6 – Git & Modularization

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

https://www.bccms.uni-bremen.de/cms/people/b-aradi/wissen-progr/python/2022

Prerequisites

Additional programs needed:

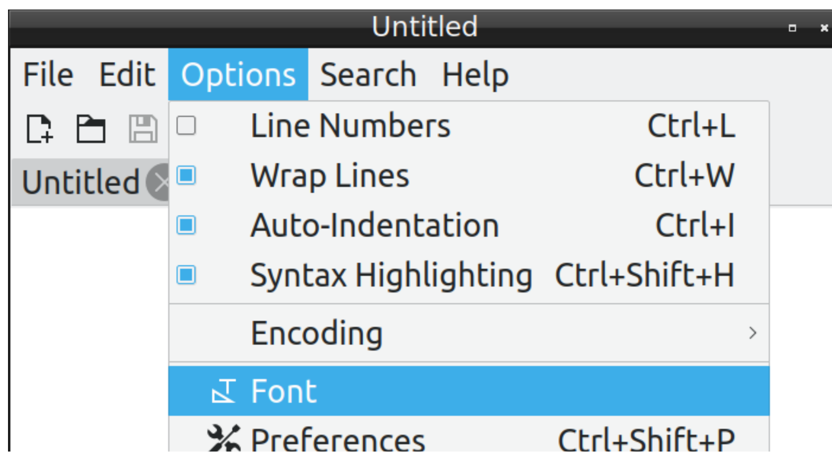
- Spyder3, Pylint3
- Git, QGit
- KDiff3 (non-KDE (qt-only) version)

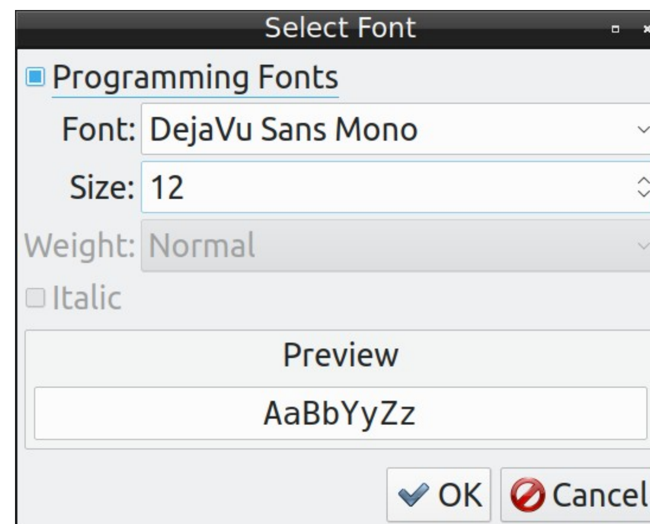
```
sudo apt install spyder3 git qgit kdiff3-qt
```

Recommendation:

- Set up your default editor (leafpad) to use monospaced fonts

```
featherpad
```





Programming project (for lectures)

linsolver

- Program package for solving linear system of equation
- It should offer Gaussian-elimination and LU-decomposition methods
- 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**.

