Writing Good Scientific Code – Part: Version Control using Git

The best way to learn Git is by using the commands yourself. Therefore, we will have many exercises today split up in three exercise blocks:

1. Git basics – from Git start to recording first file changes
2. Git as version control system – from commit history to undoing changes on different levels
3. Coexistence of multiple versions – Git Branching and Integrating

Team up in pairs (ideally using the same operating system as you normally use). For your team, use one of the breakout rooms to collaborate. At the beginning of each block, determine an expert for the current exercise block in your team.

It makes sense to go through the commands at your own pace, but it will also help your understanding to talk about it frequently. Therefore, either you can work as a team using the screen-share feature or, if you prefer to work on your own, you can try to do the exercises yourself and discuss your ideas and solutions with your team after every task.

At the end of each exercise block, notify one of the tutors. The expert should be able to explain the exercise solutions and general concepts of the block to the tutors. The other team members can help, but the expert should take over the main part of this discussion. This way, we want to be sure that you do not just copy the commands from the exercise sheet, but instead understand the commands and the underlying concepts. This sounds annoying, I know. Nevertheless, doing the tasks first and explaining them in your own words afterwards makes it easier to remember. Therefore, the blocks will not only have exercises but also explanations of basic terms and concepts.

After talking with one of the tutors, your team will get the exercises for the next block and the expert role will go over to the next member of your team.

Do not feel pressured in the explaining session with the tutor; it is not a test, just a small discussion to get a feeling if you have understood the concepts right and to correct you if it is not the case. Of course, you can also ask every time if you do not understand something or you just want more insight.

In the end, after all three blocks, we want to have cross-team meetings: for each block, we will assign expert groups for each block. Discuss shortly and visualize/write down as an expert team for this block the most important points (main concept) in a short presentation. You will then present these to all students.

If you use Git already, try to help your team members and if you still feel bored have a look at the advanced user tasks of the third session after you have completed all the tasks quickly.

We based the exercises on modifying a python script. If you want, you can also use jupyter notebook instead to edit and run the code. In this case, just copy-paste the contents of the provided .py files into the cell of a new notebook and save that as instructed.

Note: If you want to work with jupyter notebooks and git, we made a small adjustment to the git configs to better handle the notebook format. These are in the files ~/.gitconfig and ~/.gitattribute. Briefly, the .ipynb file contains not only the source code, but also some metadata like how often a file was run etc. When talking about version control, you usually don't want to consider your notebook to have a new version, just because this metadata has changed (i.e. you have just executed it again). The modifications in these files takes care of this. You do not need to understand them in all detail, but keep this small trick in mind if you want to use notebooks + git.

Exercise block 1: Git basics – Git basics – from Git start to recording first file changes

1. Getting Git:

1.1. Install Git on your personal computer.

linux: very simple, just use your packet manager, e.g.:

```
fedora: sudo yum install git-all
debian: sudo apt-get install git-all
```

mac os: In case you are using homebrew, simply run: `brew install git`
Otherwise go to https://git-scm.com/download/mac and download manually.

windows: Go to https://git-scm.com/download/win, the download should start automatically. Otherwise download manually and follow the setup instructions (use the default settings).

Note: Make sure to select that you want to add git to your PATH variable during installation

Git (on mac os and windows) comes with a built-in GUI, for now we want to **use the command line** (bash, windows: git-bash) to learn the Git commands without any bells and whistles. If you understand the concept of Git and know the commands, you can use a GUI later. The commands there are the same but partly restricted. So, prefer the terminal over the GUI for now.

2. Your first git repository:

You can think of a repository (short repo) as a directory containing all current files of your project, but also information about all previous versions of these files. There are two ways to get a repo, either you create a new one or you clone an existing one.

2.1. Choose a location on your hard drive and create a folder called *gitTutorial*
2.2. Within this folder create a new folder *myFirstGit*
2.3. Go into the new folder and open up a git shell by right clicking into the folder and selecting "`Git Bash Here`" (Windows). If you are using Linux/MacOS and have added Git to your PATH, simply open a terminal and cd into the *myFirstGit* directory.
2.4. Turn the *myFirstGit* folder into a git repository by running `git init`. It will give you `Initialized empty Git repository <path>`
once it has completed
2.5. Run `ls -la`. You will notice that a new folder *.git* has been created. If you take a look into it (`ls .git`) you will see that Git automatically created some sub folders and files for the organization of the repository.

3. Configure your git

3.1. Run `git config --list`
This command will give you a list of the current Git configuration.

If user.name and user.email is already set, change it to something new (figure out for yourself how). If these two doesn't exist in your configuration so far, figure out how to set them.

Get familiar with the configuration option. What is the meaning of the options `local` and `global` here?

