Prof. Dr. A-S. Smith
am Lehrstuhl für Theoretische Physik I
Department für Physik
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Advanced Python for Research Projects

Exercise sheet 5: Git version control

Problem 5.1 *Some git nomenclature*

To be able to better understand git, its usual errors and use cases, it is important to know a few terms of the nomenclature surrounding git. Familiarize yourself with the following terms and potential differences:

a) Plumbing vs. porcellain commands: What is the difference and which of the command we have talked about in the lecture are part of which group?

b) Blob vs. Tree vs. Commit: What do these terms refer to, how do they correspond to common terms in computer use and how does git use these in its usual version control workflow?

c) Working copy/directory vs. Index/Staging area vs. Repository vs. Remote: How do these different terms relate to each other? In which order do changes to files flow through these stages of storage/version control?

d) Head vs. Branch vs. Tag vs. Release: What is the difference between these terms (if there is one) and how do they relate to each other?

e) What is the difference in effect between a `git pull` and a `git fetch` call?

Problem 5.2 *Setting up a git repository the simple way*

In the lecture, we have illustrated the details of creating a git repository manually. In practice, you will rarely have to manually set up object storage, tree writes or a manual commit yourself. Here, we instead want to set up a repository in a way that you may go about it in your scientific work through porcellain commands.

a) Go to the folder of the previous exercise, where you set up the package structure for your pypi package project to be installed to a local user. We now want to put that folder under version control and create a git project in an existing folder structure. You can do this, by calling `git init <directory>`, where the `<directory>` option refers to the folder where you want to create the git repository. If the folder does not exist, it will be created first. As we want to create the project in an existing folder, open your terminal in the root of the existing package project and initialize the git repository for the local directory denoted by the relative specifier `.` to indicate the git repository should be created here. Git will then automatically create the `.git` folder structure and a main or master branch (depending on your version of git).

b) We soon want to add the different files from the previous exercise to this repository, but first, we want to initialize the appropriate branches for git workflows. Let's create a `develop` branch using the `git branch develop` command. Then confirm whether you are still on the main or on the develop branch using the `git branch` command. If you are not on the develop branch, use `git checkout <branch>` to switch to the correct branch. If we opt for a very fine-granular branch setup, we may now create a new branch for the initialization of the project, but in most cases, the initialization can just be done on the develop branch.

c) On the develop branch, we first want to avoid adding temporary files generated by python or other tools. Let us create a `.gitignore` file in the root of the project and add the appropriate contents for a python project. You can generate a `.gitignore` file for example with a generator like `https://www.toptal.com/developers/gitignore` and specify the Python language. The resulting contents can then be copied into your local `.gitignore` file. Add the `.gitignore` file to the git index with a `git add .gitignore` call.

d) You should now check, which files git registers as new. User `git status` to get a list of new files and directories git has recognized. You should see the `.py` files in the `fauap_advanced_functions` folder, the `.py` files in the `tests` directory and the documentation files in the `docs` directory. Additionally, meta files in the root directory for the project (like `Readme` and `pyproject.toml`) and the `setup.py` file. Make sure you see no folders anmed like `__pycache__` or files witht a `.pyc` suffix. Those are temporary files generated by python and should not be added to a git repository. Other programming languages have different suffixes for temporary files that need to be listed in `.gitignore` so make sure to generate an appropriate `.gitignore` file for your respective project. If you still see those temporary files in `git status`, check the spelling of your `.gitignore` file, look if there are rules in there to exempt those temporary files and, if they are missing, add such rules. Then, once you only see appropriate files in `git status`, add the files of your project. We recommend at least going folder-by-folder as `git add -all` often adds unwanted files. After adding all files, check the status again to see, wich files are now part of the index. If there are temporary files that have been accidentally added, remove them via the `git reset` or the `git restore` commands. The `git status` overview will give you the appropriate instructions for how to remove them from the index/staging area.

