git

Yvan Lengwiler
Faculty of Business and Economics
University of Basel

yvan.lengwiler@unibas.ch

File: git.tex

Contents

1 Version control and sharing

Git is a version control system. It allows you to work on a project, save milestones (or even small steps), and go back to previous milestones if you have made a serious mistake along the road that would be difficult or tedious to correct. Projects are held in so-called *repositories*, which is really a folder with attached history and all the files needed for version control.

The online versions of git — these are websites where repositories can be stored publicly — are extremely important for collaborative work. If multiple people can access and contribute to the same repository, a project can more easily be developed by a team, and the changes made by each team member are retraceable. The most important public git server today is `https://github.com`, is owned by Microsoft, and is free to use.[1] Other offers are `https://gitlab.com` and `https://bitbucket.org`.

`git` and `Github` are not the same thing. You do not need `Github` to use `git`. However, you do need `git` to use `Github`.

By the way: I learned the basics of git and Github from Nick White's video at `https://youtu.be/mJ-qvsxPHpY?si=zcRIjqRiU_PJ6uXE`. I highly recommend to watch this 20 minutes video.

In the following, sections that are more advanced have a star (*) in the title. Please skip these if you are just beginning with git.

2 Get git

2.1 *Installation*

We start by installing the git software on our local machine. Browse to `https://git-scm.com/downloads` and follow the instructions. Git is contained in many Linux distributions[2] and is easily available for Mac if you have homebrew installed. In other cases, and for Windows, you have to download a binary file and install that.

If you have to install a binary, I suggest you test first whether it is free of malware. You can do that with your local antivirus software. Alternatively, you can upload the downloaded file to `https://www.virustotal.com/` and let this website check your file using multiple virus detection packages.[3]

Before we can use git, we have to configure it.

[1] You should be aware that all code you make public, especially in a prominent place like Github, is used to train large language models.

[2] Check with `'which git'`. If it is not already installed, you can most likely install it with your distribution's package manager.

[3] The Linux and Mac versions simply provide access to the `git` program from the command line. The Windows version does that, too, i.e. you can use the `git` from the command terminal. But in addition, the Windows installation also provides a new type of shell, called `bash`. This is accessible when you right-click in some folder, or from the Windows start button. If you use `git` from the `bash` shell in Windows, you can use the Linux syntax. This means that you can also use the Linux commands described below. (The Windows commands reported below only work in the normal Windows command shell.) There are also different GUIs available for `git`, which are not described in this handout.

Global config Linux, MacIntosh, Windows

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
git config --global init.defaultBranch main
```

The first two lines simply identify the user. This is important for git because this software is meant for collaboration, and it is important to keep track who does what. The third line looks a bit mysterious, and it is not necessary, but has become a convention. Git uses 'branches' (we will see later what that means), and these branches have names. Before, the standard name of the default branch was `'master'`, but in recent years it has become custom to call the default branch `'main'`. The third line makes us adhere to this de-facto standard. But you do not have to. You could call your default branch `'home'` or `'startrek'` or whatever you like, or you could not do the third line and then the default branch is called `'master'`.

2.2 *Trying it out: First commit*

Make a new folder. Then, declare this folder as a git repository.

Initialize repository Linux, MacIntosh, Windows

```
mkdir test
cd test
git init
```

This folder is now initialized as a git repository. Next, make some random files,

Make some files

Linux, MacIntosh

```
$ touch app.py solution.R
$ ls
```

Windows

```
> echo "Python program" > app.py
> echo "R program" > solution.R
> dir
```

Now save these files in your repository,

Save in repository Linux, MacIntosh, Windows

