Computational Methods in Economics (winter term 2019/20)

Tutorial 1 - Introduction to Version Control and Git

In [1]:
```
# Author: Alex Schmitt (schmitt@ifo.de)

import datetime
print('Last update: ' + str(datetime.datetime.today()))
```

```
Last update: 2019-10-18 11:36:53.072414
```

In [2]:
```
import IPython.display as display
```

This Lecture

Introduction: Version Control

"Brute Force" Version Control

In [3]:
```
display.Image('PhDcomics.gif', width = 500, height = 800)
```

Out[3]:
```
<IPython.core.display.Image object>
```

"Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later." (ProGit https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control)

In other words, version control systems allow you to access previous versions of files *without having saved them as individual files*. This means you can load an old version of a file, but also compare your current version with previous versions, and track the changes you have made.

Hence, you can modify your code, e.g. by experimenting with new features, while knowing that you can always go back to a working version.

These systems are widely used in software development (e.g. when experimenting with new features to a program), but are also extremely useful in scientific work (e.g. when writing a paper).

In particular, version control allows you to always be able to replicate old results (e.g. from a previous version of your model) or go back to previous versions of a paper.

It should be noted that there are different version control systems out there. For example, the popular filehosting service *Dropbox* features automatic version control (which you may not be aware of).

Here, we're going to use the by far most popular version control system, *Git*, which is free and open source.

Note that Git, in particular in connection with a website like *Github*, is also very useful in the context of *sharing files and collaborating with other people*.

There are a lot of resources on version control and Git online.

For a more in-depth treatment, compare the free book on the Git website: https://git-scm.com/book/en/v2

I can also highly recommend a (free) online course on Udacity: https://classroom.udacity.com/courses/ud775

Installing Git

There are different ways of working with Git. Some people like to use a "GUI" (graphical user interface) version, while others (myself included) prefer the "bash" version.

Here, I will start with a fundamental primer on Git using the bash version, and then briefly talk about a simple GUI that comes with Git on Windows. There are several GUIs out there (some for free, others not), so feel free to explore more sophisticated GUI version if you prefer. It doesn't matter how you use Git, as long as you use it :)

You can download Git here: https://git-scm.com/download/

Note that if you already have Git installed, check its version using the command **git --version** in a terminal or command line (see below).

An overview of different GUI versions can be found here: https://git-scm.com/downloads/guis

Installation notes for Windows: when setting up Git,

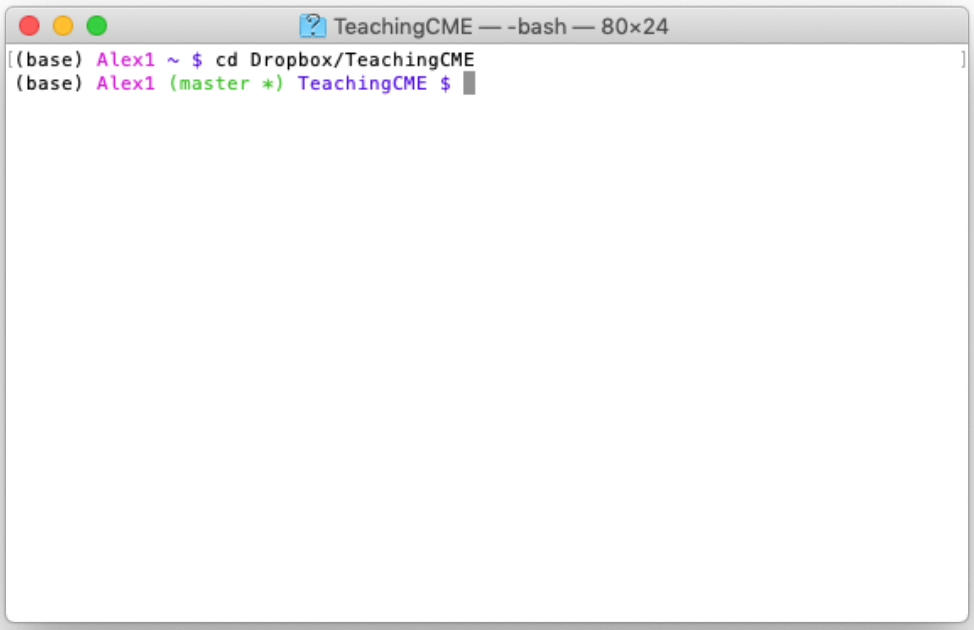
- choose the recommended or default option whereever applicable
- for the command line environment, choose "**mintty**" if you want to have the same environment as in the screenshots below
- depending on your Windows version, you may be asked which (external) text editor to use during the installation process; if you don't have one, you should get one, not only for Git, but also for writing scripts later on
    - There are several options that are compatible with Git and great for writing scripts. My preferred one is Sublime Text 2 or 3: https://www.sublimetext.com/3
    - Make sure to install your editor of choice before installing Git, so you can specify which text editor you want to use during the installation process. Otherwise, you can also set it up manually (see information at the end of the notebook).

Installation notes for Mac:

- Text editor: to use an (external) text editor with Git, first finish the installation process for Git. If you don't have a good editor, you should get one, not only for Git, but also for writing scripts later on
  - There are several options that are compatible with Git and great for writing scripts. My preferred one is Sublime Text 2 or 3: https://www.sublimetext.com/3
  - After installing the text editor of your choice, follow the instructions at the end of this notebook to link it to Git.
- When trying to open the Git installer, you may get a warning that the app cannot be opened because it is from an unknown developer. Go to **System Preferences** and then **Security and Privacy** to open it anyway (see also the following tutorial: https://codeburst.io/installing-git-for-the-first-time-on-mac-osx-bf9c513af2b8)
- **Git Bash** in Windows (or rather the **mintty** shell) comes with prompts in colors and more importantly, display the name of the branch you are on and whether there are uncommitted changes. The standard setup of Mac's **Terminal** does not have that. I provide instructions on how to change this setup at the end of this notebook, so that it looks the following.

In [4]:
```
display.Image('terminal_git.png', width = 800)
```

Out[4]:

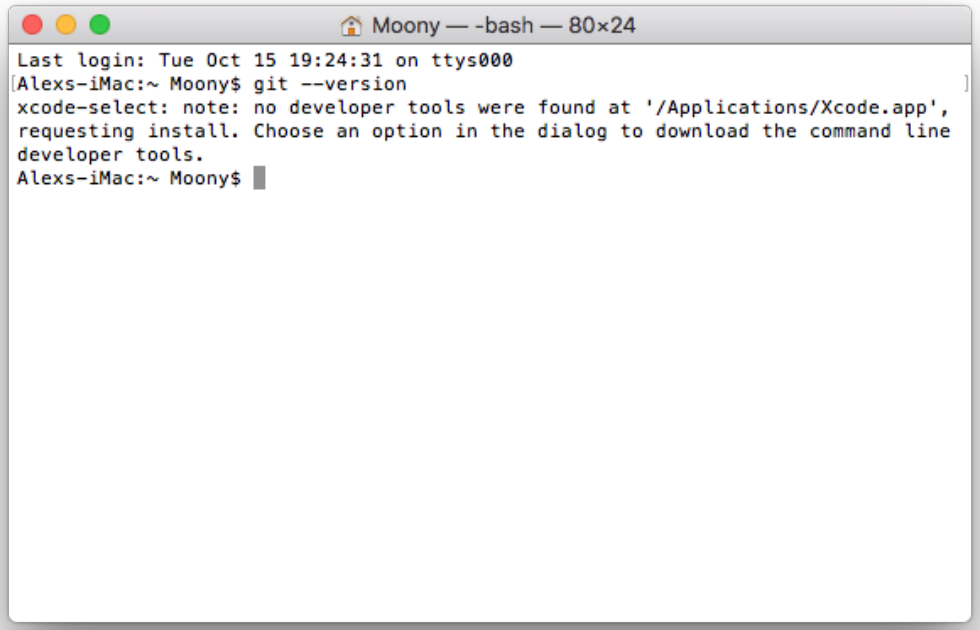


Installation notes for Mac (cont.):

- After successfully installing Git, when trying to open Git from the **Terminal**, you may encounter an error message asking you to download Apple's Xcode developer tools (which also include Git). Note that this happens on some, but not all versions of macOS.

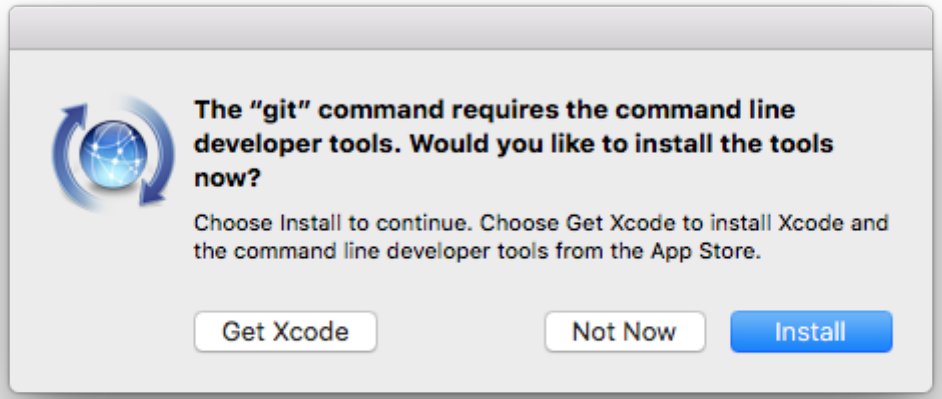
In [5]:
```
display.Image('git_terminal2.png')
```

Out[5]:



In [6]:
```
display.Image('git_terminal3.png')
```

Out[6]:



Essentially, your OS wants you to run Git out of the Xcode package. You CAN do this, but note that Xcode requires a lot of disk space (3-4 GB), and most of it is not necessary.