e) We now want to create our initial commit of the project to git, so check that all relevant files of the project are part of your staging area in `git status`. If there are some secrets like passwords written somewhere in your project files, you should make sure, that none of those parts of the code are part of your index before committing. Otherwise, it is usually hard to get rid of those entries again. Once you are sure, that you only have the relevant files, create an initial commit on the `develop` branch with an appropriate commit message. Commit messages are usually written in present tense, like "Add functionality to write files" or "Fix bug preventing file output on a Tuesday". Some projects opt for a style more along the lines of "<File or topic>: <message>" to make it easier to see, in which aspect a commit alters a project from the topic or filename before the colon. For an initial commit, it may suffice to write "Initialize project with existing package setup from previous exercise". (It is also common to not add fullstops at the ends of commit messages, but your project is yours to govern. Just make it consistent and write down such rules, e.g. in a `Readme` or in a `CONTRIBUTIONS` file.)

f) look at your newly created commit either via `git log` or via `tig` and confirm all files have been added. You can also confirm this with `git status` if the files in your project are no longer listed as new/added or modified.

Problem 5.3 *Dealing with remotes*

We now want to incorporate remotes (or at least "a" remote) into our project as a remote backup and to allow for collaboration between different developers and computers. You have multiple options to go about your choice of a git provider. Generally, you can go with `github` or `gitlab` as a recommendation. Both offer free hosting of projects and you can just create a free account on their cloud services. `gitlab` can also be downloaded and installed on your own computer or server if you want.

a) The university hosts a gitlab server for computational sciences that we can use for our project test purposes. You can find the service at `https://gitlab.cs.fau.de`. Use the option to log in with your university account (SSO/Single Sign On) and your account should be automatically created. This instance of gitlab is limited in how much storage you may use with your project. Don't upload unnecessary data or your projects may be deleted automatically after a warning email. For github or the cloud gitlab, the account creation may vary and other restrictions may apply for free accounts.

b) Create a project on the gitlab. As a name for this exercise, `fauap_advanced_functions` would be appropriate. The projects are located in namespaces (or groups), meaning that the same project name may exist for different users and groups. You should add a short description of the project on the creation page and for now set the project to private. Then create the project.

c) Now that you see the project page, gitlab will provide you with some examples of how to clone the empty project or how to add the empty project as a remote to an existing local repository. We want to add a remote called `origin` that points towards this gitlab project. For this, we need the project url. Click on the blue "Code" button and you should see an https and an ssh link for referring to the remote. With the https link you will have to specify your password every time you try to interact with the remote. We therefore recommend using the ssh link (Looks something like: `git@gitlab.cs.fau.de:<user>/<project>.git`. Don't worry, the user here is always git, no need to modify it). Copy the ssh link. In your local project, run `git remote add origin <url>`, where you put the ssh link in place of the url placeholder. if you then run `git remote` you should see the `origin` remote listed and if you run `git remote get-url origin`, it should specify the ssh link you have copied previously.

d) Now, to interact with the remote, we need to set up the ssh connection. For this, we need to put our public ssh key from the local machine into gitlab. The local key is available in `~/.ssh/id_<method>.pub` in your local home directory. The method refers to the encryption algorithm used and may vary. Copy the contents of this file. It should be a text file with contents of the form `ssh-<method> <encoded key> <key name/comment>`. Now go click on your user avatar on gitlab, go to ">Preferences > SSH-Keys". If there are already keys registered with your account, you can manage them here. Click "Add new key" and copy your public key into the big text field. You can also specify and expiry date for the key and assign a name. The name should tell you, which machine the key is on in case you ever have to figure out, which of your keys has been compromised. Confirm the addition of the key. Now you should be able to interact with your remote `origin` via the usual commands.