```
git add .
git commit -m 'first version, just two files'
```

The `'add'` command stages the files (makes them ready for putting into the repository). The example here stages the whole directory (note the `'.'` after the `add`), but you could also stage a few particular files instead. The `'commit'` command actually puts the staged material into the repository. The `-m` adds a comment (a free string) that identifies the commit. This is helpful so that we later know where we are returning to.

2.3 *Commit some more changes . . .*

Suppose we decide to solve the problem in Python and not R. So we remove the R files. We also add a file containing some slides to show in the next Zoom session. We then commit these changes to the repository.

More changes

Linux, MacIntosh

```
$ rm solution.R
$ echo "adding informative presentation" > slides.pdf
```

Windows

```
> del solution.R
> echo "adding informative presentation" > slides.pdf
```

We can see the current status of our repository (i.e. if something needs to be committed or not) with `'git status'`, and we can take a look at the history of our repository with `'git log'`,

Current status and log of changes Linux, MacIntosh, Windows

```
git status
git log
git log --oneline
```

`'git status'` should give us an output like this,

```
------------------------------------------------------------------------
On branch main
Changes not staged for commit:
```

```
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
 deleted:    solution.R

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

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

After staging the changes with `'git add .'`, the `'git status'` is

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
 modified:   slides.pdf
 deleted:    solution.R
```

When we then commit the changes with

```
git commit -m "removing R attempt, adding some slides"
```

then `'git status'` returns

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

`'git log'` should give us the history of out past commits,

```
commit 9a2684dfe66534d052ff03c0e27ed0d3f7d9765e (HEAD -> main)
Author: Yvan Lengwiler <yvan.lengwiler@unibas.ch>
Date:   Thu Mar 7 21:55:05 2024 +0100

    removing R attempt, adding informative presentation

commit c83e67a10b1469a1b0a03acde2b70e6d05c36c58
Author: Yvan Lengwiler <yvan.lengwiler@unibas.ch>
Date:   Thu Mar 7 21:54:24 2024 +0100

    first version, just two files
```

Each commit has a hashtag (the non-sensical looking hexadecimal code on the lines that start with `commit`. These hashtags are important.

As you work on your project, the commits will accumulate. The output of the log function will become too verbose. You can generate a much more compact output with `'git log -oneline'`. This uses only one line for each commit containing only the most relevant information.

2.4 `'git diff'` *and* `'git restore'`

If you have changed a file since the last commit, you can see the differences between the current and the last committed file with `'git diff FILE'`. This can be very instructive.

If you have changed, added, or deleted files and not staged them yet with `'git add'`, you can undo the changes with `'git restore FILENAME'`. This is a convenient way to going back to the last commit for a specific file. You can also restore a file that you have deleted and that was still present in the last commit. This amounts to undeleting the file.

You can also revert just parts of the changed file. You do this with `'git restore -p FILE'` (`-p` is short for `--patch`). This opens the file in your terminal and goes through the changed sections, asking you if you want to revert or not.

If you want to revert a file that has already been staged with `'git add'`, simply add the option `-staged`, such as `'git restore --staged FILE'`, for instance.

2.5 *Subfolders and* `.gitignore`

The root of your repository (i.e. the folder we have now always worked in) can have subfolders. These subfolders are managed just like files and are part of the version control. In other words, you can add, delete, change files in subfolders, and also add and remove subfolders themselves, and these changes will be reflected in the version control mechanics.

You can also exclude some files or subfolders from version control. Make a file called `.gitignore` and simply list all file names that gut should ignore, one file per line. You can also use globals (e.g., `*.temp`) to exclude a whole group of files.

Depending on your programming language, there may be files you typically want to ignore. `https://www.toptal.com/developers/gitignore` is a website that produces sensible `.gitignore` templates for your chosen programming language. But you are free to create or edit this file according to your needs.

You can also ignore file-types globally for all our repos by creating a corresponding file at `/.config/git/ignore`.

2.6 ** Going back in time*

You can go back to a previous state of the project. Copy the hashtag of the situation you want to go back to and type this command,

Going back in time Linux, MacIntosh, Windows