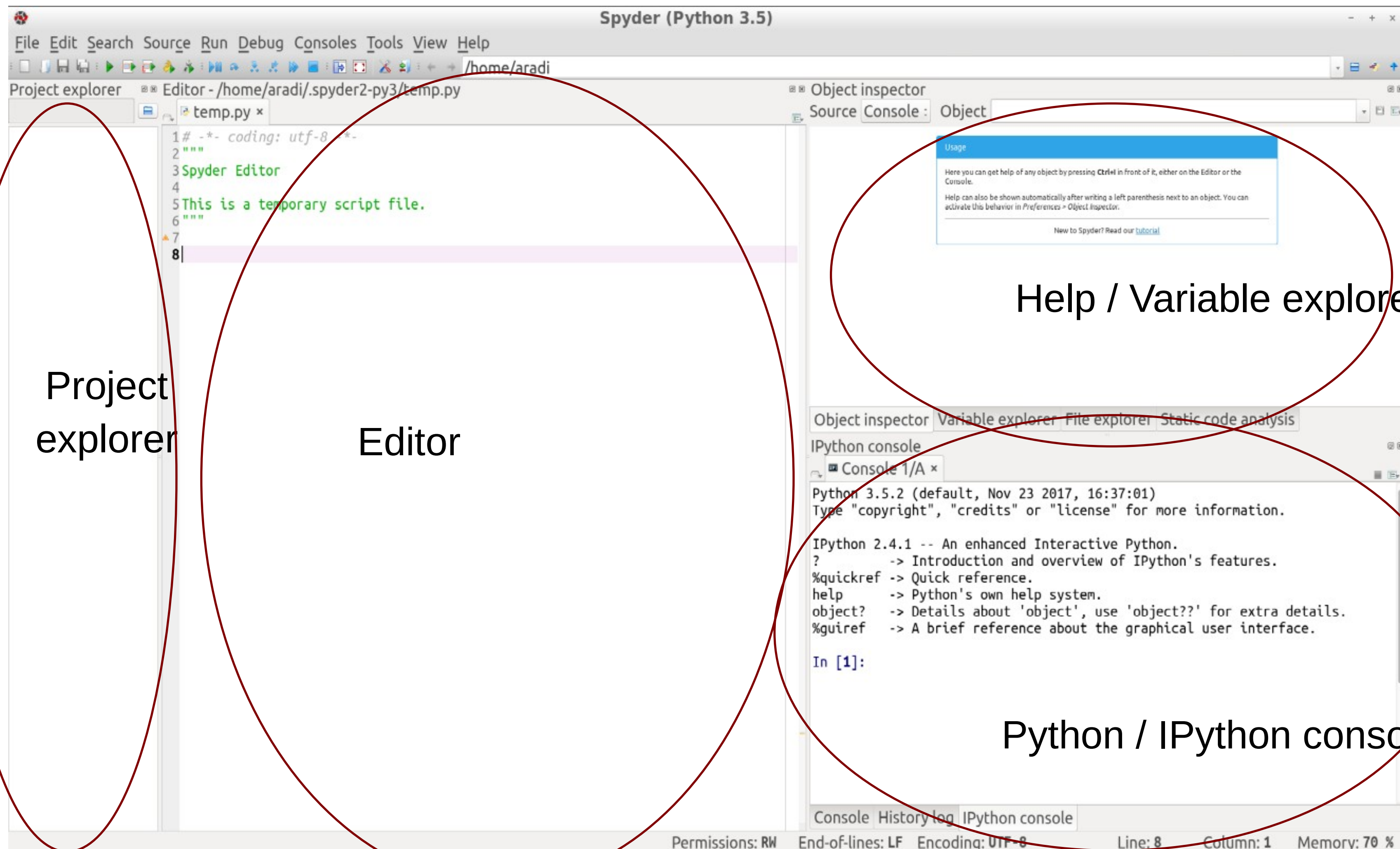
Note: This project serves didactical purposes only, the optimized routines of SciPy should be usually used to solve a linear system of equations.

Spyder3 editor

- Enables easy development of large scientific Python **projects**

```
spyder3 &
```

