

## Exercise block 3: Coexistence of multiple versions – Git Branching and Integrating

After all this back and forth between versions, make sure your python script still has at least a function which gives you a random number (*get\_random\_number*) and a function which gives you a color depending on a number (*get\_color\_by\_dice\_roll*).


If you have lost some of these, there is (almost) always a way to get it back in Git, but to save time for now just download the *randomNumber\_backup\_to\_start\_task14.py* and copy the missing functions in your own *randomNumber.py* script in case you don't have these two functions anymore.

Make sure everything is committed and your working tree is clean, before you start this exercise session.

### 14. Branching – create and delete branches

#### 14.1. Run the `git status` command again.

Beside the information we have already considered from the output of this command, it also tells you on which branch you are currently working on.



```
user@PC ~/Documents/gitTutorial/myFirstGit (master)
$ git status
On branch master
nothing to commit, working tree clean
```

If you use the git shell (windows), it also shows you your current branch all the time.

The default branch name is *master*. The master branch is not a special branch but just the default name when you create a git repo with `git init` (done in task 2)

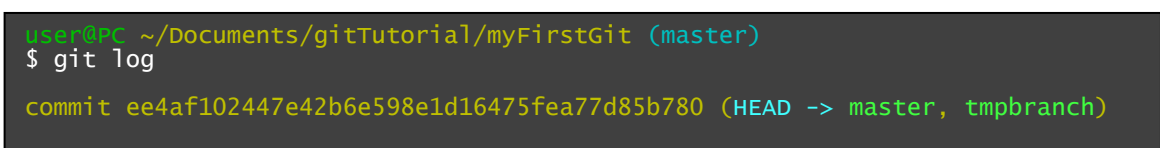
#### 14.2. Let's create a new branch named *tmpbranch* by using

```
git branch tmpbranch
```

#### 14.3. Check the output of `git log`

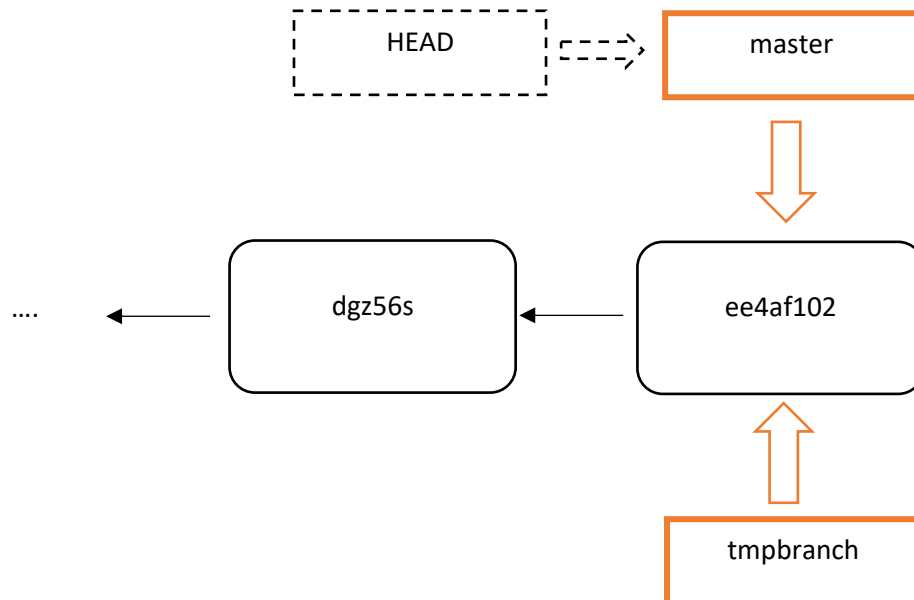
You should still see your old commits, but also the name *master* and the name *tmpbranch* behind the last commit. A branch is not a special location, it is just a lightweight movable pointer to a commit and currently both point to your last commit. However, how does Git know on which branch you are? For this, it keeps a special pointer named HEAD

You can see in the `git log` output also to which branch HEAD currently points.



```
user@PC ~/Documents/gitTutorial/myFirstGit (master)
$ git log
commit ee4af102447e42b6e598e1d16475fea77d85b780 (HEAD -> master, tmpbranch)
```

Have a look at the `git log` screenshot and the following schematic to understand this concept.