If you want to use Git without Xcode, run the following steps after successfully installing Git:

1. Start a terminal and type **cd ~** to get your home directory.
2. Here, look for the name of your Bash profile. Type **ls -a** to list all (also hidden files). Look for either **.profile** or **.bash-profile** (you should have at least one of them).
3. In your terminal, run the following command, depending on the name of your Bash profile: **echo "PATH=/usr/local/git/bin:$PATH" >> ~/.bash_profile (~/.profile)**
4. Open a new terminal window, or run **source ~/.bash_profile (~/.profile)** to update the change.
5. Run **git --version**. It should not give an error and instead show the your Git version, e.g. 2.23.0.

With these steps, you essemtially make your OS disregard the Xcode stub, and instead look in a different folder for your stand-alone Git installation. More information can be found here: https://blog.bobbyallen.me/2014/03/07/how-to-install-git-without-having-to-install-xcode-on-macosx/

If you run into problems, let me know!

Using Git For Version Control

Using Git Bash

On Windows, after installing Git, you can start both **Git Bash** and **Git GUI** from your Start menu.

On Mac, run the bash version of Git using the **Terminal**.

git init

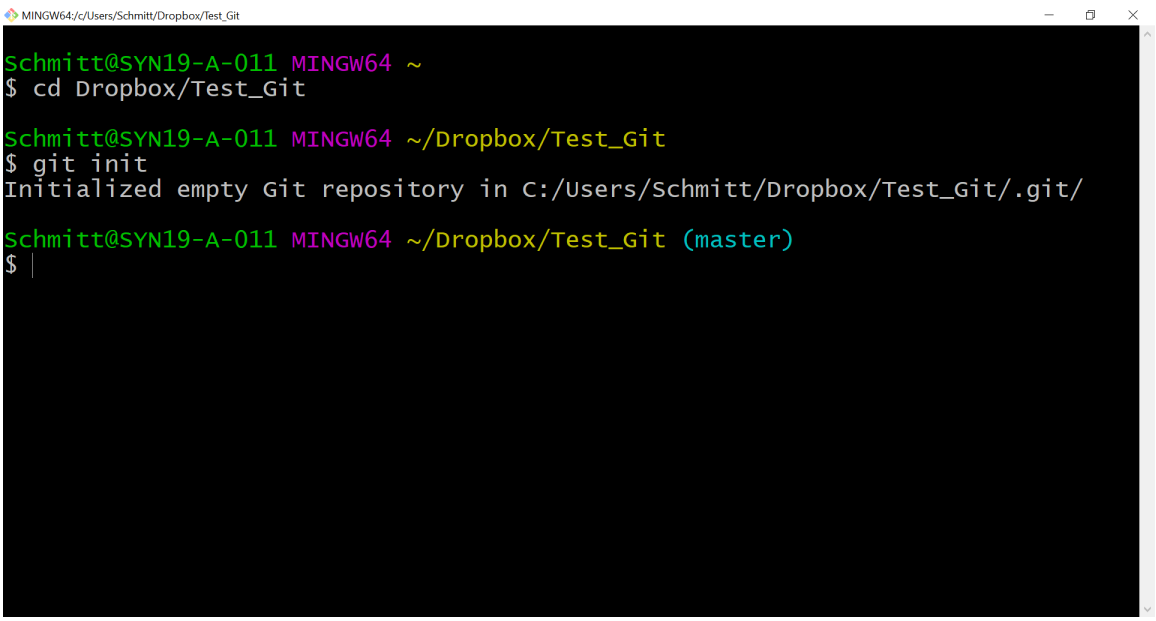
The first step of using Git is to initialize a *repository*.

After starting the bash version of Git, navigate to the directory/folder where your project files are located (or where you want to start a new project) and type **git init**.

As a side note, both **Git Bash** (Windows) and the **Terminal** (Mac) are examples of *command line interfaces* or *command line (bash) shells*. You can use **cd *directory_name*** to navigate down the tree and **cd ..** to navigate back. Use **dir** (in Windows) or **list** (on Mac) to see which files are contained in the current directory.

In [4]:
```
display.Image('git1.png')
```

Out[4]:



As a side note, **Git Bash** (or rather the**mintty** shell) comes with prompts in colors, displaying the name of the user (here in green) and the current directory (in yellow). The meaning of

(master) will become clear later.

A repository is a collection of files that have been *"committed"* (more on this below). This means, loosely speaking, that it is the set of files in a directory that Git keeps track of or "knows about".

A (local) repository can contain all the files in a folder, but can also contain only a subset of them.

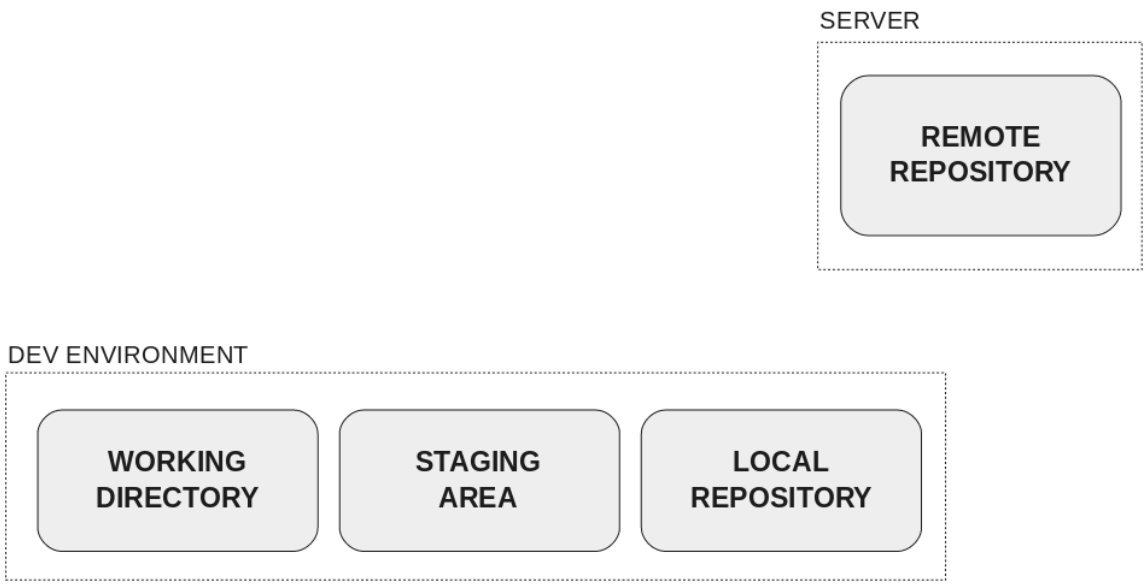
When we run **git init**, we create a new (and hence empty) repository for the current folder.

For better understanding the concepts to follow, it is useful to think of a repository as a directory that is linked to your current working directory (in the example above **~/Dropbox/Test_Git**) and that contains versions of some or all its files.

Compare the following illustration (ignore the other parts for now):

In [5]:
```
display.Image('git_schema1.png', width = 600)
```

Out[5]:



This picture and the ones used below are from https://rachelcarmena.github.io/2018/12/12/how-to-teach-git.html.

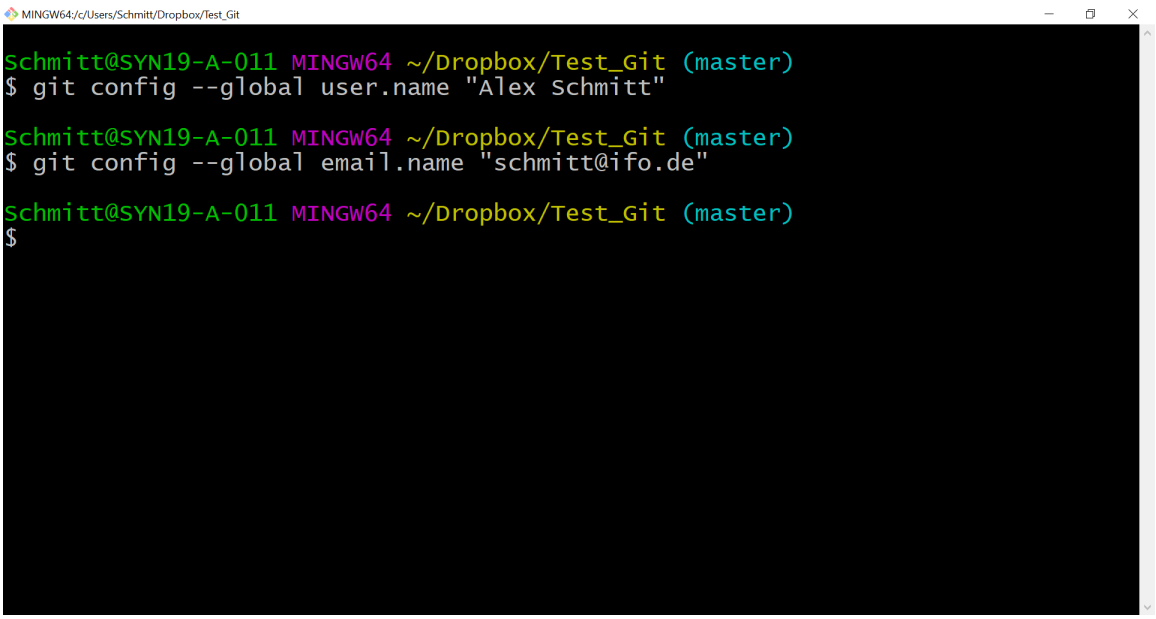
Before moving on, it is useful to set up your Git environment by setting your name and email address, so that Git keeps track of who makes changes to a repository. Type the following in Git Bash:

git config --global user.name "*your_name*"

git config --global user.email "*your_email*"

In [6]:
```
display.Image('git1b.png')
```

Out[6]:



Commits

A key concept in Git is the *commit*. Think of a commit like a snapshot of your files at one point in time.