Start application in background and return command prompt immediately



Go through the **tutorial**: Help / Spyder tutorial

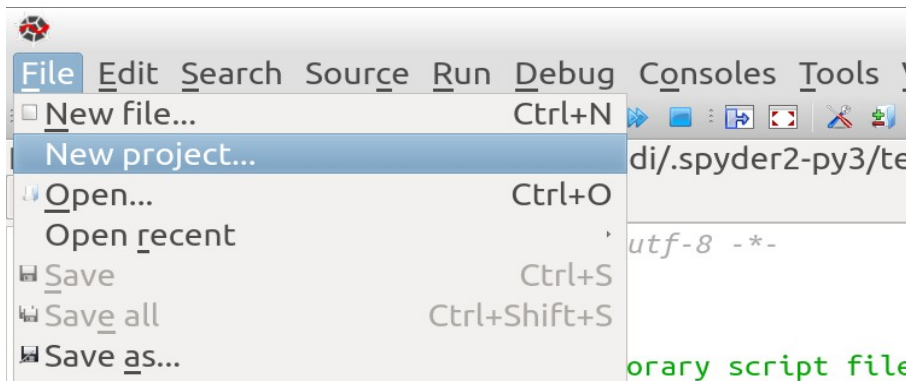
Create a new project

- Create a directory for all your projects (optional)

```
mkdir ~/projects
```

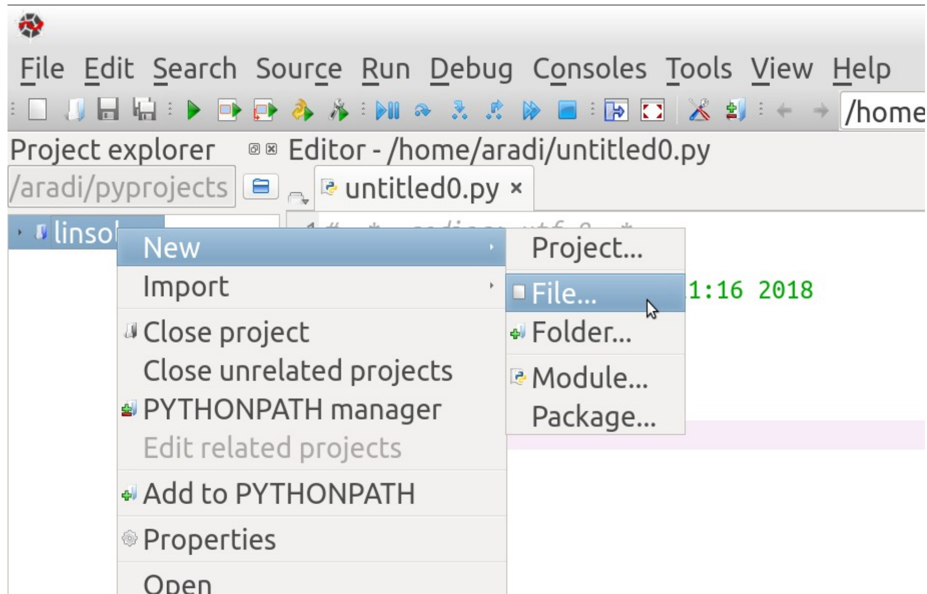
Directory for all Python-projects

- Create the Python project linsolver with Spyder (select ~/projects/linsolver as target directory):