3.2. Afterwards run `git config --global core.pager "less"`
This setting determines which pager is used when Git pages output such as `log` and `diff`, two commands which we will use later. It guarantees us today that the output of the two commandos looks the same for everyone today.

4. Start to record file changes

4.1. Download file *randomNumber.py* from the course page in *studip* and put it into the myFirstGit folder. Ignore the content of the python script for now. Run `git status`. What does git show? Try to understand the output

4.2. Run `git add randomNumber.py`. How does the status of the file change?

4.3. Commit the file with `git commit -m <commit message>`, choosing a meaningful message and check the status with `git status`. What happened so far?

4.4. Think about the 3-area concept of Git (Working directory, staging area, repository) and draw a small picture to visualize it. Which different states can a file have? In which area does a file have which states? How are the commands `add` and `commit` related to these three areas?

4.5. Open the file *randomNumber.py* and replace the test usernames with the names of all team members

4.6. Run `git status`

➔ Git will automatically tell you that your python file has changed.

➔ By running `git diff` you can even see the exact lines that have changed. This is because Git has saved a snapshot of our committed version against which we can compare. This is a quite useful tool to have. You can work on your project and use `git diff` to check what you have changed.

4.7. Add and commit your file changes.

5. Don't track everything

We want to keep the changes of our project over the time, so it's useful to track most of the files. But there are some exceptions, for example log files.

5.1. Now have a look in the python script *randomNumber.py*, and try to understand the content and run the python script using

```
python randomNumber.py
```

5.2. Run `git status` after that. The python script has generated a new file called *randomNumber.log*

5.3. The new file is untracked so far, but it's shown. We want Git to ignore changes on this file because it's just a log file and we don't want this file to be recorded. Create a new file called *.gitignore* containing a line *randomNumber*.log
5.4. Add and commit the *.gitignore* file
5.5. Run `git status` again. The file *randomNumber.log* should no longer appear in the output.
5.6. Change the name of the *outputfilename* variable within the python script and run the script again.
5.7. *randomNumber.py* create a new log file with the new name plus the extension .log. Run `git status`, Git will still show you the new log file in the status output, but we don't want to track any log file, whatever its name is. Figure out how to do this!

Besides log files, the .gitignore file is often used for tmp directories, tmp files or compiled source code (.o, .x, .pyc, …). In fact, you should try to ignore anything that is created during runtime.

Now, make sure that your working directory is clean at this point: every file should be either committed or ignored with the help of .gitignore

Running `git status` should give you:

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

6. Names are not so interesting

Let's *rename a file, does git understand this?* Try to understand the following process by using the `git status` command in every step.

6.1. Rename the file *randomNumber.py* (any name of your choice). What does `git status` say?
6.2. Add both files to the staging area by typing:
```
git add <oldfilename>
git add <newfilename>
```
6.3. Run `git status` again, what does it say? Commit your changes. Think about the whole renaming process: when does git understand that a file has been renamed? How can git understand that? To get an idea of the latter, also try out the following:

Change the lower border of the number generator, rename the file to coloursDice.py, and add the file with the new and old name by using

```
git add <oldfilename>
git add <newfilename>
```

Don't commit yet!
Use `git status` to see if Git still reports the files to be the same(only renamed and slightly modified) or as two different files. Do not commit this status yet, instead do the following changes:

Change the lower boundary to one and the upper boundary to six to simulate a dice throw. Write a new function called

```
get_colour_by_dice(spots)
```

which returns for each outcome of your dice throw a different color name. This function should be called in the main with roll (the return value of `get_random_number`) as input. Write this result to the log file.

If you have problems to write this function because you don't feel familiar with python, you can copy the code from *coloursDice.py* (available in *studip*) into your python script. The focus here is not python programming.

Add the changes. Does Git still understand that it is the same file after adding all this code? How many changes + rename does it take until Git does not recognize that it is the same file but views them as two separate ones? Do a google search if it is not yet clear to you.

6.4. Commit all your changes

6.5. So far, task 6 has shown you how Git sees files. However, renaming is even easier if you use a Git command directly. Find out which command you can use to rename a file without having to add the file with the new name to the staging area afterwards (Hint: try `git –help`). Use this command to rename the python script for the last time (back to the original file name *randomNumber*.py).

6.6. Commit all your changes

7. The fast way *(extra-task, if you get here in time)*

7.1. Add the name of your expert for the first exercise-block in a new comment line under the team member line in *randomNumber.py*:

```
# expert of exercise block 1: <name of your team expert>
```

7.2. Run the command

```
git commit –am <your commit message>
```

What does `git status` say? What is the difference from before?

7.3. Extend your 3-area concept picture from task 4.4 by adding the commands you have used in the whole exercise block