This implies that you can always go back to previous commits, i.e. restore files to previous versions.

In contrast to for example Dropbox, which saves versions automatically, Git makes you *manually* commit files.

At first glance, this may seem less practical than automatic commits; however, it lets you save a version of your file *at meaningful points in time*, for example when you have added a new feature to your code or completed a section in your paper.

Moreover, manual commits let you decide which files you want to keep track of (and which not), i.e. which files are part of the repository.

Each time you make a commit, Git asks you to add a description about the changes you have to your file, which makes it much more pleasant to to track old versions than in Dropbox.

When programming, it is common to have multiple files that you work on simultaneously. A commit can include multiple files, i.e. you save a version of all files simultaneously.

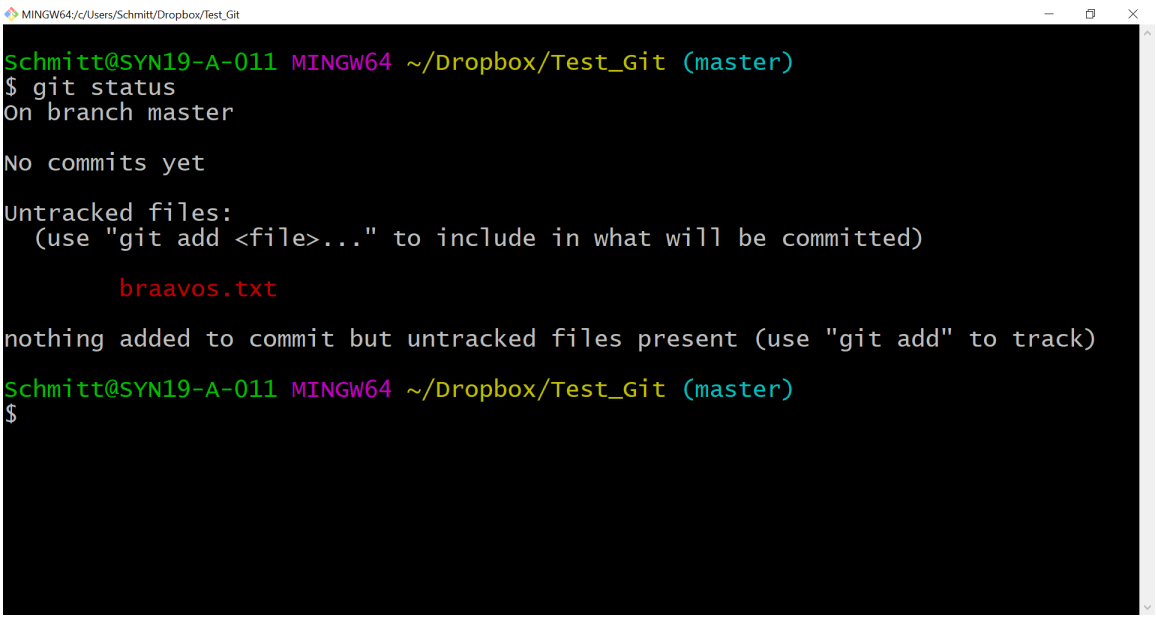
git status

An important command is **git status**. It allows you to see what files in your current directory have changed since their last commit. Moreover, it displays *untracked files*: these are files which are in your directory, but have not been added to a repository. Hence, they won't be affected by commits.

In the example below, the directory contains a txt-files (**braavos.txt**), which has not been added to a repository yet, hence it is untracked.

In [7]:
```
display.Image('git2.png')
```

Out[7]:



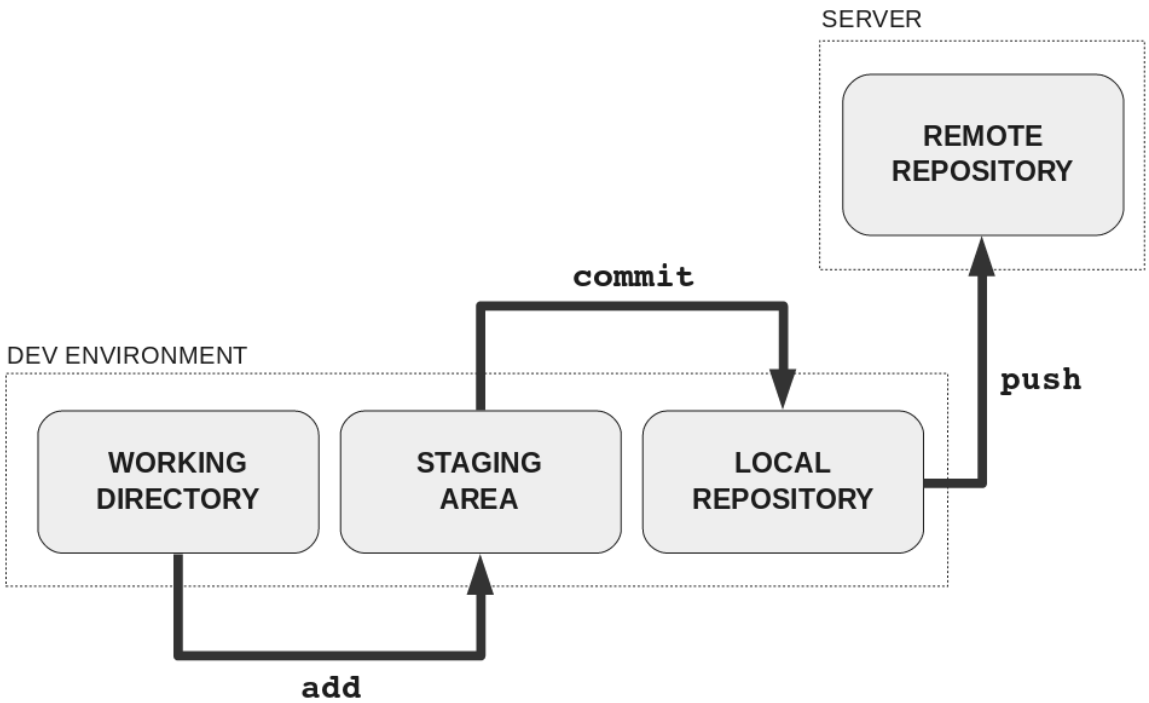
git add

Committing a file to a repository is a two-step process. First, we use **git add *file*** in order to add a file to the *"staging area"*.

This does not mean that Git moves the file to a different location; loosely speaking, it adds a copy of the file to a directory called the staging area.

In [8]:
```
display.Image('git_schema2.png', width = 600)
```

Out[8]:



Note that this doesn't commit changes to the file yet. Rather, we can see that **braavos.txt** is no longer "untracked", but now appears under "Changes to be committed".

In [9]:
```
display.Image('git3.png')
```

Out[9]:

git commit

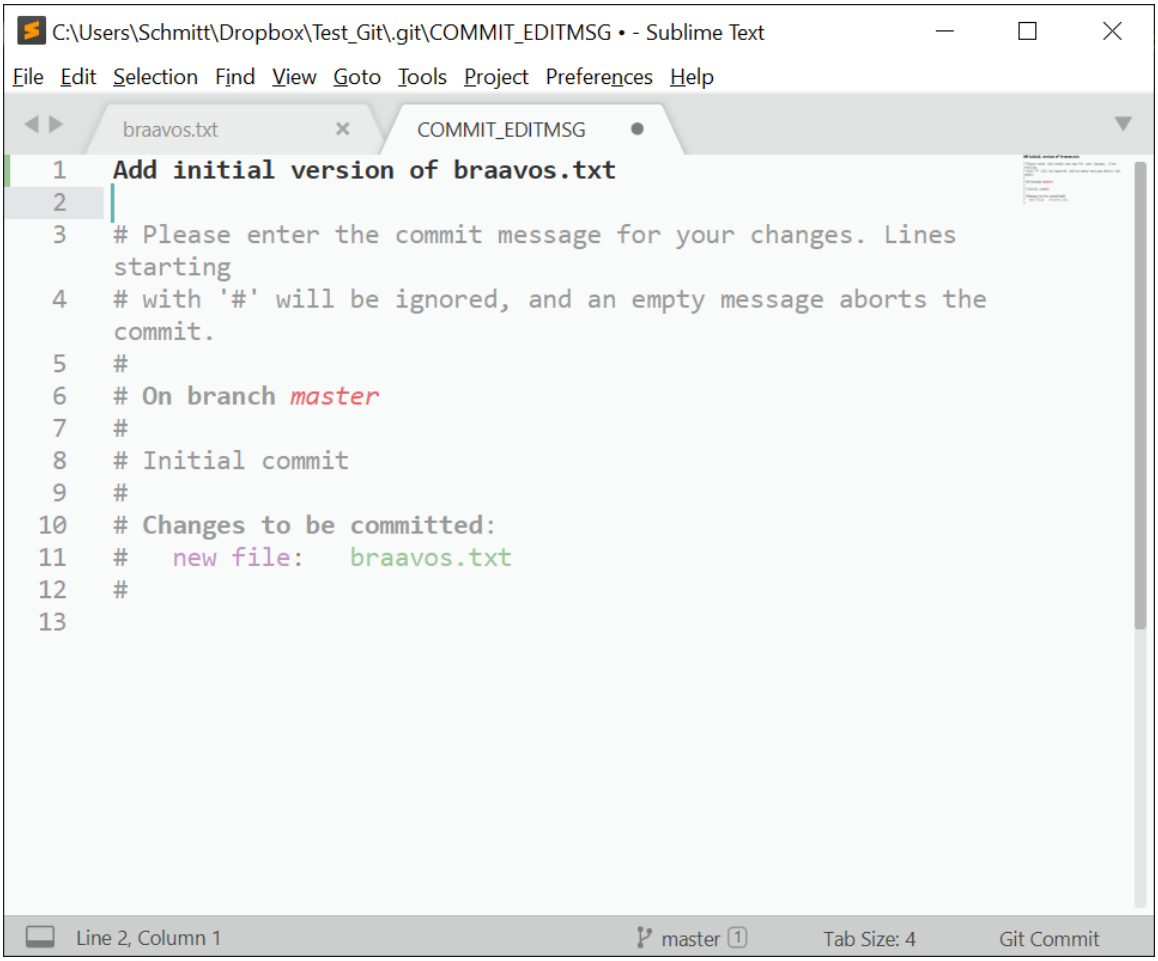
Once we have added a file to the staging area, we can use **git commit** to commit the changes made to the file. Essentially, this moves the file from the staging area to the repository.

At this point, you can think of two copies of the same file existing: one in the working directory, and one in the repository.

Before executing the commit, Git will ask you to provide a commit message. Such a message should contain a description of the logical changes that you have made to the file. Depending on your setup, you can write this message in an external editor (recommended) or directly in the terminal.

In [10]:
```
display.Image('git4b.png')
```

Out[10]:



In [11]:
```
display.Image('git4.png')
```

Out[11]:

A few things are important to note here:

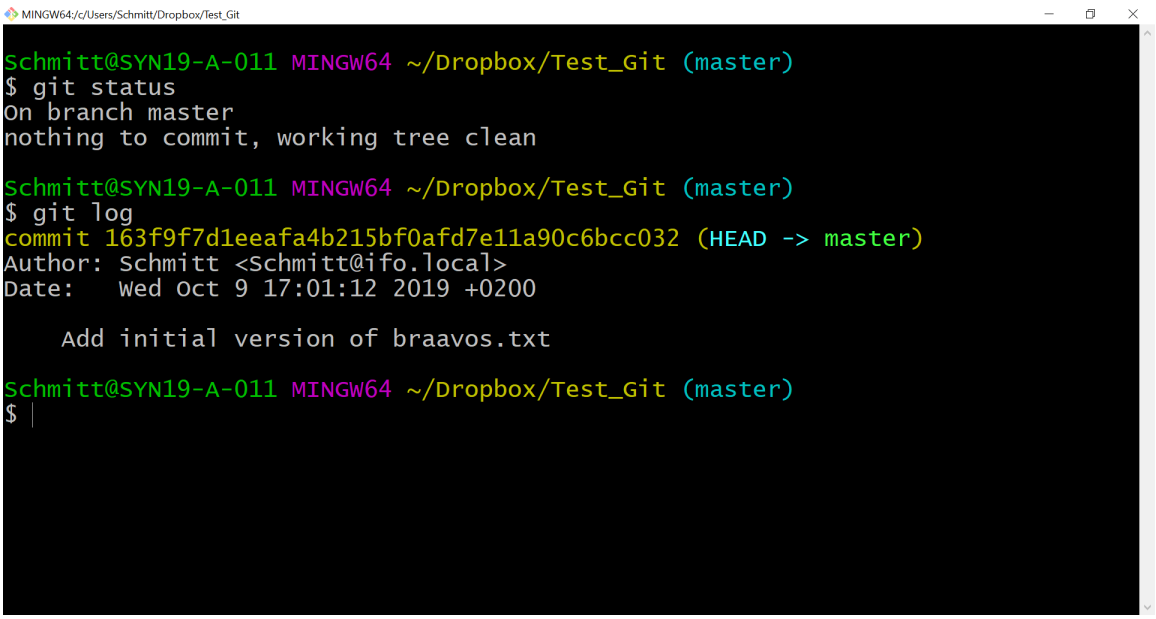
- Using **git commit** commits all changes made to the files *which are currently in the staging area*. If you have made changes to another file that you don't want to commit yet, just don't add it to the staging area.

- This is also the motivation for having this two-step process of adding and committing: it gives you more control about which changes you want to include in a single commit. A good rule of thumb is to keep the number of logical changes in a single commit small (ideally one logical change per commit).

- Committing a *previously untracked* file adds it to the repository.
- If we use **git status** now, we can see that **braavos.txt** no longer appears. In general, files that are tracked and have not been changed since their last commit will not show up in the status message.

In [12]:
```
display.Image('git5.png')
```

Out[12]:



git log

We can display the history of commits using **git log**. It will show the author and time of each commit, as well as our commit message.

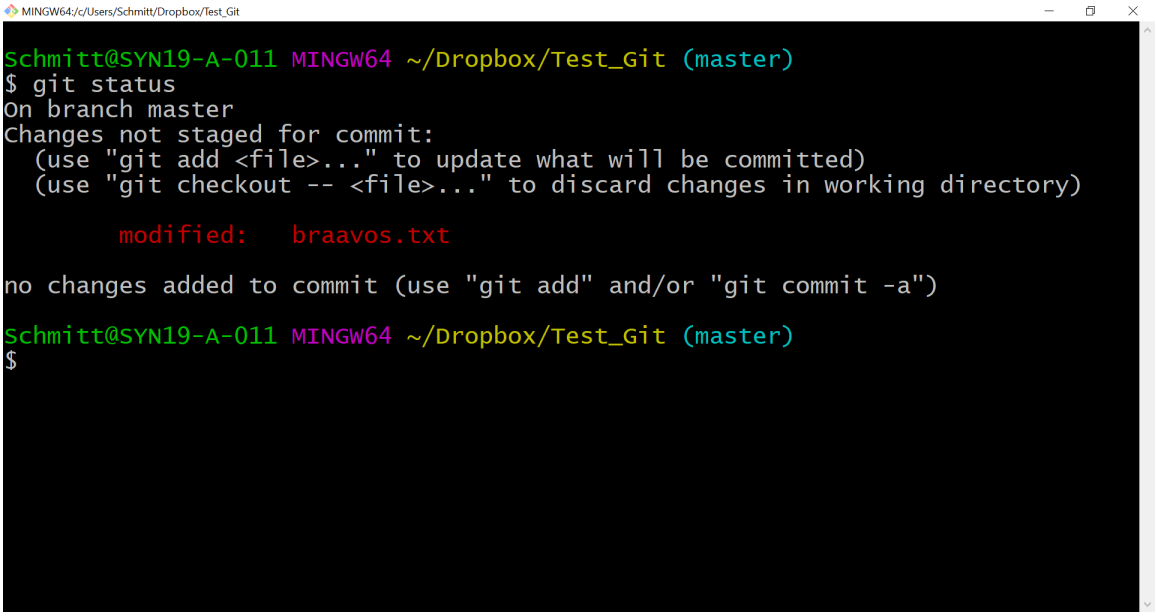
Moreover, each commit has an ID (in the first line of the log output), which will be important when accessing old versions of the file, as outlined below.

git diff

When changing a file, we can use **git diff *file*** to compare its current version to the most recent *committed* version. For the example, I added a line to **braavos.txt** and then run **git status** to verify that it has changed.