Add files to the project

- Add the new files **solvers.py** and **test_solvers.py** to your project

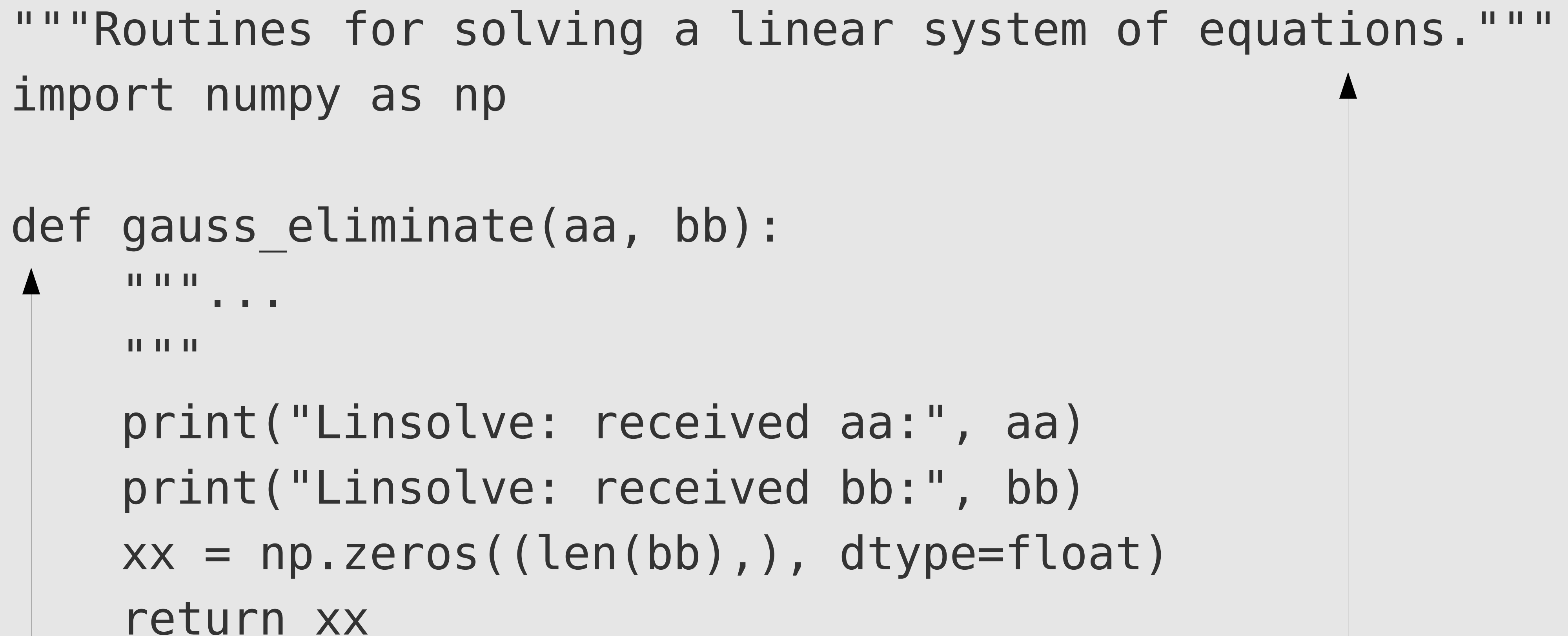


- Download and copy & paste the following content to your files:
  - solvers.py
  - test_solvers.py
- Make sure you save both files (**Ctrl-S**)

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
```

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:
  - 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 **.bashrc** if it should be always set):

```
export PYTHONPATH=/home/.../some_directory
```

Python executables

- When a python script is run all Python commands in it are executed
- In order to make all such Python scripts importable, the commands to be executed should be placed into a function (usually called **main()**)
- Pythons internal __**name**__ variable can be used to check, whether the script is executed as standalone script (otherwise imported as module)

test_solvers.py

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

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

Execute a python script

- A Python script can be executed from the current shell by starting the Python-interpreter and passing the script as first argument:

```
python3 test_solvers.py
```

- If the first line of a script contains a **special comment** starting with **#!**, the shell automatically calls the specified interpreter and passes the script content to the interpreter when the script is executed directly.

- The executable attribute has be set in order to execute the script directly

test_solvers.py

```
#!/usr/bin/python3
:
```

or

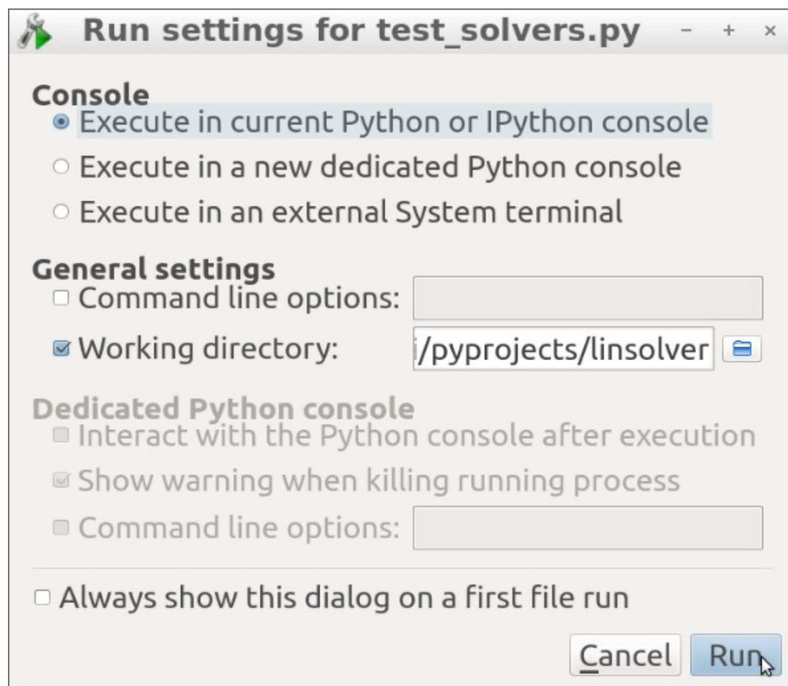
test_solvers.py

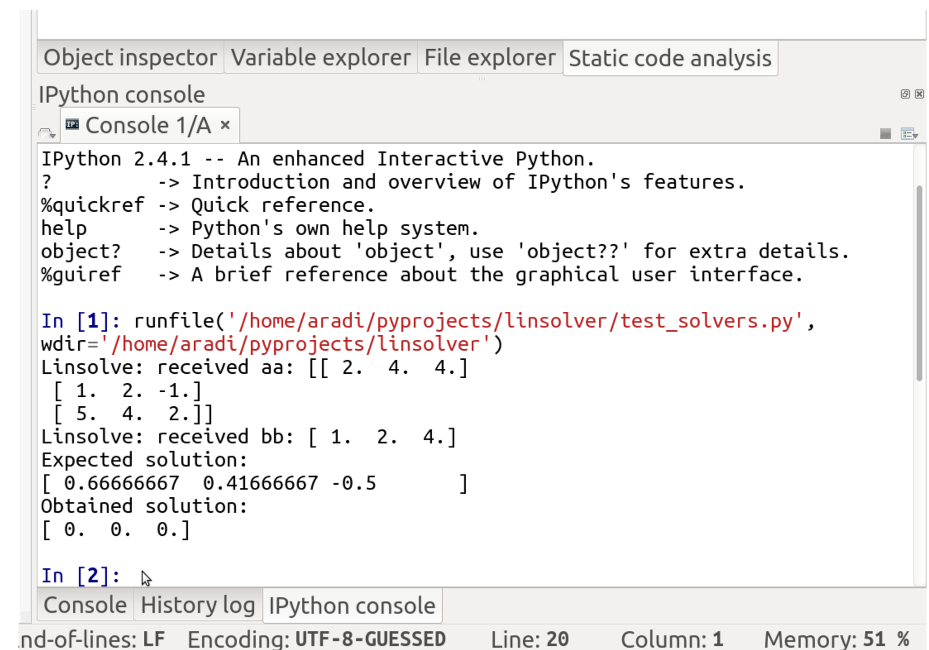
```
#!/usr/bin/env python3
:
```

