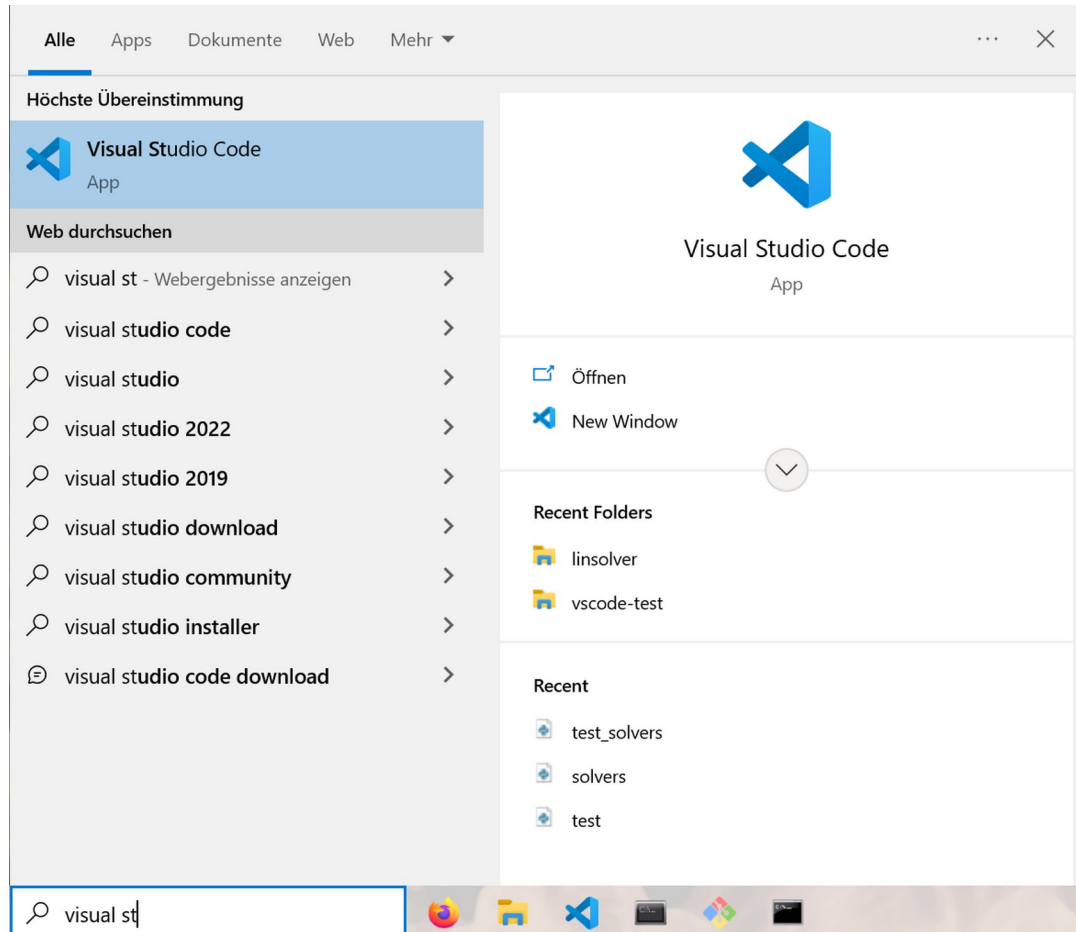


## Prerequisites - Git & Git related tools (Linux)

- Install **git** via the package manager of your Linux distribution
- Install also a git-viewer package: **qgit** or **gitk**
- Install a diff-viewer tool: **kdifff3 (kdifff3-qt)** or **meld**
- Make sure, that a simple text editor (**featherpad**, **pico**, **nano**, etc.) is installed

# Prerequisites - Visual Studio Code

- Download and install [Visual Studio Code](#) for your operating system
- You can start Visual Studio Code via menu or by entering “code” in the Git Bash shell



- See [Getting started with Visual Studio Code](#) for a quick introduction



## linsolver

- Program package for solving linear system of equation
- It should offer the Gaussian-elimination method (LU-decomposition)
- It should **read data** either from file or from console and write results to file or to the console
- It should have an **automatic test framework** for unit tests
- It should be well **documented** and **cleanly written**.