```
git checkout c83e67a10b1469a1b0a03acde2b70e6d05c36c58
```

If you check the content of your directory now, you will see that the R file has reappeared and the slides have disappeared.

You also get an explanation of what goes on here:

```
---------------------------------------------------------------------------
Note: switching to 'c83e67a10b1469a1b0a03acde2b70e6d05c36c58'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at c83e67a first version, just two files
---------------------------------------------------------------------------
```

In other words: You have gone back in time, so to speak, and you can develop the project further but differently than you did before. You can make this new branch the main one with `git switch -c "new-branch-name"`. You can also go back to the original main branch with `git switch -`. Let us do that now

```
git switch -
```

Make sure that the R file has disappeared again.

2.7 ** Branching and merging*

Maybe a cleaner way to do such a branch is by explicitly create a new branch with a new name.

Making a new branch Linux, MacIntosh, Windows

```
git checkout -b newbranch
touch data.csv
git add .
git commit -m "adding some great data"
git log
```

There is a little bit to unpack here. The `git checkout -b newbranch` makes a new branch and gives it a name. This is actually a short form for a sequence of two commands,

```
git branch newbranch
git checkout newbranch
```

With `'git branch'`, you get a list of all the branches that exist in your repo.

We then add a file,[4] stage it (`add`) and finally commit it to the repository. The final log should give us something like this,

```
---------------------------------------------------------------------------
commit 75f95c400688a6da039a99115dfc45b00ea70069 (HEAD -> newbranch)
Author: Yvan Lengwiler <yvan.lengwiler@unibas.ch>
Date:   Thu Mar 7 22:14:03 2024 +0100

    adding some great data

commit 16759f528385aa34efbd9402fe6a231cb1cfad77 (main)
Author: Yvan Lengwiler <yvan.lengwiler@unibas.ch>
Date:   Thu Mar 7 21:55:05 2024 +0100

    removing R project, adding informative slides

commit 5a10542af999da7c9dc13c375cb5a7e51ff539c2
Author: Yvan Lengwiler <yvan.lengwiler@unibas.ch>
Date:   Thu Mar 7 21:54:24 2024 +0100

    first commit, just two files
---------------------------------------------------------------------------
```

Note that the `"HEAD"` (i.e. the branch we are currently working on) is called `"newbranch"`, but the former main branch called `"main"` still exists. With

[4] Note: under Windows, there is no `touch` command, so make the file in some other way, e.g. as before with `echo` and piping the output into a new file.

List of branches Linux, MacIntosh, Windows

```
git branch
```

you get a list of branches of the current project. The one with the star in front is currently HEAD (i.e. active). You can change the active branch like this:

Switch active branch Linux, MacIntosh, Windows

```
git checkout NAME-OF-NEW-ACTIVE-BRANCH
```

We are currently on the `"newbranch"`. We can switch back to `"main"` with `'git checkout main'`. Verify that you are indeed in `"main"` now with `'git branch'`. Also see the list of files in your directory: `"data.csv"` should have diappeared (because that was only in the `"newbranch"`).

If you have developed your project within a branch and are happy with that, you might want to merge it with the original main branch. You can do this with the following:

Merging Linux, MacIntosh, Windows

```
git checkout main
git merge NAME-OF-BRANCH-TO-MERGE
```

So again, in our case, we can merge the `"newbranch"` with `'git merge newbranch'`. The `"data.csv"` file is now part of the `"main"` branch.

2.8 * *Changes within a file and merging conflicts*

What happens if you not only add or delete files, but change the content of some file in a branch and then want to merge that back to main?

This is the real beauty of git: That works in exactly the same way. The merge will change the content of the file, not just its existence.

Note, however, that this can lead to conflicts. We will now force such a conflict:

1. We add `file.txt` and write the line `"This is the first line, created in the MASTER branch."`
2. Make a new branch: `git checkout -b sideline`
3. We add a second line to `file.txt`: `"This line was created in the branch SIDELINE."`
4. Add and commit: `git add .` and then `git commit -m "2nd line to 'file.txt' in SIDELINE branch"`

5. We switch back to main: `git checkout main`
6. Add a second line to `file.txt` in the MASTER version: `This is the second line created in the MASTER branch.`
7. Add and commit: `git add .` and then `git commit -m "2nd line to 'file.txt' in MASTER branch"`

We have now two versions of `file.txt` in the two branches that are incompatible with each other, because their second lines are different. In the MASTER branch, the file looks like this:

```
-------------------------------------------------------------------------
This is the first line. It was created in the MASTER branch.
This is the second line created in the MASTER branch.
-------------------------------------------------------------------------
```

In the SIDELINE branch, the file contains this:

```
-------------------------------------------------------------------------
This is the first line. It was created in the MASTER branch.
This line was created in the branch SIDELINE.
-------------------------------------------------------------------------
```

Merging will expose this conflict: `'git merge sideline'` produces an error message:

```
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

`file.txt` now contains this:

```
-------------------------------------------------------------------------
This is the first line. It was created in the MASTER branch.
<<<<<<< HEAD
This is the second line created in the MASTER branch.
=======
This line was created in the branch SIDELINE.
>>>>>>> sideline
-------------------------------------------------------------------------
```

You have to manually sort this out. To do that, you have to edit the conflicted file and remove the lines that indicate the conflict (<<<<<<<, =======, and >>>>>>>). You can edit the conflicted file with any editor. You can also use `'git mergetool'` if you are proficient with the VIM editor. After resolving the conflict, add and commit the file that has the conflict to the repo again. By the way, branches can be deleted like this:

Deleting a branch Linux, MacIntosh, Windows

```
git branch -d NAME-OF-BRANCH-TO-DELETE
```

2.9 *Where is all this stored?*

If the system can resuscitate deleted files, they must be stored somewhere. Also, the log of all events must also be stored somewhere. Where is that?

All of this data is stored in a hidden subfolder called `'.git'`. You can browse to this folder and study what is inside. For instance, have a look at the file `'config'` inside this directory. Moreover, if you completely delete this subfolder, all git-activity is removed and your folder is no longer a git repository.

3 Going online

A git server is simply a place where git repositories can be stored. The largest one is probably `https://github.com`, but there are other ones as well, such as `https://gitlab.com`, `https://codeberg.org`, `https://bitbucket.org`, or `https://sourceforge.net`. A public git server has the advantage, that you can share your files with everyone. Likewise, you can download files from other people who have shared their files easily over the internet using such a public git server. It is also possible to have a private repository on a public git server, and to share it only with specific people. Modern software development relies heavily of git and on public git servers today.[5]

3.1 *Cloning*

You can retrieve projects from Github even if you do not have an account there. In fact, this is the most common use of git and Github: most users simply download material from Github but do not upload anything ever.

The simplest way to download a repo from Githib is by 'cloning' it. On each Github project, there is a green or blue `Code` button that opens a dropdown menu. From there, you can choose the links to use with HTTPS or SSH. At this stage — since we have not yet made our own account and have no SSH keys,[6] use the HTTPS link. Copy that link and say

[5] You can host your own git server, see here, `https://git-scm.com/book/en/v2/Git-on-the-Server-Setting-Up-the-Server`. If you want to see an expert do that, look here, `https://www.youtube.com/watch?v=ju9loeXNVW0`. You could use such a server in your home LAN to ease the transfer of files between different machines that you use. You could, of course, also make this server face the open internet (but be careful about hackers if you do that...). We will not go there in this course and only use git locally, as well as on a well-established public git server.

[6] What's that, you ask? You will see in a minute.