```
chmod +x ./test_solvers.py
./test_solvers.py
```

Execute a Python script within Spyder3

- Spyder3 can execute any Python script of the project
- The script can be either run in a **separetate Python process** or in the **IPython console**
- Execute the **test_solvers.py** file in the IPython console (press **F5** when the **test_solvers.py** is the active file in the editor)





- **Congratulation! You successfully started your first Python project!**
- Before you start to develop it, it 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

Version control

Main tasks of a version control system

- 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 version control system (CVS, Subversion, …)

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

Distributed version control system (Git, Mercurial, Bazaar, …)

- Every developer has a local copy of the full development history
- Most version control 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"
```

Command Sub-command Option

- Specify standard tools to be used

```
git config --global core.editor featherpad
git config --global diff.tool kdiff3
git config --global merge.tool kdiff3
```

- Option **--global** stores options globally (for each git project), otherwise they are only valid for the current project
- Global options are stored in the **~/.gitconfig** file
- Current options (global and project specific) can be listed with **--list**

```
git config --list
```

Create a repository

- Initialize a repository in the project directory

```
cd ~/projects/linsolver
git init
Initialized empty Git repository in
/home/aradi/projects/linsolver/.git/
```

Creates an empty revision database in ~/projects/linsolver

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

Add files to the version control

```
git status
On branch master

Initial commit

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

        .spyproject
        __pycache__/
        solvers.py
        test_solvers.py

nothing added to commit but untracked files present (use "git add" to track)
```

Staged files

- When issuing **git add**, corresponding files (changes) are staged
- Staged files (changes) are written to the database at next commit

```
git add solvers.py test_solvers.py
git status
On branch master

Initial commit

Changes to be committed:

   new file:   solvers.py
   new file:   test_solvers.py
```

Ignoring non-version controlled files

- **Files not supposed to be version controlled** can be listed in the **.gitignore** file in the project directory

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

Initial commit

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

	new file:   .gitignore
	new file:   solvers.py
	new file:   test_solvers.py
```

```
.spyproject
__pycache__
```

Save (**Ctrl-S**) and exit featherpad (**Ctrl-Q**)

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

Commit project status

- When a **commit** is made, the staged changes are written into the database

```
git commit
[master (root-commit) 04d3866] Add first stub files
 3 files changed, 39 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 solvers.py
 create mode 100644 test_solvers.py

git status
On branch master
nothing to commit, working directory clean
```

Opens editor (featherpad)

Write log message (“Add first stub files”), save and exit featherpad

Checking project history

- Show project history:

```
git log
04d386638495386aa29ee99e4928aad2e7731f39
Author: Bálint Aradi <aradi@uni-bremen.de>
Date:   Fri May 18 21:17:48 2018 +0200

    Add first stub files
```

- 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 (and shows also shortened checksums)

```
git log --oneline
04d3866 Add first stub files
```

Checking project history

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

```
commit 2a3186299e14575a40b870cc3f8eb21c1e886809
Author: Bálint Aradi <aradi@uni-bremen.de>
Date:   Fri May 18 21:37:48 2018 +0200