**Note:** This project serves **didactical purposes only**, the optimized routines of SciPy should be usually used to solve a linear system of equations.

# Let's start to develop!

## Create the project folder

- Open a konsole (Linux, Mac) / Git Bash (Win) window
- Make the (new) directory (folder) "SciPro"

```
mkdir SciPro
```

- Change to the directory "SciPro"

```
cd SciPro
```

- Make the (new) directory "linsolver"

```
mkdir linsolver
```

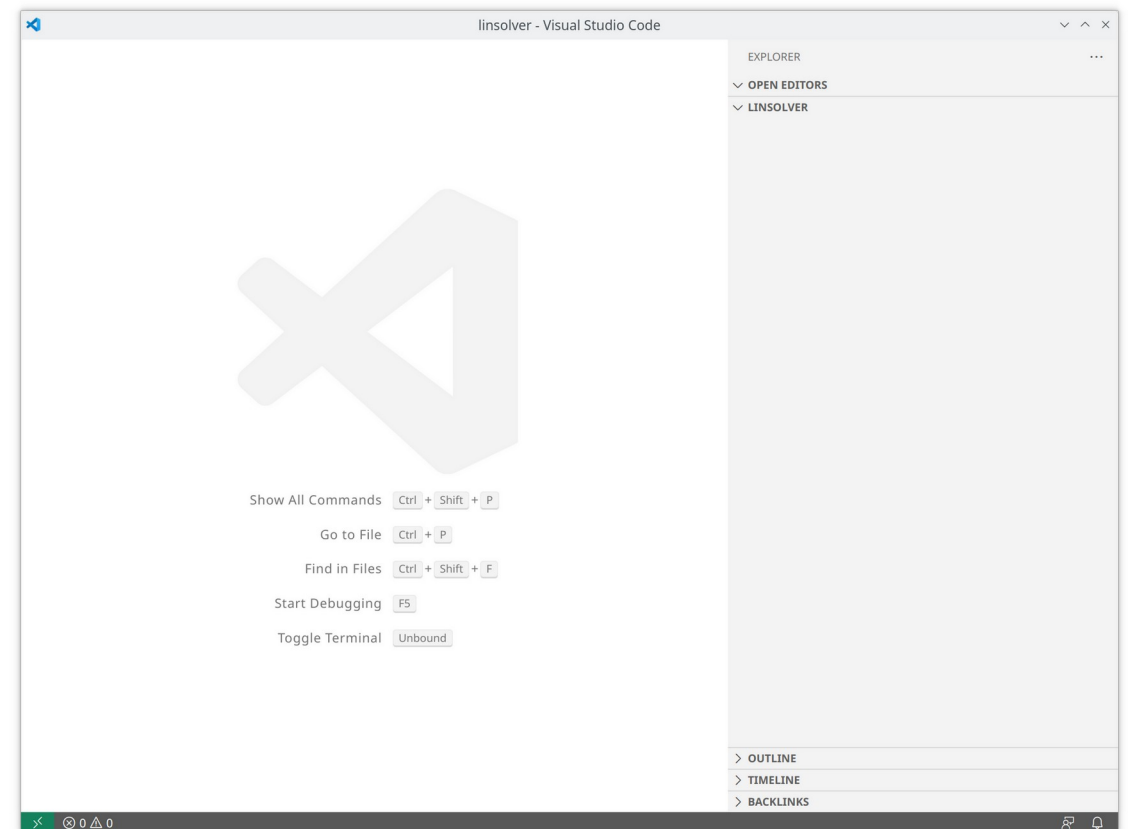
- Change to the directory "linsolver"

```
cd linsolver
```