`'git clone PASTE-IN-THE-LINK'`. The files and all the history of the repo is placed into a subfolder of your current directory that has the same name as the repo you are cloning.[7]

If you are interested only in the active branch and no history, you can add the option `'--depth 1'` to the command like this: `'git clone --depth 1 PASTE-IN-THE-LINK'`. The `'--depth 1'` option instructs git to only clone the latest version of the repo. All the history is not cloned. This is often enough if you just want to use the content of the project you are cloning. If you want to develop it further, you might want to download the complete history instead.

3.2 *Getting an account*

We will use the market leader `https://github.com`. Browse there and sign up for an account. If you already have one, you can use that, or you can make a new one just for university work — it's up to you.

Note that Github requires two-factor authentication (2FA). You can use an authentication app (such as Authy or the like), a physical authentication device (like Ubikey), or SMS. SMS is considered the least secure because nothing is encrypted in SMS and SIM cards can be spoofed. In any case, it is advisable to install two types of 2FA because that way you can still access your account even if one method does not work (for instance, because your smartphone died). Github will also show you a list of numbers, the `recovery codes`. Store this in a safe place as this will allow you to regain access to your account if all else fails.

3.3 *Authentication part 1: making an SSH key pair*

Github will not allow you to manage your repo just with a password. A more secure authentication method is requires. One possibility is to access Github through the "secure shell protocol" SSH. For that, we need to create SSH key pairs. A key pair consists of a private key (akin to an actual key) and a public key (akin to a lock). You can share the public key file; you should never share the private key file.

You can create an SSH key pair like this,

SSH key pair Linux, MacIntosh, Windows

```
ssh-keygen -t ed25519 -C "Github:unibas-account"
```

The `'-t ed25519'` determines the type of cryptography that is used. The default is `rsa`, which produces longer keys that are, however, less secure than ed25519 (or so I am told). The string that follows the `'-C'` switch is a comment. You are completely free what you want to write here. It has no effect on the functionality of the key. It is generally a good idea to succinctly describe who this key belongs to or what it is used for.

[7] If you have the `gh` utility installed, you can also use the corresponding `gh` command provided in the green pulldown menu.

This command will ask for a name of the files. The default is `"id-ed25519"`. You might already have files with that name that you probably want to keep, so you should choose another name. Since we create these explicitly for Github, we could use the name `"github-ed25519"`, for instance.

The dialog will also give you the opportunity to use a passphrase for the key. Choosing a passphrase means that the key can only be used by a person who knows the passphrase. If someone steals your private key but does not know the passphrase, the key is useless.

The command produces two files, `github-ed25519` and `github-ed25519.pub`, or whatever the name of the file you chose. The `.pub` file is the public key, the other one is the private key. Make shure to put there files in the `$HOME/.ssh` (Linux, Mac) or `"C:\Users\YOUR-USERNAME\.ssh"` (Windows) directory. It will not work otherwise. If this directory does not exist, create it.

3.4 *Authentication part 2: storing your public SSH key in your Github account*

Now, view the content of the **public key** on the screen, with `'cat github-ed25519.pub'` (Linux, Mac) or `'type github-ed25519.pub'` (Windows). Mark it with your mouse and copy the content (this should be just one line) to the clipboard.

Now, go back to the Github website. In the main menu, go to `"settings"`, then `"SSH and GPG keys"`, then click on the `"New SSH Key"`. Give the key a descriptive name that allows you to later understand which key this belongs to, choose `"Authentication key"`, and paste the content of your new **public key** into the form. *Never ever paste a private key in there!*

3.5 *Authentication part 3: associating your private key with your Github account*

There are (at least) two ways to acheve this. a simple way is to issue the following command,

Setting the key to use Linux, MacIntosh, Windows

```
git config --global core.sshCommand "ssh -i $HOME/.ssh/YOUR-PRIVATE-KEY-FILE"
```

This adds a line to the `.gitconfig` file.