    Add readme file

commit 04d386638495386aa29ee99e4928aad2e7731f39
Author: Bálint Aradi <aradi@uni-bremen.de>
Date:   Fri May 18 21:17:48 2018 +0200

    Add first stub files
```

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

Navigation:		
	[space]	Page down
	b	Page up
	q	Quit pager

Git-workflow

- Set up git global for your Unix account

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

- Set up the repository for your project

```
git init
```

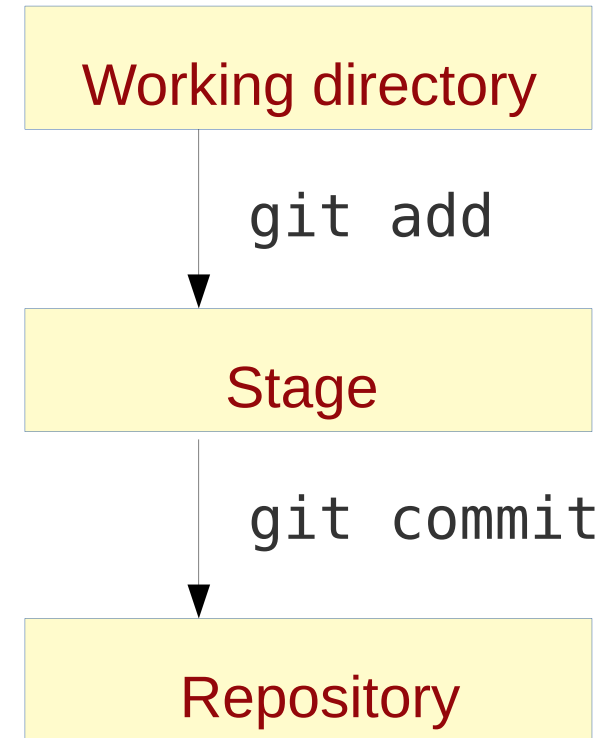
- Edit files in your project

- 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 notes

- Changes should be commited, 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"
```

Rename files

- **Rename a file** under version control:

```
git mv README README.txt
git status
# On branch master
# Changes to be committed:
#
(use "git reset HEAD <file>..." to unstage)
#
#
renamed:
README -> README.txt
git commit -m "Rename readme file"
```

- Corresponding file in working directory will be renamed immediately
- The name change must be committed like any other change

Delete files

- **Delete (remove)** a file under version control

```
git rm unnecessary_file
git status
# On branch master
# Changes to be committed:
#
(use "git reset HEAD <file>..." to unstage)
#
#
deleted:     unnecesary_file
git commit -m "Delete unnecessary file"
```

- Corresponding file in the working directory will be deleted immediately
- The removal must be committed like any other change
- The file will be not present in **future revisions**, but stays part of the previous commits.

Investigating changes

- **Changes between working copy and last checked in / staged version**

```
git diff README.txt
diff --git a/README.txt b/README.txt
index 8eab0a7..770eee5 100644
--- a/README.txt
+++ b/README.txt
@@ -1,5 +1,5 @@
-**********
-Linsolvers
-**********
+*********
+Linsolver
+*********
```

Lines removed

Lines added

- If no file name is specified, all changes in all files are shown

```
git diff
```

Investigating changes

- **Changes between two committed revisions** can be queried by specifying the revision hashes