In [13]:
```
display.Image('git6.png')
```

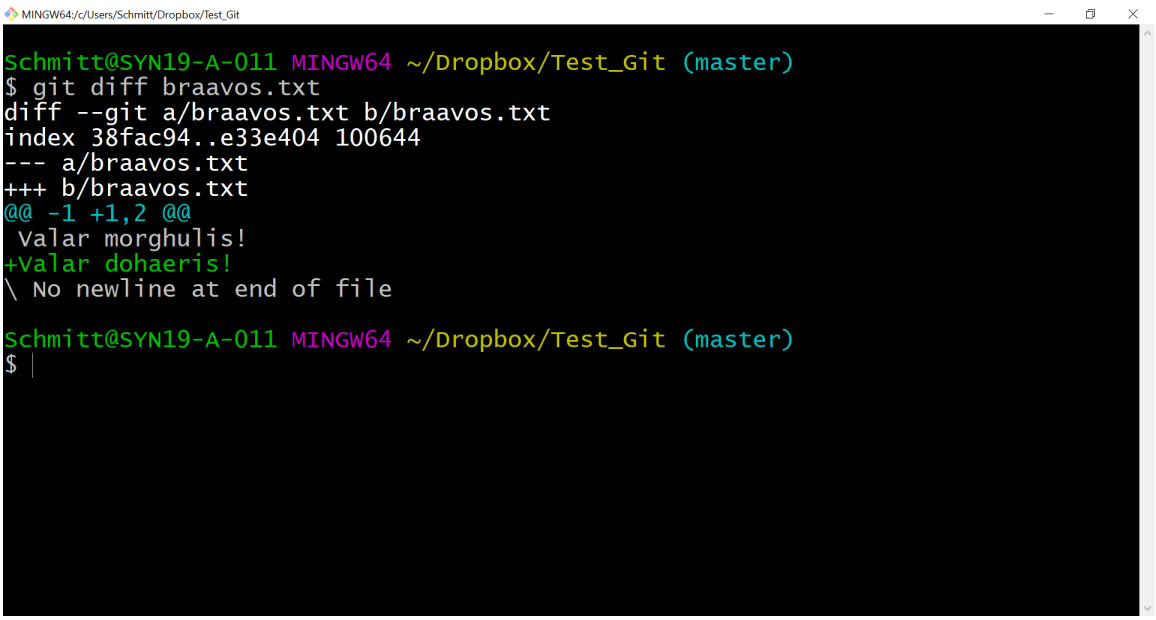
Out[13]:



When running **git status**, Git essentially compares all tracked files in the working directory with their versions in the repository. In the case of **braavos.txt** above, it discovers a difference, as the file was changed in the working directory, but the version in the repository has not yet been updated.

We can use **git diff** to compare these two versions:

In [14]:
```
display.Image('git7.png')
```

Out[14]:


In the output, lines that have been changed in the new version are marked with a "+", while their counterparts in the old version are marked with a "-".

Note that running **git diff** without a file name compares all files in the repository that have been changed since their last commit.

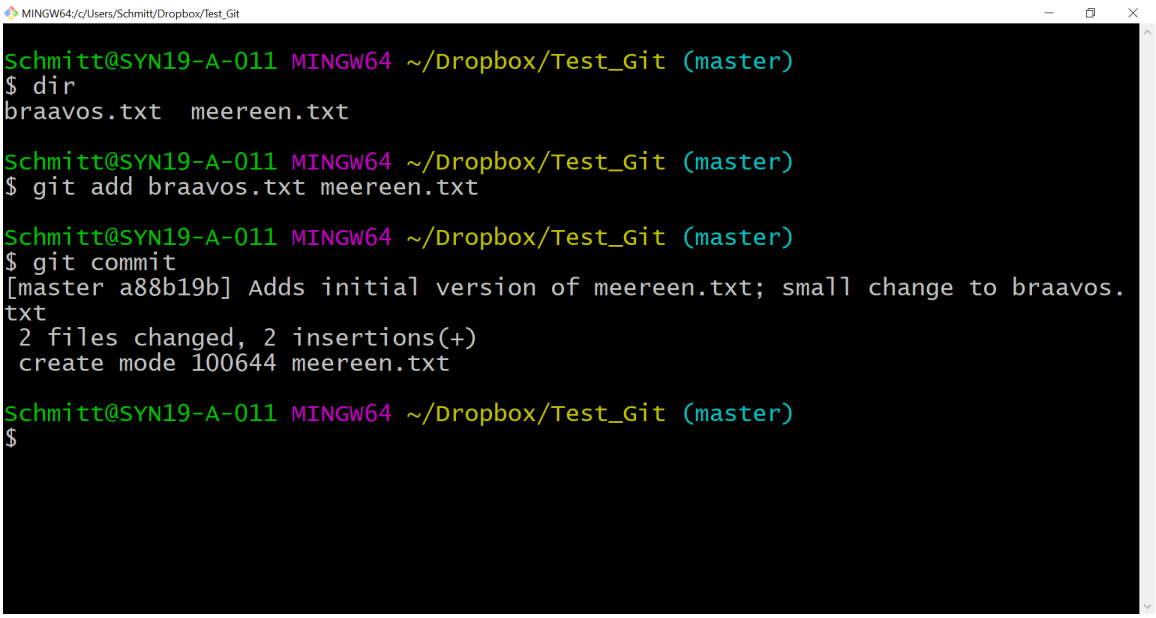
Committing multiple files

In the following, I want to commit both the change made to **braavos.txt**, as well as add the new and untracked file **meereen.txt** to the existing repository.

In both cases, as seen above, I need to add the file to the staging area and then commit them. I could do this two-step process individually for both files. Alternatively, after adding both files the staging area, I can commit them together.

In [15]:
```
display.Image('git8.png')
```

Out[15]:

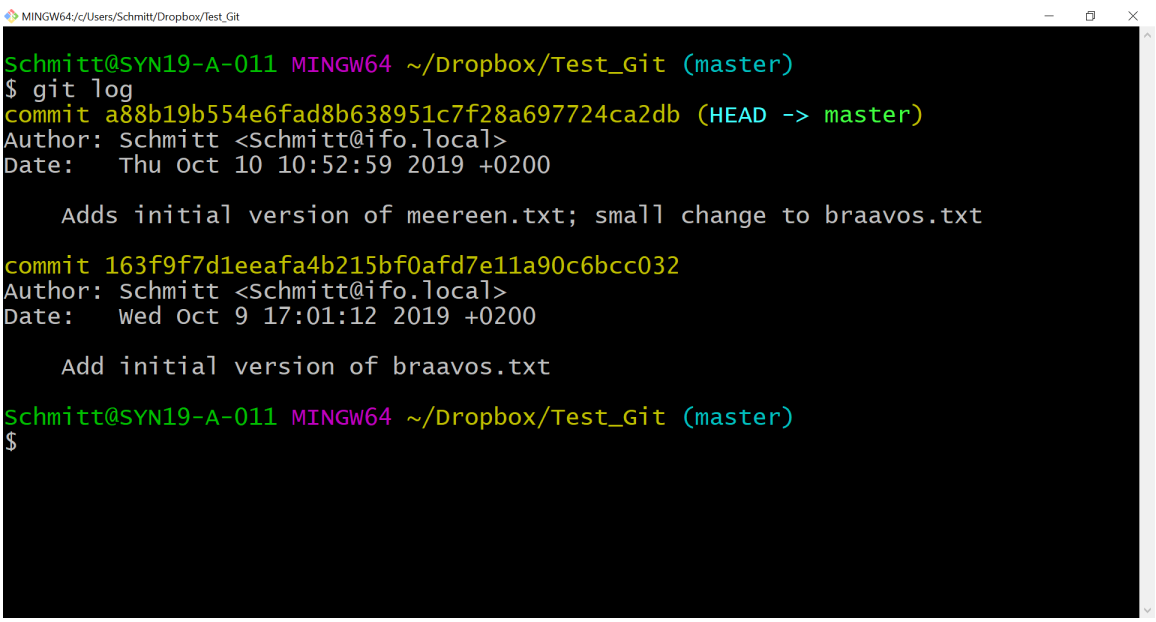


git checkout

We can use **git checkout** to temporarily change a file back to *how it was at the time of a commit*. In other words, this would essentially load an old version of the file in the working directory. Start by looking at the commit history.

In [16]:
```
display.Image('git9.png')
```

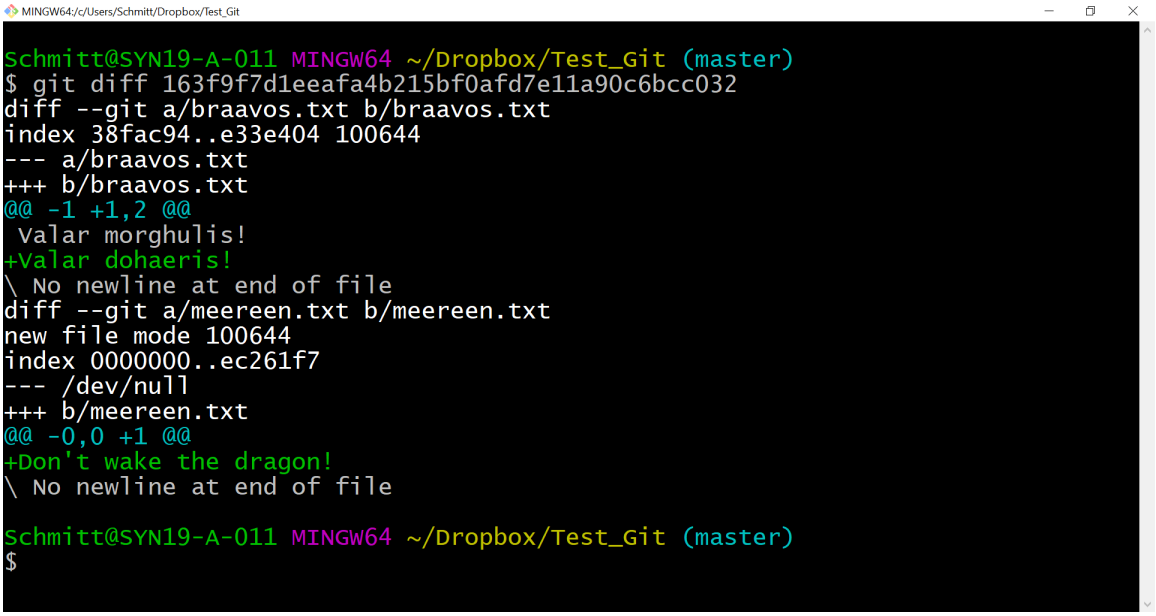
Out[16]:



In the last commit, I have changed something in **braavos.txt**. To get more information, run **git diff** plus the commit ID, which we get from **git log**.