The alternative is to edit the configuration of `ssh` itself. Back in the terminal, go to the `"$HOME/.ssh"` directory (in Linux or Mac) or `"C:\Users\YOUR-USERNAME\.ssh"` (in Windows), respectively. The new public and private key files should be there. It is useful to also create an additional file called `"config"` in this location. If you already have such a file, just add the content below to it.

```
Host github.com
    Hostname github.com
    User git
    IdentityFile ~/.ssh/github-ed25519
```

```
AddKeysToAgent yes
```

The user and host name in the internet is specified in the 'Hostname github.com' and 'User git' lines and this must be entered exactly as shown here. 'github-ed25519' (the name of the 'IdentityFile') should be substituted for the name of your key. This config file will make sure that whenever we try to reach git@github.com through SSH, your new SSH key will be used to authenticate.

You can check if this works with the command 'ssh git@github.com'. This should produce an output similar to this,

```
---------------------------------------------------------------------------
'PTY allocation request failed on channel 0
Hi yvan-lengwiler! You've successfully authenticated, but GitHub does not
provide shell access.
Connection to github.com closed.
---------------------------------------------------------------------------
```

This is, in fact, an error message, but it is the message you want to get. The connection was established but then was closed because Github does not allow access through a shell. If the output mentions "connection refused", then it did not work.

3.6 *Authentication part 4: SSH agent*

Now, if you have defined a passphrase for your ssh key (which you probably should do), you will have to enter this passphrase every time you use the key. This is tedious. One can get around this with an ssh agent. This is a piece of software that stores the passphrases you enter until you log out of the system. That way you have to enter the passphrase only once per session.

In Linux and MacIntosh, you start the ssh agent with 'ssh-agent'. To start this automatically, you could put the line

```
eval "$(ssh-agent)" > /dev/null
```

into the .bashrc (Linux) or .bash_profile (MacIntosh) startup script. Use the equivalent if you use a different shell. For Windows, in services, look for the "ssh-agent" and start it.

None of this is necessary. Everything works fine without the SSH agent. It is only for convenience.

3.7 *Making an online repository on Github*

Browse to the Github website and log into your account. Under "Repositories", click the "NEW" button, give the repository the name "test", fill out the rest of the form, click "OK".[8]

[8] Github has released a set of utilities that allow you to manage your Github account from the command line. You can create and manage online repositories, place and remove SSH keys etc. The tool is called 'gh'. You can install and use it if you wish. This handout does not cover it, though.

3.8 *Pushing*

After creating the repo on the Github website, the website gave us instructions on how to connect the local repo with the online one.[9] In the terminal, go to the folder where you have created your local repository and do as shown in the next box.

Pushing local repository, first time Linux, MacIntosh, Windows

```
git remote add origin git@github.com:YOUR-GITHUB-ACCT-NAME/test.git
git push -u origin main
```

Let us go through this. `'git remote add origin ...'` links your local repository to the online repository. Here, `yvan-lengwiler` is my Github username and `test.git` is the name of my Github repository. Adapt according to your situation.

`'git push -u origin main'` then pushes your local repository to the Github website. The `'-u origin main'` is only necessary for the first push. Subsequent pushes can be done just with `'git push'`. Please check on the Github website if your files were pushed to your online repository.

3.9 *Pulling*

Now, if your repository is a collaborative effort with multiple people contributing, it can happen that the online repository on Github gets out of sync with you local git repository. This is what pulling is here to address. With

```
git pull
```

you download all the changes in the online repository that are not reflected in you local repository. After that, your internet and local repositories will be in sync again. (Note that this might remove files from your local repository if these files have been removed from the Github repository in the meantime.)

In fact, if you work collectively on a project and use Github for this, make it a habit at the end of a working session to push your work. At the beginning of a working session, the first thing you should do is pulling from the online repository, so that you work on the latest version. If two people work simultaneously on the project, things can still get messed up pretty easily and a `'git pull'` will create a mere conflict.

You can test this before pulling by issuing the `'git fetch'` command. This downloads the changes from the online repository but does not integrate them into your HEAD branch. With `'git status'` you can see the changes.