- 14.4. Now we want to switch from the master branch to the tmpbranch, so we want HEAD pointing to *tmpbranch* instead of pointing to the *master* branch. We can do this by using
 

```
git checkout tmpbranch
```
- 14.5. Check with `git status`, that you have switched to the *tmpbranch*
- 14.6. Check also with `git log` that we HEAD has moved
- 14.7. Another command which tells you on which branch you're currently on and additional which branches exists is: `git branch`  
Run this command. How does it tell you on which branch we are currently on?
- 14.8. Rename the function `get_random_number` in your python script, close the script, add and commit the change.
- 14.9. How does the picture of 14.3 change? Draw a picture of the current state.
- 14.10. Assume that we have created the *tmpbranch* just to try something out but do not want to continue with these changes. We want to delete this branch. Of course, we cannot delete a branch we are currently working on, because it would be like this:



<https://de.toluna.com/polls/6542669/Redewendungen-und-Redensarten-Welche-sind-euch-bekannt>

So, first switch back to your *master* branch by using

```
git checkout master
```

and delete the *tmpbranch* afterwards with

```
git branch -D tmpbranch
```

Note: Deleting a branch is in general not necessary; you could have just gone back to the master (in this case) and continued from there by treating the *tmpbranch* like a dead branch. It is just a pointer so we do not care about it in general. Moreover, maybe you need things later again that you would not have expected to need again.

## 15. Branching – fast forward merge

However, branches are often created to try something out or to continue the work with keeping a stable version on another branch (usually master). This is especially helpful in collaboration projects. Everyone can work on their own branch and commit their daily work even if it is not compiling or has other open to-do's. If a part is finished and working properly, it can be merged into the master branch. Another use case could be a program, which other users are actively using. Then you do not want to work on the same version, but instead use a branch for your daily work and take it over in the master branch only if you are sure that everything will work again.

Of course, it is not necessary to use branches if you work alone, but it makes life easier. For example, you could have a branch which always has a stable version of your project and branches which you use to develop, to test, to hotfix, or to implement an idea which came to your mind and you want to try out.

For all these, we have to know how to take over changes from one branch into another.

15.1. Create a new branch called *dev* and switch directly to the new one in one command by using `git checkout -b dev`

It is the combination of the two commands `git branch <branchname>` and `git checkout <branchname>` we used in task 14.

15.2. Add the following code in the main function of your python script (adjust it to your function names if necessary):

```
rolls=[]
for i in range(6):
    roll = get_random_number(1, 6)
    rolls.append(roll)
print(rolls)
sys.stdout.flush()
```

This call creates an array of size 6 with random generated values between 1 and 6. Make sure that your script still works.

15.3. Close your script, add and commit this change to your *dev* branch.

15.4. Switch back to your master branch using `git checkout master`

15.5. Have a look at your python script, the array code should not exist here

15.6. Take the changes you have done on the dev branch into the master branch by using

```
git merge dev (be sure you are currently on the master branch)
```

With this command, you merge the change into your master branch commits.

Git can use a fast-forward-merge strategy in this case, because the dev and the master branch have the same commit history except the last commit of the dev branch. So, the pointer of the master branch is just fast-forwarded to this commit.

You can imagine that often merging is not so easy, because both branches might have changed in different ways since their common ancestor commit. Let us have a look at this.

## 16. Branching – recursive merge

16.1. Switch back to your dev branch and add the following import command right below the existing import commands in the script

```
import matplotlib.pyplot as plt
```

16.2. Add and commit your change and close your script.

16.3. Switch back to your master branch, it does not have the import command. We want to introduce here another change instead:

Extend the code from 15.2 by adding the following two lines in the end:

```
plt.bar(range(6), rolls)
plt.show()
```

16.4. Add and commit your changes to the master branch.

16.5. Your two branches master and dev are diverged now, because they have the same commit history until the second to latest commit, but then both of them have an

additional commit, which the other one does not have. You can also see this easily with the `git log` command if you use

```
git log --oneline --decorate --graph --all
```

Try to understand the output of this command.

- 16.6. We want to keep both changes by taking over the commit from the dev branch into the master branch, because we need both of them to plot our `rolls` array successfully.

Be sure you are currently on the master branch and then type `git merge dev`

With this command, you merge the change into your master branch commits. This time, Git performs a recursive merging strategy because it has to consider both changes. Thereby, it will create a “merge commit”. This is why a window with a commit message opens automatically when running this command. Keep the message as it is and just close it by typing `:q`

- 16.7. Have a look at the script to see if git has taken over both changes. Afterwards, run the script to plot your array.

**Note:** If the following error occurs: `ModuleNotFoundError: No module named 'matplotlib'`, you have to install the `matplotlib` package to be able to run the script.

```
python -m pip install matplotlib
```

- 16.8. Draw a new scheme like the one in 14.3 to visualize the two merging processes  
16.9. Delete the dev branch or consider it as a dead branch in the following.