In [17]:
```
display.Image('git12.png')
```

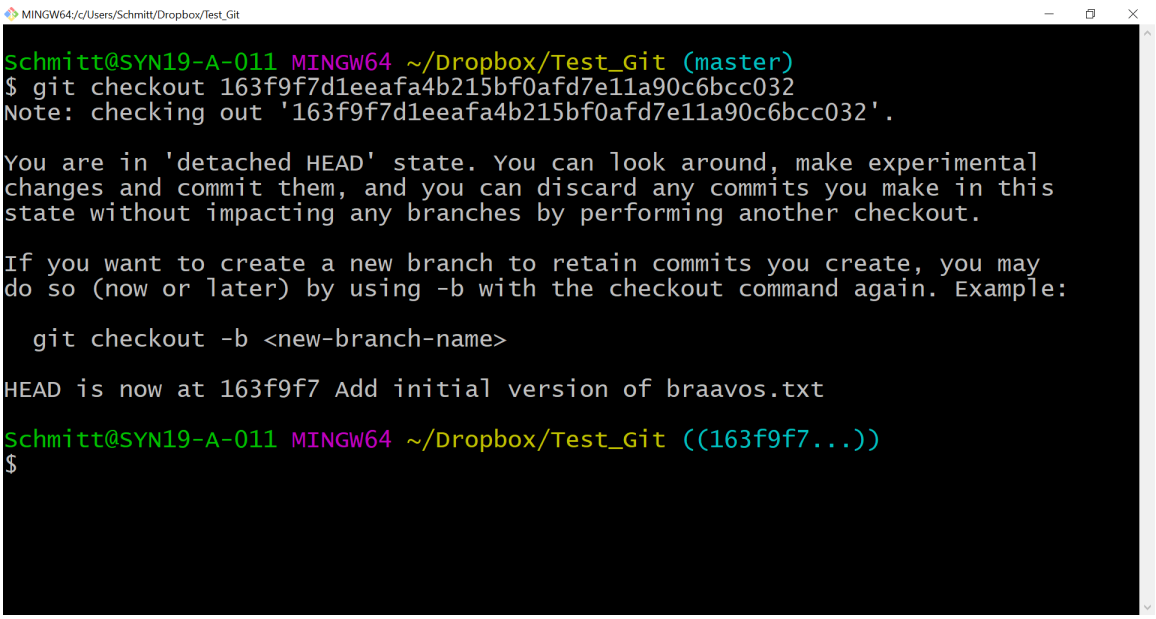
Out[17]:



If we want to go back to how the file was at the second-to-last commit, we need to run **git checkout** plus the commit ID.

In [18]:
```
display.Image('git10.png')
```

Out[18]:

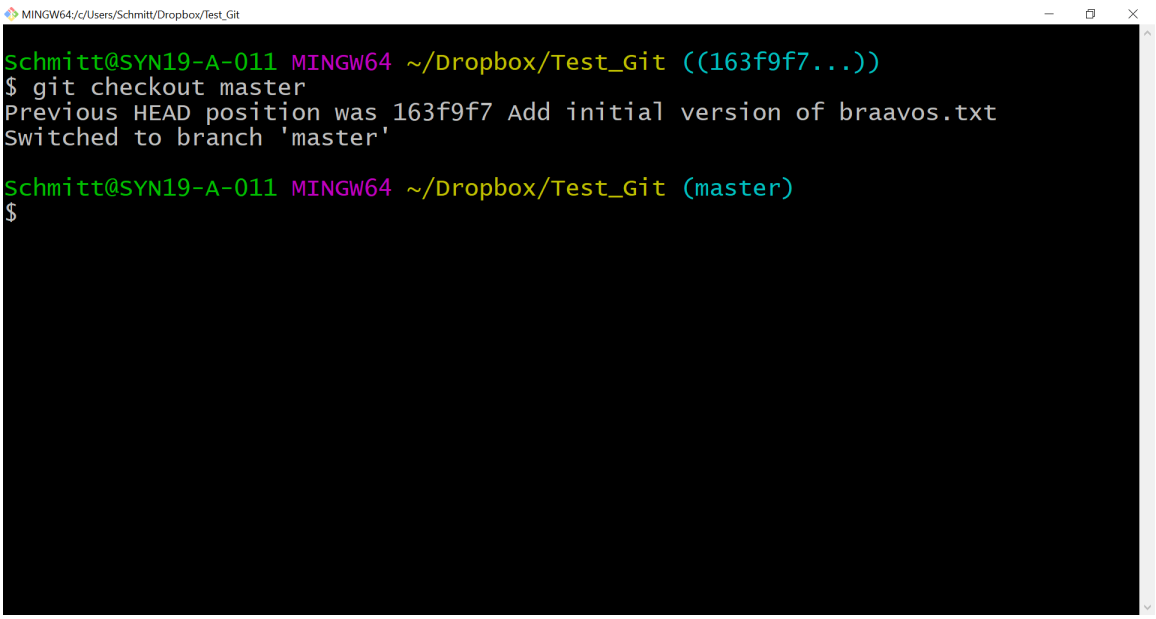


A few things are important to note here:

- When checking out an older version of a file, Git tells you that you are in a "detached HEAD" state. This basically means that your files are different compared to your most recent commit.
- You can go back to your most recent commit by running **git checkout master**.
- Note that if you check out a previous commit, *all* files in your repository that have been changed since that commit will be restored to how they were at the time of this commit.

In [19]:
```
display.Image('git11.png')
```

Out[19]:



Finally note that running **git checkout** plus the name of a file in the staging area (before it is committed), it would remove this file from the staging area.

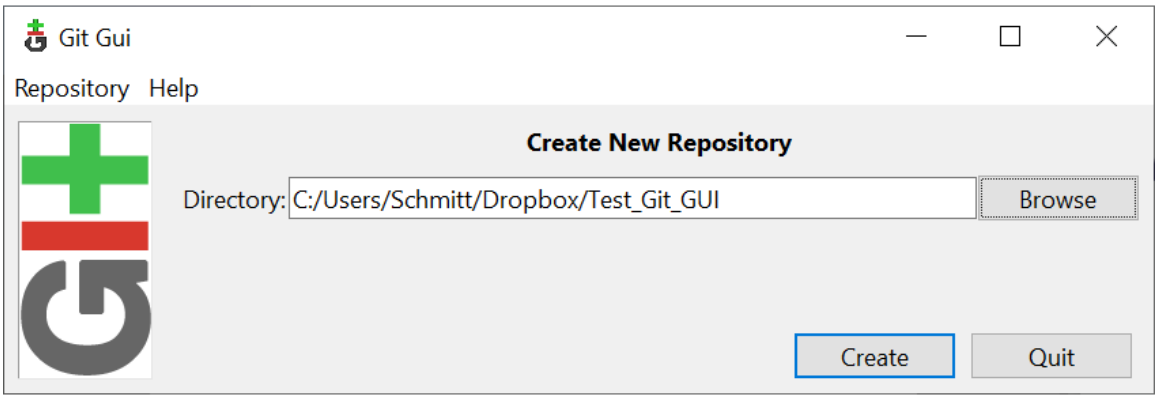
Using Git GUI

On Windows, you can start Git GUI from your Start menu. On Mac, ...

All the concepts introduced above - **git init, git add, git commit, git checkout** - are implemented in the GUI.