Optional, if missing all changes shown

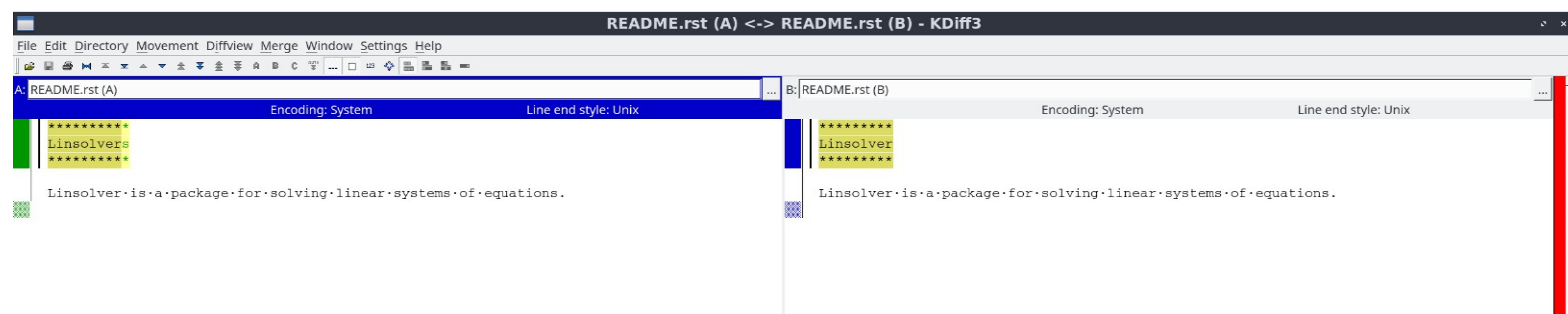
```
git diff 04d386 2a3186 -- README.txt
diff --git a/README.txt b/README.txt
new file mode 100644
index 0000000..8eab0a7
--- /dev/null
+++ b/README.rst
@@ -0,0 +1,5 @@
+**********
+Linsolvers
+**********
+
+Linsolver is a package for solving linear systems of equations.
```

Investigating changes via external tools

- The **difftool** sub-command calls the default diff-viewer to **visualize changes**

```
git difftool

Viewing (1/1): 'README.rst'
Launch 'kdiff3' [Y/n]?
```



Discard changes in working copy

- **Set working directory back** to last committed / staged version:

```
git status
On branch master
Changes not staged for commit:
  (use "git checkout -- <file>..." to discard
changes in working directory)

   modified:   README.txt

no changes added to commit (use "git add" and/or
"git commit -a")

git checkout -- README.txt

git status
On branch master
nothing to commit, working tree clean
```

Note: overwrites working copy immediately!

Unstage files

- Staged files can be **unstaged**, if they should not be part of the next commit
- Corresponding file in the work directory is not changed by the opertation

```
git status
On branch master
Changes to be committed:

   modified:   README.txt

git reset HEAD README.txt
Unstaged changes after reset:
M  README.txt

git status
On branch master
Changes not staged for commit:

   modified:   README.txt
```

Check out an earlier version

- **Previous commits** can be checked out by specifying their hash value

```
git checkout 2a31862
M  README.rst
Note: checking out '2a31862'.

You are in 'detached HEAD' state...

HEAD is now at 2a31862 Add readme file

git status
HEAD detached at 2a31862
```

- You have to change back to the current version (or to create a branch) to commit any changes

```
git checkout master
Switched to branch 'master'
```

Git aliases

- Aliases help to **abbreviate often used git commands and options**

```
git config --global alias.ci commit
git config --global alias.co checkout
git config --global alias.st status
git config --global alias.gdiff difftool
git config --global alias.slog "log
--pretty=format:\"%h | %ad | %s%d\"
--graph --date=short --all"
```

Please create these aliases for your account, since the following examples will make use of them!

- If an alias is used, the corresponding command / options will be substituted

```
git ci -m "Add quick changes"
git co 2a31862
git st
git gdiff README.rst
git slog
```

Tagging versions

- **Commits with special importance** (e.g. release) can be tagged
- Annotated tags are commited with a log-message
- By default the last checked in commit is tagged

```
git slog
* 2a31862 | 2018-05-18 | Add readme file (HEAD -> master)
* 04d3866 | 2018-05-18 | Add first stub files

git tag -a 0.1

git slog
* 2a31862 | 2018-05-18 | Add readme file (HEAD -> master,
tag: 0.1)
* 04d3866 | 2018-05-18 | Add first stub files
```

- Tag names can be used instead of revision hashes in git commands

```
git diff 04d3866 0.1
```