## Start VS Code from the project folder

```
code .
```

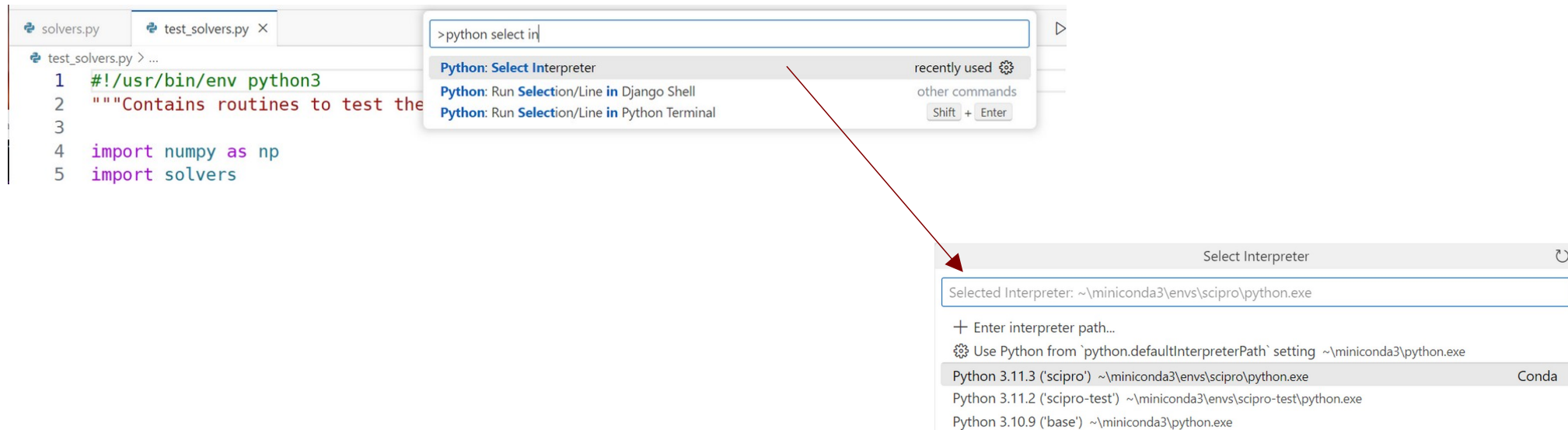
Pass the current directory as argument



(your editors appearance might differ slightly)

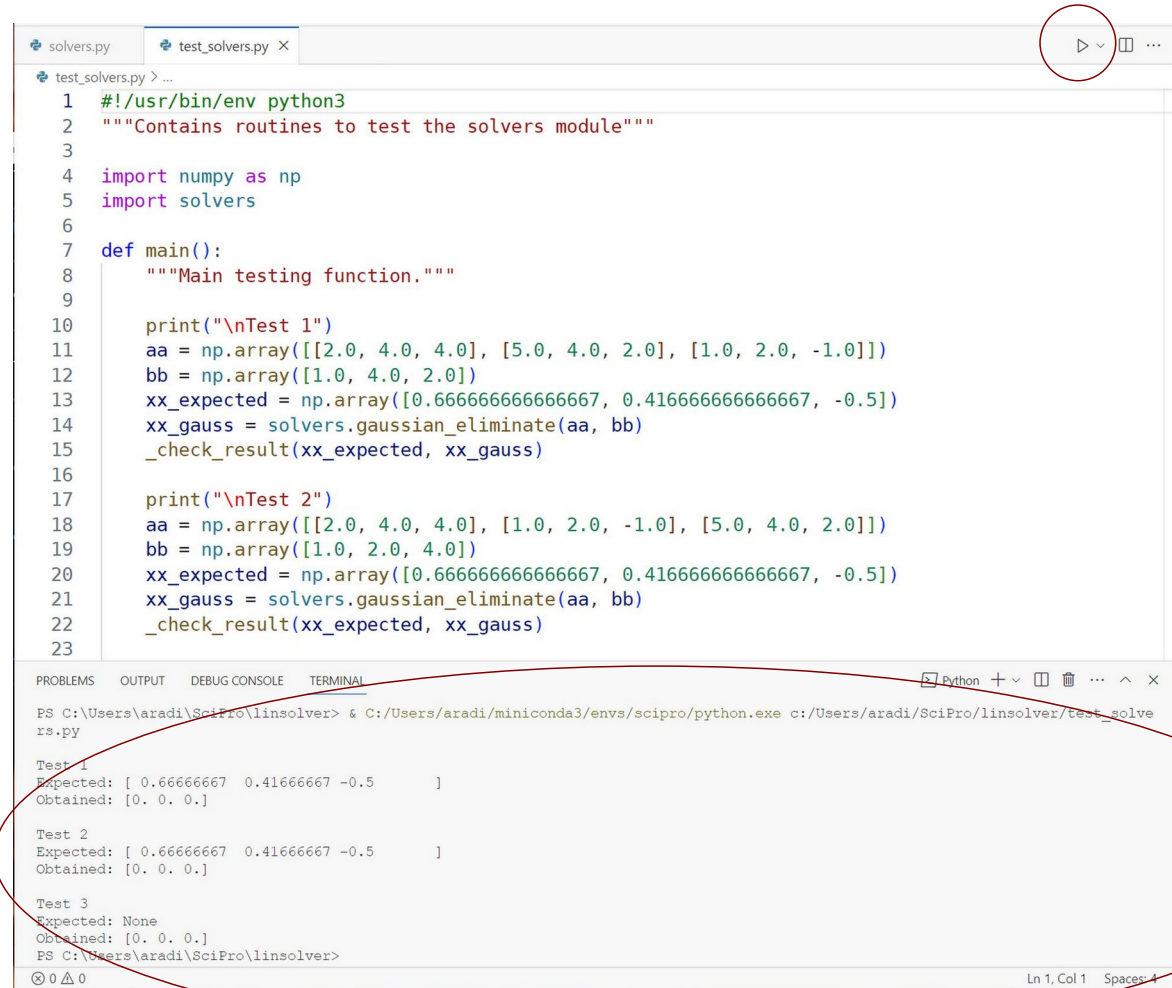
# Let's start to develop!