## 17. Branching – advanced recursive merge

- 17.1. Create a new branch named *plotting-playground*, you should already know the commands for this. Then switch to this new branch (or do both in one step)
- 17.2. Change the command of the bar plot to a horizontal bar plot.
- ```
plt.barh(range(6), rolls)
```
- 17.3. Run the script again with this new plot command to be sure no error occurs. Close your script, add and commit your changes.
- 17.4. Switch back to the master branch, it still has the `.bar` instead of `.hbar` command. Again, we want to do another change here: rename the array (change it from `rolls` to another name) and adjust all lines in the script which use this variable
- 17.5. Close your script file, add, and commit your changes.
- 17.6. The two branches `master` and *plotting-playground* are again diverged. Both of them have a commit, which the other one does not have, but both of them changed the same line. You can visualize this by running

```
git log --oneline --decorate --graph --all
```

again.

So, what will Git do if we try to take over the changes from the *plotting-playground* into the master? Try it out.

- 17.7. Open the `randomNumber.py` script again to see how Git handled this situation.
- 17.8. Make sure that in the end your script has both changes the array name you chose in 17.4 and the `hbar` plot instead of `bar` plot from the change in 17.2
- 17.9. Make sure that in the end, your script is still working and your working directory is clean.
- 17.10. Visualize what has happened by running again

```
git log --oneline --decorate --graph --all
```

and try to understand it

## 18. Another way to integrate

So far, we have learned how to integrate changes with `merge`, but Git as a second way to integrate changes from one branch into another: `rebase`

- 18.1. Create a new situation where you have two branches (*master* and *experiment*) which both have at least one commit, which the other branch does not have, i.e, viewed from a common ancestor commit, each of them has at least one additional commit, similar situation like in task 16.5. For now, try to do these commits in a way, that they could be merged automatically (e.g. change a function name on one of the branches and add a comment line with the today's date on the other)
- 18.2. Now be sure that you are currently on the *master* branch and run the following command  

```
git rebase experiment
```

What happens?
- 18.3. Draw again a scheme like the one you have for the `merge` strategy to visualize how the `rebase` strategy works
- 18.4. What are the differences? Can you find any advantages or disadvantages for some cases? When should you prefer one strategy for the other?

## 19. Further commands (extra-task, advanced users)

If you use Git already or understand the basics quite fast, there could be some time left to look at the following commands, which are not so frequently used but are actually helpful in some situations.

### 19.1. `git blame`

With this command, you can see who changed what in a file. Besides its negative sound, it can be helpful to identify changes in your file. Imagine you are working on a larger project with others and the *master* branch has changed. You integrate the changes from the

master into your current branch, everything runs smoothly and is merged automatically. It can be helpful to run `git blame` on the file you are currently working on to identify changes and who to turn to if you have questions about them.

To simulate such a situation, change the `user.name` in your git configuration, change something in the python script, add and commit your changes. Then run

```
git blame <filename>
```

It should give you the information who has changed what in the file and when.

### 19.2. `git cherry-pick`

At this time, you should also know two concepts of getting commits from one branch: rebasing and merging. Until now, we have always taken over the complete branch history since it branched off from the *master*. Sometimes, it can be useful to take over only some certain commits instead of all. Let us try to get single commits from another branch.

- a. Because you have reached the advanced user level now, generate the following initial situation yourself: new branch with three or more additional commits in comparison to the *master* branch. Check with `git log`
- b. Switch back to your *master* branch. Our goal is now to get only the first commit you have done in task a) into your *master* branch. In fact, this is possible with the `git cherry-pick <SHA1>` command. Try to run it to get the commit from your branch to your *master*. Do you understand how this works? Have a look at your commit history.

### 19.3. `git bisect`

Knowing when a bug was introduced in a project can be very useful to find the cause of the error. This command uses a binary search algorithm to find the commit that introduced a bug. Therefore, you have to find a “good” commit without the bug and a “bad” commit containing the bug. Afterwards `git bisect` will pick a commit between those two endpoints. Then you have to decide whether the commit is “good” or “bad”. The binary search then continues until it finds the exact commit that introduced the bug. Let us try.

- a. For this task, we want to use an existing repository. Go back to the parent folder *gitTutorial* and use

```
git clone https://github.com/simondolle/git-bisect-sample.git
```

to clone the repository for sample code to demo the `git bisect` command made by Simon Dollé.

- b. Follow the *How to run the demo* instructions of the readme.