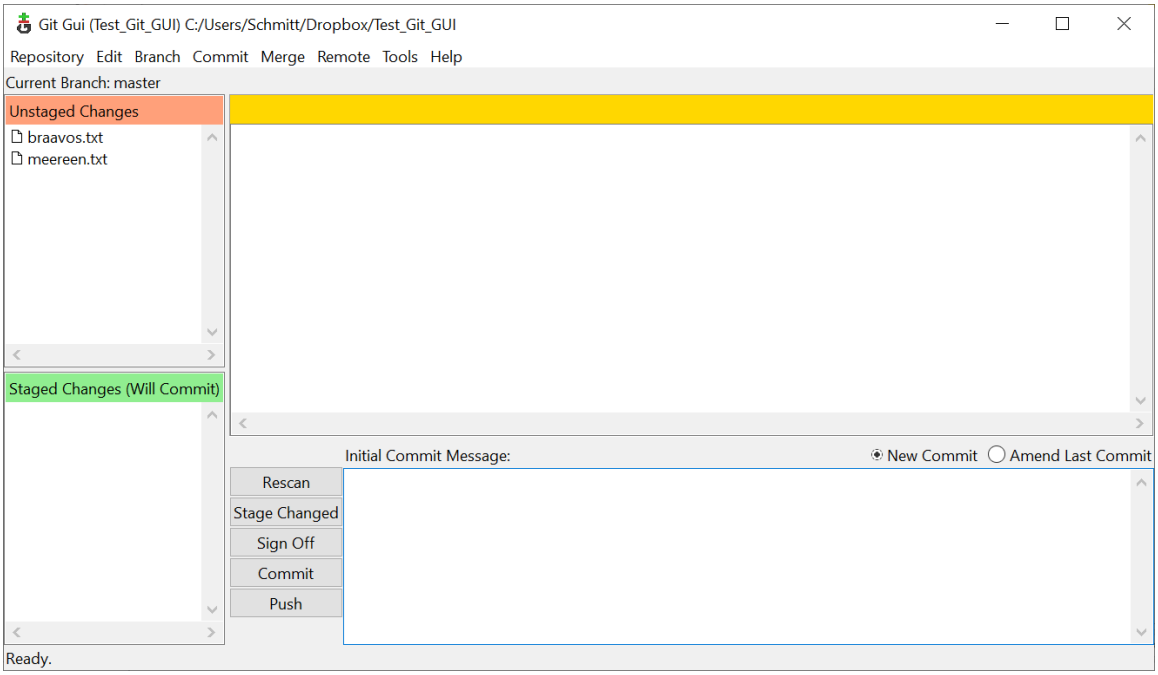
In [20]:
```
display.Image('git_gui1.png')
```

Out[20]:



In [21]:
```
display.Image('git_gui2.png')
```

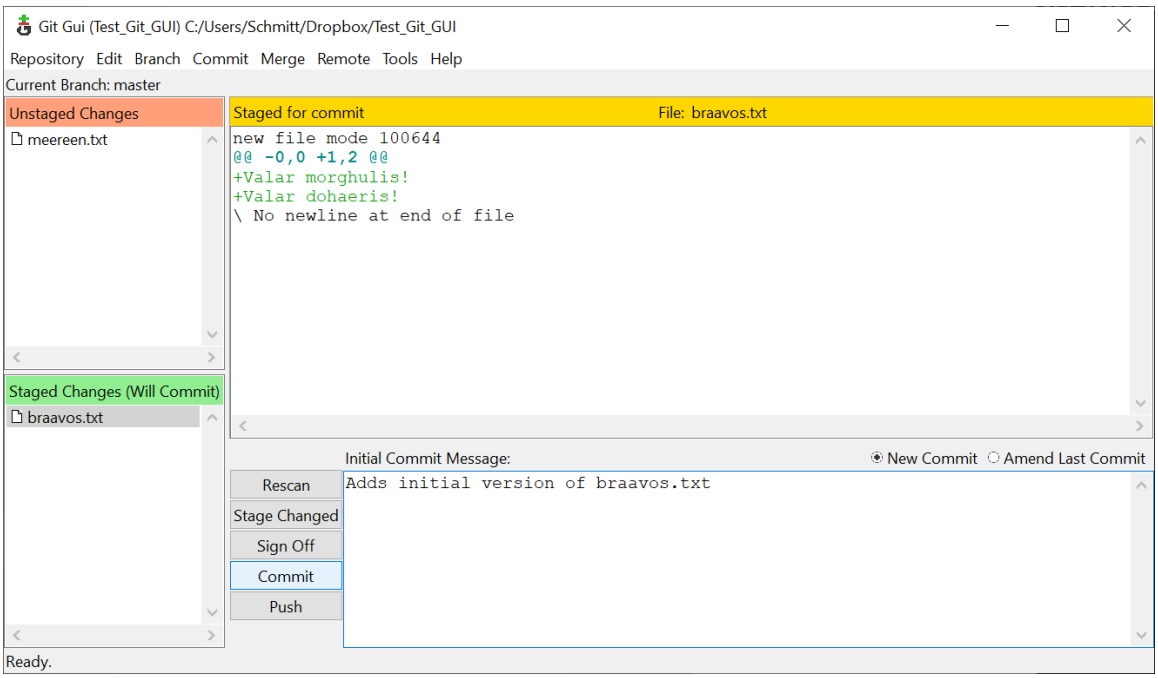
Out[21]:



Adding a file to the staging area can be done with the key combination **Crtl+ T** (**Strg** + **T** on a German keyboard) or via the "Commit" tab in the menu bar.

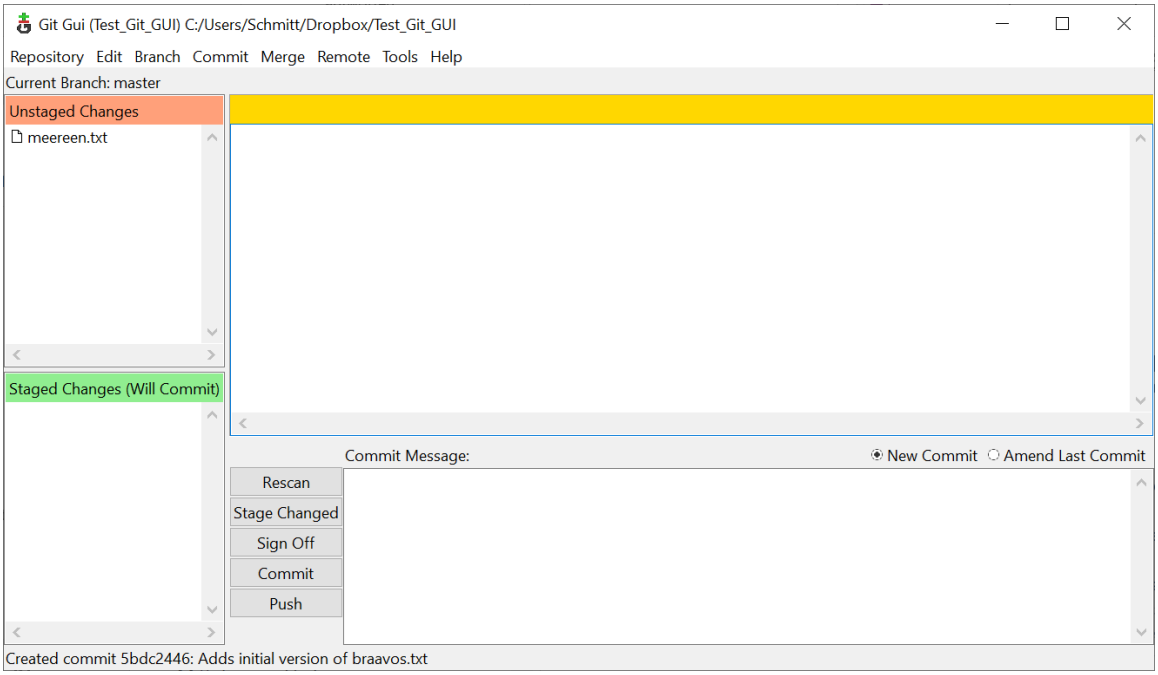
In [19]:
```
display.Image('git_gui3.png')
```

Out[19]:



In [22]:
```
display.Image('git_gui4.png')
```

Out[22]:



Other important Git commands

In addition to the fundamental concepts and commands introduced so far, there are other important, but slightly more advanced topics when using Git, in particular the concept of *branches*. For time reasons (and because they are not essential for getting started with Git), I will skip those here, but instead refer to the material linked above.

Using Git and Github/Gitlab For Downloading and Sharing Files

Git is a version control software that runs purely on your local machine and creates local repositories. Optionally, you can link them to "remote repositories", by using certain websites that essentially provide "cloud storage".

This has two advantages:

- *Cloud backup*: similar to services like Dropbox, in addition to storing your files locally, you can also store them in the cloud.
- *Collaborating with others*: multiple people can have access to a remote repository and easily share and work on the files it contains

If you make changes to the files in your local repository, you can use Git to sync it with the remote repository, so that the changes are implemented in there as well.

Similarly, if someone else has made changes in a remote repository, you can use Git to update your local repository (if you wish to do so).

As with commits, the main difference between Dropbox and the Git/remote repository workflow is that you can decide when the syncronization between your repositories takes places, rather than it happening automatically.

A very popular remote repository site, especially in the context of software development, is *Github*: https://github.com/

Other examples are Bitbucket and Gitlab. For each of these sites, you can get a free account and then create your own remote repositories.