- Create two new files (Ctrl-N) and save them (Ctrl-S) as **solvers.py** and **test\_solvers.py**, resp.
- Download the content of the following two files and paste them into the empty files (or copy them into the project folder)
- Select the Python Interpreter from your Conda environment (**Ctrl-Shift P** opens the command palette)



# Let's start to develop!

- Run “test\_solvers.py” from within your IDE



The screenshot shows an IDE window with two tabs: 'solvers.py' and 'test\_solvers.py'. The 'test\_solvers.py' tab is active, displaying the following Python code:

```
1 #!/usr/bin/env python3
2 """Contains routines to test the solvers module"""
3
4 import numpy as np
5 import solvers
6
7 def main():
8     """Main testing function."""
9
10    print("\nTest 1")
11    aa = np.array([[2.0, 4.0, 4.0], [5.0, 4.0, 2.0], [1.0, 2.0, -1.0]])
12    bb = np.array([1.0, 4.0, 2.0])
13    xx_expected = np.array([0.666666666666667, 0.416666666666667, -0.5])
14    xx_gauss = solvers.gaussian_eliminate(aa, bb)
15    _check_result(xx_expected, xx_gauss)
16
17    print("\nTest 2")
18    aa = np.array([[2.0, 4.0, 4.0], [1.0, 2.0, -1.0], [5.0, 4.0, 2.0]])
19    bb = np.array([1.0, 2.0, 4.0])
20    xx_expected = np.array([0.666666666666667, 0.416666666666667, -0.5])
21    xx_gauss = solvers.gaussian_eliminate(aa, bb)
22    _check_result(xx_expected, xx_gauss)
23
```

Below the code editor is a terminal window with the following output:

```
PS C:\Users\aradi\SciPro\linsolver> & C:/Users/aradi/miniconda3/envs/scipro/python.exe c:/Users/aradi/SciPro/linsolver/test_solve
rs.py
Test 1
Expected: [ 0.66666667  0.41666667 -0.5      ]
Obtained: [0. 0. 0.]
Test 2
Expected: [ 0.66666667  0.41666667 -0.5      ]
Obtained: [0. 0. 0.]
Test 3
Expected: None
Obtained: [0. 0. 0.]
PS C:\Users\aradi\SciPro\linsolver>
```

The terminal output is circled in red. The IDE interface also shows a red circle around the run button in the top right corner of the code editor.

Terminal output

# Let's start to develop!

- Run “test\_solvers.py” from the command line  
(in a command line window, where Conda had been already activated)

```
python test_solvers.py
```

```
python3 test_solvers.py
```

```
(scipro)
aradi@virtwin MINGW64 ~/SciPro/linsolver
$ python test_solvers.py

Test 1
Expected: [ 0.66666667  0.41666667 -0.5      ]
Obtained: [0. 0. 0.]

Test 2
Expected: [ 0.66666667  0.41666667 -0.5      ]
Obtained: [0. 0. 0.]

Test 3
Expected: None
Obtained: [0. 0. 0.]
(scipro)
aradi@virtwin MINGW64 ~/SciPro/linsolver
$
```

The project apparently needs some development ...