In large projects, typically one one person or a small core group of people are allowed to push to the remote repository. All other contributors need approval to merge their changes. They can do that with a pull request (we will briefly discuss this later). For the context of this class, this is not relevant. We do not develop software jointly, so everyone has their own

[9] These commands also work if we created the online-repo with the `'gh'` utility from the command line.

repo and nothing should get mixed up.

Note that `'git pull'` is similar to `'git clone'` in the sense that it downloads the files to your local machine, but its purpose is different. `'pull'` updates an existing local repo that is already connected to the online repo to reflect all changes of the online version in the local repo. `'clone'` copies the remote repo and creates a new local repo on your machine from this that is connected to the online repo. After a `pull` and after a `clone`, the local and the online repos are in sync.

3.10 *Licensing*

This topic is not relevant for this course as long as you keep your repo private, but it will be very relevant if you plan, first, to share your work online, or second, if you use software that is made available online by someone else.

When using other people's software, check the license. The license determines how you can use the software and what you can do with the source code.

When you create software and make it available publicly, you should think about the license that fits your needs and select the one you like. `https://choosealicense.com/` and `https://github.com/readme/guides/open-source-licensing` can help you with that choice. It is not a good idea not to have any license, because you could be subject to liability for damages that result from someone using your software. The MIT license is the most permissive license. Users can do whatever they like with your software and even turn it into proprietary, closed source software. But this license still protects you from damage claims, so it is better than having no license at all.

Github allows you to choose a standard license easily from a drop-down menu. The license file will just be added to your repository. (If you are a lawyer, you can write your own — good luck.)

3.11 ** Pull requests*

Each public git repository has an owner. If someone pulled your project, that person might have created a new branch and developed the project further, without interfering with your work. This person can then send a pull request to your Github repository. This will inform you that someone has worked on your project and contributed to it, and proposes to merge his work with the main tree. You can then inspect the changes and accept or reject each change.

3.12 ** Forks*

A fork is similar to a pull and then creation of a new branch, but the owner of the new branch changes. There is no intention of merging with the main tree anymore and the project takes on a new life of its own under new ownership.

4 A final word

There is much more to git. Very large projects are today managed with git and with git servers. It is easy to see why: substantial software projects are often collaborative, and version control is extremely important, because sometimes a but is discovered (much) later and then it is important to trace back when it got introduced, so as to properly remove it.

The official documentation is available at `https://git-scm.com/doc`. There is a shorter documentation at `https://docs.github.com/en/get-started/getting-started-with-git` that focuses more on Github and the Github utility `gh`. Moreover, Github has also released a `'Github Desktop'`, which is a mouse-guided interface for interacting with git and Github. This is not covered by this handout.

Even if you will not be a code developer, people working professionally in finance today are expected to code at least to some extent, and that also means sharing code and contributing to the codebase of the company. In most modern finance houses, this means R, Python, C/C++, and, yes, git as well. (In other companies, it just means Excel :-)

By the way, Github repositories are not confined to software projects. You can push any kinds of files. I have seen some people storing their CVs there for submission to employers. If you want to convey a nerdy aura, that could be an interesting choice...

APPENDIX

A Multiple Github accounts

If you are just starting out, you should ignore this section. I makes managing Github more complicated than it needs to be. This section explains how to manage multiple SSH keys that are necessary if you have more than one Github account. This is not relevant for most people. If this is relevat for you, then read on.

There are two approaches to address this. The first uses the configuration of your git repository. The second approach uses the ssh configuration.

A.1 *Using `.git/config`*

Git allows you to specify the identity file (i.e. the private key) for each repository separately. The command to do that is

```
git config core.sshCommand "ssh -i $HOME/.ssh/YOUR-PRIVATE-KEY-FILE"
```

This adds the line

```
sshCommand = ssh -i $HOME/.ssh/YOUR-PRIVATE-KEY-FILE
```

in the [core] section of the configuration of the repo. This specifies the key file that is used for authentication via ssh with the Github website.