e) We want to publish our branches to the remote server. We first need to set up git to automatically use the remote `origin` for push and pull operations. For this, use `git branch -u origin develop` to set the upstream (i.e. the default remote) for branch `develop` to `origin`. Depending on what your main/master branch is called, to the same for that, i.e.: `git branch -u origin master` or `git branch -u origin main`. Now, to publish your develop branch, run `git push`. Git may tell you to set up your author name and email address using `git config -global user.email "<your email address>"` and `git config -global user.name "<your name>"`. Put your name (can be an alias) and your email address (you can use your fau address here, normally you would use either a public email or an email related to the entity that owns the project or employs you) into these commands. Then try and repeat the command. Follow git instructions if further issues arise. When the commands finishes, you should have successfully published the branch "develop" to the remote. You now also need to publish the main or master branch. For this you can either switch to the main branch using `git checkout main` and then push again or you can run `git push -all origin` to publish all branches to the remote `origin`.

f) Go to the project page on gitlab and confirm that now there are two branches main and develop listed.

Problem 5.4 *Cloning the project again from the remote and setting up a release*

Let us now pretend that we do not have the project locally yet. Go to a different folder in your home, create a folder `fauap_advanced_functions_dev` and clone the project from gitlab into it.

a) To clone the project, go into the created folder and run `git clone <ssh_url_of_origin> .`. The full stop `.` specifies the current directory as the place to put the project. Per default, your project may be set up to have `develop` as your default branch in gitlab. If that is the case and you are on `develop` after cloning, you can change this behavior in your Project under "> Settings > Repository > Branch defaults". Just select the main branch and save.

b) We now want to execute a common workflow and merge the develop branch into main to create a release. Due to us cloning the project, in this repository, the remotes should automatically have been set up. The remote `origin` should already be available and we should be able to switch to a branch `develop` and a branch `main` via `git checkout <branchname>`. Make sure that you are on the main branch, switch to it if necessary. You can confirm your current branch with `git status` or `git branch`. Now that we are on the main branch, we want to merge the `develop` branch into the `main` branch. The command to do this is `git merge develop` if you already have the branch develop checked out locally or `git merge origin/develop` if the file does not exist locally yet. As mentioned in the lecture, when merging a feature branch, it may be useful to compress all individual commits on the feature branch into one commit to keep the history clean, e.g. from typo correction commits, etc. In that case, you may create a `squash` merge commit by running `git merge -squash <branchname>`. There are arguments pro and contra squash commits. In some cases, you may want to maintain the context provided by the individual commits, in other cases you may want to keep the history clean and use `-squash` instead.

c) you now should have a new commit on the master branch, that needs to be published to get the remote up to date. Push the current state of the master branch. Confirm that the main/master branch on the gitlab now contains all of the files previously commited to `develop`.

d) Now, we want to add a tag to comemorate this initial commit as version `v0.0.1`. Use `git tab <tag name> -m <tag message>` to create the tag on the current main branch. Use our version string `v0.0.1` as the tag name and add a tag message specifying that this is the initial project setup after exercise 5. Make sure that the tag has been create with `git tag -list`. Now we want to publish the tag using `git push origin tag <tag name>` to publish this tag to the remote. The tag name is again the version string. Once you push the tag, it should be visible on the gitlab in the "Tag" section on the right hand side under "project information". Confirm that the tag is visible and has the desired message associated with it.

e) Finally, we want to turn our project tag into a full release of the project state. Click on the "Create release" button in the tag overview on the tag you want to create the release from. Now add a name vor the release (E.g. "Version 0.0.1 release"), set the release date if desired, set a release message, for which you can include the message of your tag with the checkbox. The release message usually includes a list of all new features, all fixed issues, all removed features and potentially some warnings related to compatibility issues. In some sense, it has the same information as module documentation in python. The more useful your release name and message, the easier it will be for others to find the relevant version they need. Then click "Create release" again and you should now see the new release listed under "Releases" in "project inforamtion". The release should contain a zip file of the full project state relevant for publications but it can also be extended to contain compiled versions of your project if it is a project in other programming languages.