- Before you change anything, the project should be set under **version control**

# Typical scenario with version control

## Scenario

- New project is started
- Program tested, everything works OK
- New functionality is added
- **Suddenly, something does not work as supposed, although it was working before** (note: testing framework apparently not satisfactory)

## Solution work-flow with version control

- **Go back in history** to the last revision (evtl. by bisection), until a correctly working version is found
- **Inspect the changes** introduced in the snapshot (commit) and find out the reason for the failure
- **Fix the bug** in the most recent program version

## Main tasks

- Document **development history** (store snapshots of the project)
- Help **coordinating multiple developers** working on the same project
- Help **coordinating** development of **multiple versions** of a project

## Centralized VC (CVS, Subversion, ...)

- Central server stores history database
- Developer must have connection to the server for most operations (especially for commits, checkouts or browsing history).

## Distributed VC (Git, Mercurial, ...)

- Every developer has a **local copy of the full development history**
- Most operations do not require network connection (except synchronization between developers)

# Introduce yourself to git

- Enter your name and email address (needed for the logs)

```
git config --global user.name "Bálint Aradi"  
git config --global user.email "aradi@uni-bremen.de"
```

- Specify standard tools to be used

```
git config --global core.editor featherpad Use "notepad" on Windows  
git config --global diff.tool kdiff3  
git config --global merge.tool kdiff3 On Windows you need to specify the path  
git config --global difftool.kdiff3.path /c/Users/[...]/kdiff3.exe
```

- **--global** stores option globally, otherwise they apply to current project only
- Global options are stored in the **~/.gitconfig** file

- Current options can be listed with **--list**

```
git config --list
```



# Create a repository

- Initialize a repository in the project directory

```
cd ~/SciPro/linsolver/
```

```
git init
```

```
Initialized empty Git repository in C:/Users/aradi/SciPro/linsolver/.git/
```

- **Files within the project directory** can be placed under version control
- Files within the `.git` directory should not be changed manually
- When copying project directory recursively (including the `.git` subdirectory) the entire revision history is copied

# Put files under version control

```
git status
```

```
On branch main
```

```
No commits yet
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
  __pycache__/
```

```
  solvers.py
```

```
  test_solvers.py
```

```
nothing added to commit but untracked files present
```

```
(use "git add" to track)
```

# Put files under version control

```
git add solvers.py test_solvers.py ←
git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   solvers.py
    new file:   test_solvers.py

Untracked files:
  (use "git add <file>..." to include [...])
    __pycache__/
```

- Puts files under version control and makes a snapshot of their current state (**stage**)
- Staged files are written to the database at the next commit

# Ignoring files

- Files that should not be version controlled can be listed in `.gitignore` in the project directory

```
featherpad .gitignore
git add .gitignore
git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .gitignore
    new file:   solvers.py
    new file:   test_solvers.py
```

→ `__pycache__`

- The `.gitignore` file should be also placed under version control

# Commit staged files

- When commit is issued, staged files (in their staged state) are written to the database

```
git commit ←
```

```
[main (root-commit) 5270fa1] Kick off project  
3 files changed, 58 insertions(+)  
create mode 100644 .gitignore  
create mode 100644 solvers.py  
create mode 100644 test_solvers.py
```

Opens editor

Write log message  
("Kick off project"),  
save & exit

```
git status
```

```
On branch main  
nothing to commit, working tree clean
```

# Checking project history

- Show project history:

```
git log
commit 5270fa191b5cbe7a83e4b1e3d406c37793e4b27a (HEAD -> main)
Author: Bálint Aradi <aradi@uni-bremen.de>
Date:    ...

    Kick off project
```

- Individual commits are identified by **hash checksums**
- Checksums can be shortened as long as they are unambiguous
- **--oneline** option gives a short summary of the log messages (shows also shortened checksums)

```
git log --oneline
5270fa1 (HEAD -> main) Kick off project
```

# Checking project history

- Revision history and log messages are shown in **reverse time order**

```
commit 2a3186299e14575a40b870cc3f8eb21c1e886809
Author: Bálint Aradi <aradi@uni-bremen.de>
Date:    ... [earlier]
```