A.2 *Using `.ssh/config`*

I have two Github accounts. One is for teaching and one for research collaborations. Github does not allow me to use the same SSH key for these accounts. Thus, I create one keypair for the teaching account (`github-teach-ed25519`) and one for the research account (`github-research-ed25519`). I upload the public keys to the respective Github accounts.

The trick comes now: In `/.ssh/config`, I define two entries,

```
Host ghub-teach
    Hostname github.com
    User git
    IdentityFile ~/.ssh/github-teach-ed25519
    AddKeysToAgent no
Host ghub-research
    Hostname github.com
    User git
    IdentityFile ~/.ssh/github-research-ed25519
    AddKeysToAgent no
```

Furthermore, after cloning an online repo locally I have to adjust the online origin link. So instead of saying

```
git remote set-url origin git@github.com:GIT-USERNAME/REPOSITORY-NAME.git
```

I select which account to use in the `.git/config` file,

```
git remote set-url origin ghub-teach:GIT-USERNAME-TEACH-ACCOUNT/
    REPOSITORY-NAME.git
```

or

```
git remote set-url origin ghub-research:GIT-USERNAME-RESEARCH-
    ACCOUNT/REPOSITORY-NAME.git
```

This approach seems more general at first sight because we can set the identity files centrally. However, the first approach is really simpler. The second approach's advantage is possibly not really an advantage because, just as in the first approach, the `.git/config` file has to be manipulated all the same. The one advantage of the second approach is that you can change the names of your keys that are associated with a particular Github account in one place globally (in `.ssh/config`), and you do not need to repeat this change for every repo. But changing keys is maybe not something you do very often.

If you access your Github repos from different computers, and these computers use different keys, then the second approach can be valuable. The `.git/config` files are interchangeable between these different computers because they do not contain a hard-coded name of a key.

A.3 *Trouble with the SSH agent*

In both approaches, the SSH agent does not work right. Remember, the SSH agent allows you to enter the passphrase for your SSH key only once per session. However, since we have two different keys that are used with the same host (`github.com`), the SSH agent is confused, and sometimes the agent sends the wrong credentials, so that Github kicks me out. So, for the moment, I cannot use the ssh-agent anymore. This is of no concern if you have only one Github account, of course (or if you have chosen not to encrypt your keys with a passphrase).

B Removing old commits

Your git repository grows as you work on it. After all, old files are kept in the `.git` directory. It is possible to squash different commits together, remove certain commits, reorder them, amend the comment messages etc. with `'git rebase -i'`. The possibilities go beyond an introduction to git and will not be discussed here.

However, there is a not very subtle way to have a clean slate and remove all the history so far. You will start fresh by keeping just the current version and deleting everything else. By pushing this cleaned local repository to Github, you can also remove all old commits in the online version as well.

A somewhat inelegant — one might be tempted to call it brutal — way to achieve that is to simply delete the repository files, keep just what you need, and start afresh. The next box explains how to do this. However, before doing this, I suggest you copy the whole repository with the old history first to a backup folder, so that you could come back to it if the procedure should not yield the desired result.

Note that this procedure temporarily stores the `'config'` file of the repo in the root of the repo (normally, this file is in the `'.git'` subdirectory, but we delete this subdirectory). If your project contains a file `'config'` in root, then you should give it temporarily another name, to avoid overwriting and then losing a piece of your project.

Clean slate

Linux, **MacIntosh**

```
You start this procedure from the root of your repository:
cp .git/config .
rm -rf .git
git init -b main
mv config .git/.
git add .
git commit -m "Initial commit"
git push --force -u origin main
```

Windows

```
You start this procedure from the root of your repository:
copy .git\config .
rmdir /S /Q .git
git init -b main
move config .git\.
git add .
git commit -m "Initial commit"
git push --force -u origin main
```