Git help

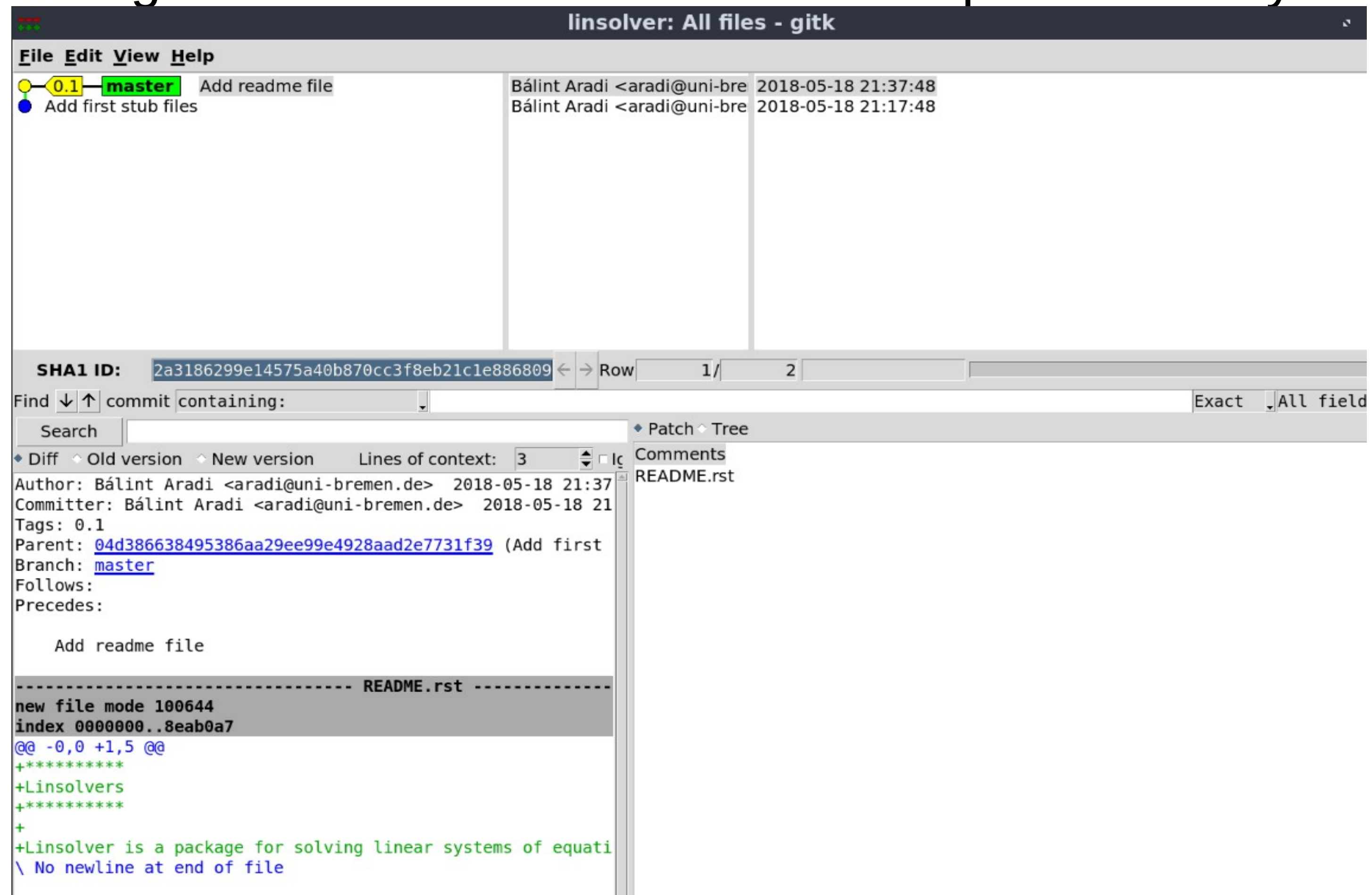
- Git can display help for every subcommand:

```
git help commit
```

Displays help for subcommand commit

- Several graphical git-clients exist to visualize development history:

```
gitk
```



Clone a repository

- Existing repositories (e.g. published on hosting sites) can be **cloned**
- Local clone will contain **full revision history and annotated tags**

```
git clone
http://www.bccms.uni-bremen.de/fileadmin/BCCMS/CMS/people/aradi/sciprog/python/linsolver.git
Cloning into 'linsolver'...
```

- You can work (change, commit to, etc.) with the cloned repository as with any other locally created one

Some further git-notes

- **Read the manual** for detailed git options
- You should **commit after each non-trivial change** of the project. **Rule of thumb**: It should be easy for other developers to follow and understand the changes of a commit.
- One commit should always contains **logically related changes**.
- Version history is stored in the .git sub-directory. If it is copied with the project, the version history is copied as well.
- Git commands must be executed in the project directory or in a subdirectory of it.
- If no files are specified, git commands have the entire project (the files which are already under version control) as target
- Revision hashes are global: They represent the status of all files in the project to a given time.