Add readme file

```
commit 04d386638495386aa29ee99e4928aad2e7731f39
Author: Bálint Aradi <aradi@uni-bremen.de>
Date:    ... [later]
```

Add first stub files

- If history is longer than a page, it is shown page-wise via the **default pager** (e.g. less)

Navigation: **[Page Up/Down]** Move up/down  
**q** Quit

# Git-workflow

- Set up git global for your Unix account

```
git config --global ...
```

- Set up the repository for your project

```
git init
```

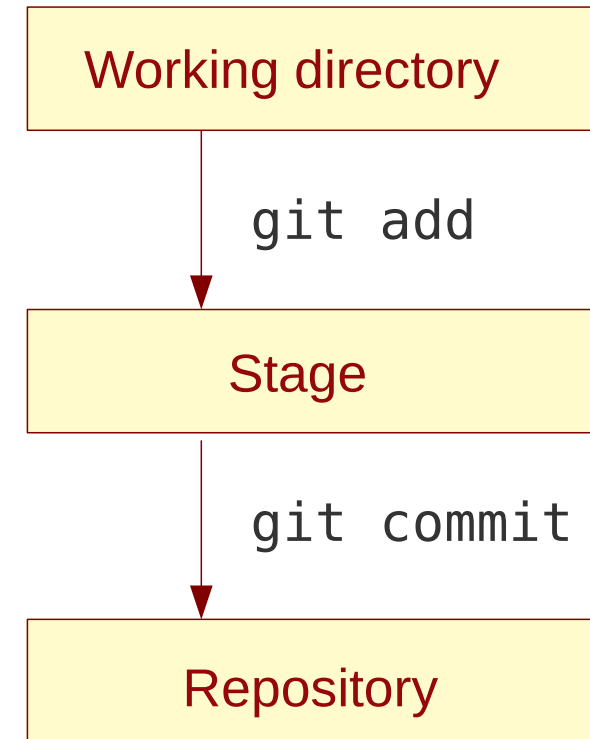
- Edit files in your project

- Stage files / changes

```
git add ...
```

- Commit staged changes into repository

```
git commit ...
```



- It is possible to stage all changes in all files which are already under version control:

```
git add -u
```



## Some git remarks

- Changes should be committed, if implementation of a feature is finished
- Development history should be easy to follow based on the log messages
- Changes within a commit should be small enough so that a developer can easily follow and understand them.
- Log messages should contain a **short sentence** (max. 50-60 chars), **optionally followed by an empty line and a more detailed description.**  
(See for example: [How to Write a Git Commit Message](#))

```
Implement LU-decomposition with back substitution
```

```
LU-decomposition is implemented without permutation. Check  
for linear dependency is not implemented yet.
```

- Short (one-liner) log messages can be passed on the command line

```
git commit -m "Add first stub files"
```

# Python module

- File containing routines, constants etc. which can be **used by other Python scripts**.
- Modules enable **logical structuring and reusability**

solvers.py

```
"""Routines for solving a linear system of equations."""
import numpy as np

def gauss_eliminate(aa, bb):
    """ ...
    """
    print("Linsolve: received aa:", aa)
    print("Linsolve: received bb:", bb)
    xx = np.zeros((len(bb),), dtype=float)
    return xx
```

Function within a module

Doc-string documenting the module

# Using a module

- Modules can be imported by the **import** command

```
import solvers
```

- The module content can be accessed by the **dot-notation**

```
xx_gauss = solvers.gauss_eliminate(aa, bb)
```

- At import Python **looks up** following places for the module:
  - Local directory
  - Directories contained in the **PYTHONPATH** environment variable
  - Package directories of the Python distribution
- The **PYTHONPATH** environment variable can be set for the current BASH shell (or in `~/.bashrc` if it should be always set):

```
export PYTHONPATH="/.../some_directory"
```

# Executable Python script

- When a Python script is run / a Python module is imported: all commands in it are executed
- In order to make also Python scripts importable, the commands to be executed should be placed into a function (usually called **main()**)
- Python's internal `__name__` variable can be used to check, whether the code is executed as a standalone script (or imported as a module)

`test_solvers.py`

```
def main():
    """Main script functionality."""
    :

if __name__ == '__main__':
    main()
```



**Have fun!**