For this course, we will make use of the LRZ which provides a Gitlab server to all users. All the Jupyter notebooks used in the lectures and tutorials will be stored in a (internally) public remote repository on LRZ-Gitlab:

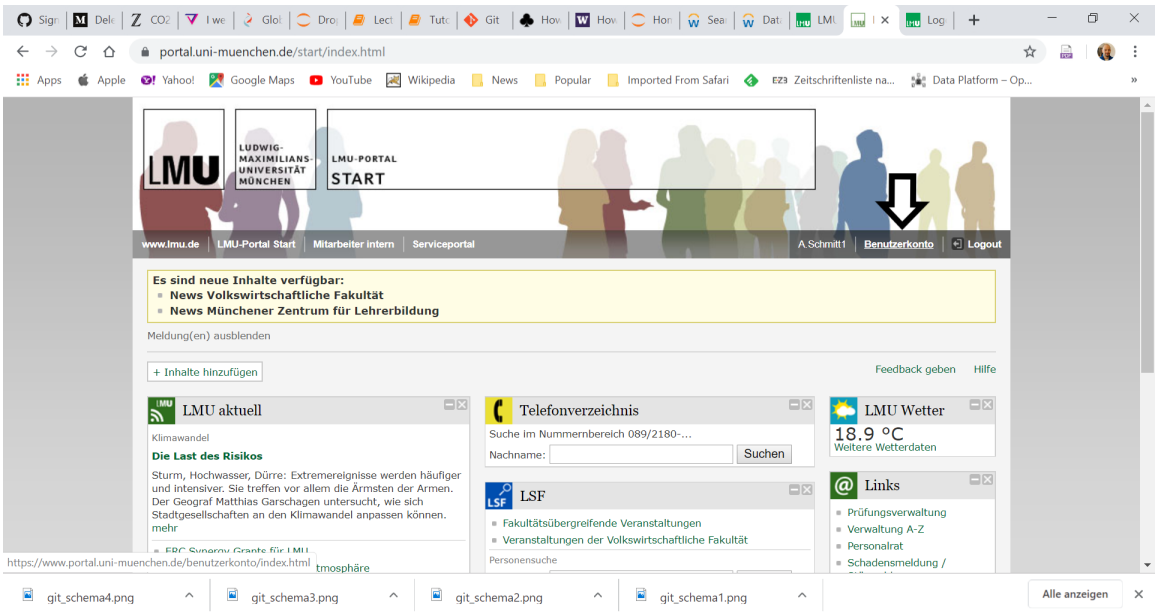
https://gitlab.lrz.de/ru67zef/cme-lmu

The great thing about this is that as an LMU student, you already have access to LRZ-Gitlab. Moreover, instead of servers located somewhere in the world, all files you upload there will stay within LMU/LRZ.

You will need your "LRZ-Kennung" to access LRZ-Gitlab. If you do not know it, go to www.lmu.de and click on **LMU Portal**. Log in with your LMU user name and password, and then click on **Benutzerkonto** in the menu bar. The following page shows your LRZ-Kennung.

In [23]:
```
display.Image('lmu1.png')
```

Out[23]:



There are two ways to access the course material:

1. You can go the Gitlab page and just download it manually, either everything at once or as individual files, similar to how you would do it in LSF.
2. For a faster and much more convenient way, you can (and should!) use Git.

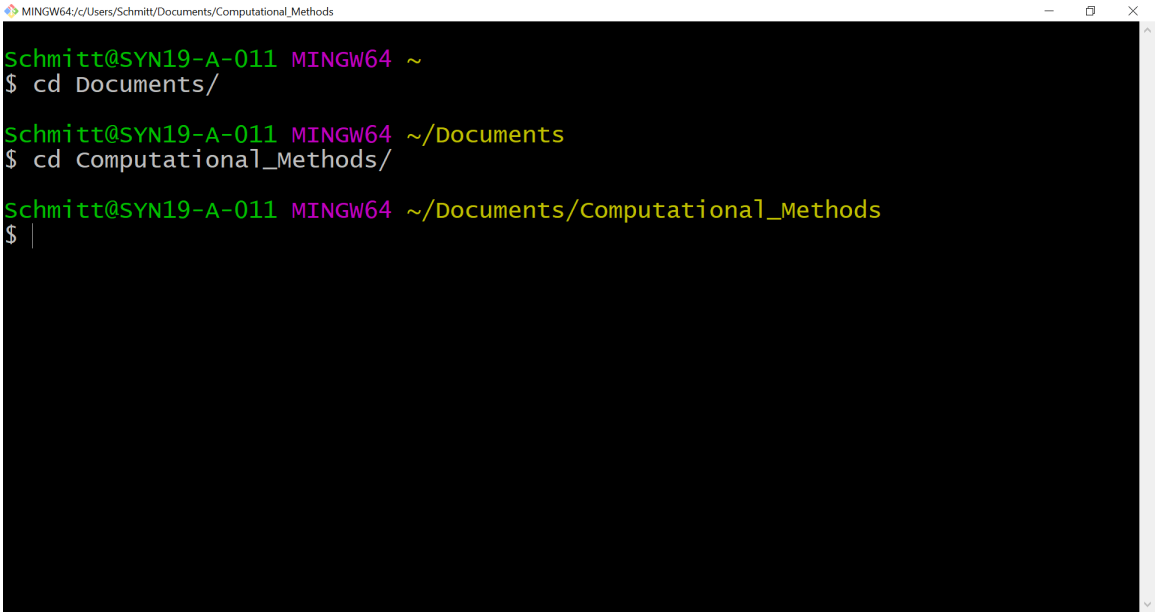
Cloning a remote repository using Git

As a first step, open Git Bash and navigate to the directory in which you want to have the course material.

For illustration, I have a directory called **Computational_Methods** within my **Documents** folder.

In [24]:
```
display.Image('git13a.png')
```

Out[24]:



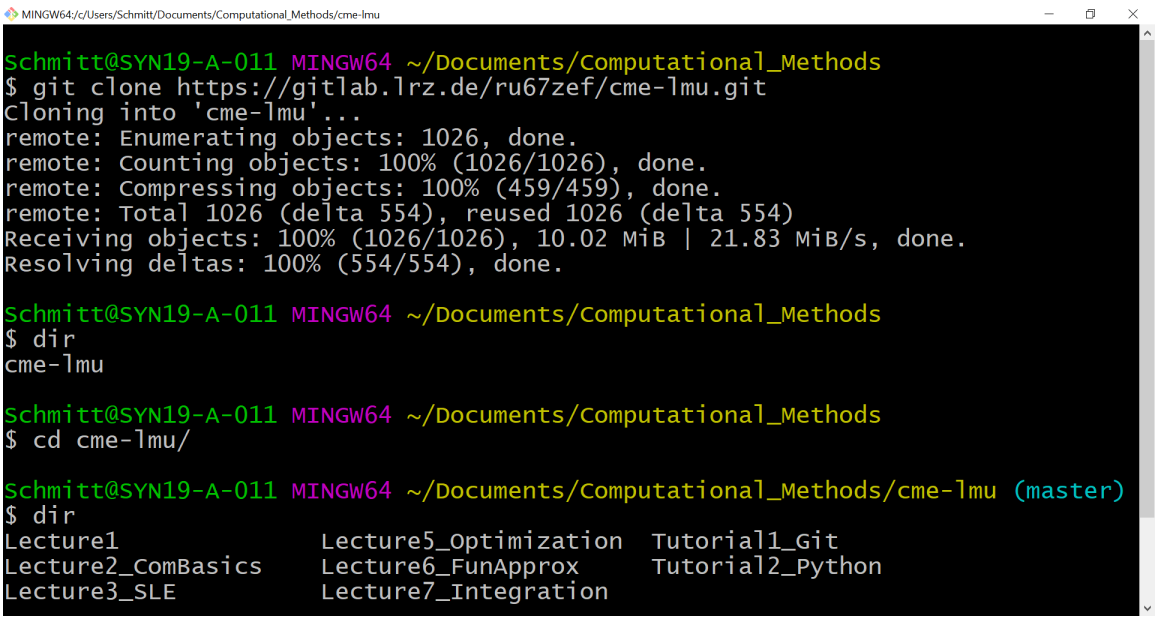
Next, type **git clone** plus the link to the remote repository provided above to clone it to your local hard drive:

git clone https://gitlab.lrz.de/ru67zef/cme-lmu.git

At this point, you will see a window asking for authentication: type in your LRZ-Kennung (see above) and your LMU password.

In [25]:
```
display.Image('git13.png')
```

Out[25]:



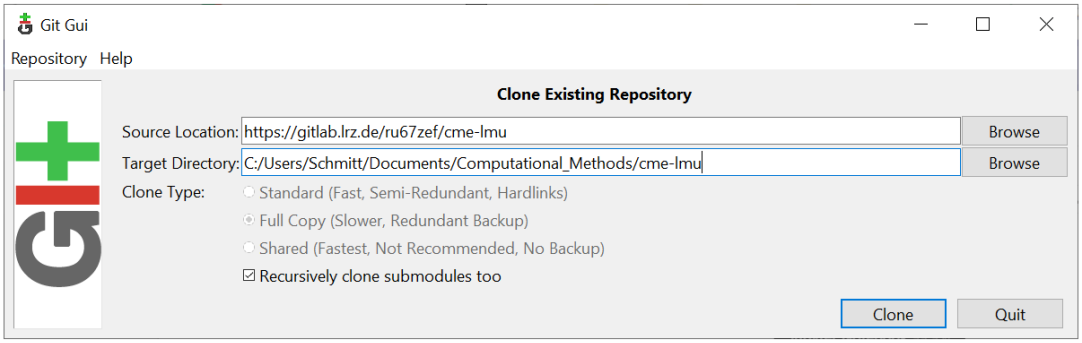
After executing this line, a folder **CME-LMU** should appear in your working directory, which contains the notebooks and other files that will be used in this class.

When using Git GUI, open it and choose the option **Clone Existing Repository**. The **Source Location** is the URL of the repository, while the **Target Directory** is your working directory.

Note that in contrast to Git Bash, you also need to specify the name of the folder in which the contents of **CME-LMU** is downloaded into. I recommended to just give it the same name, to avoid confusion.

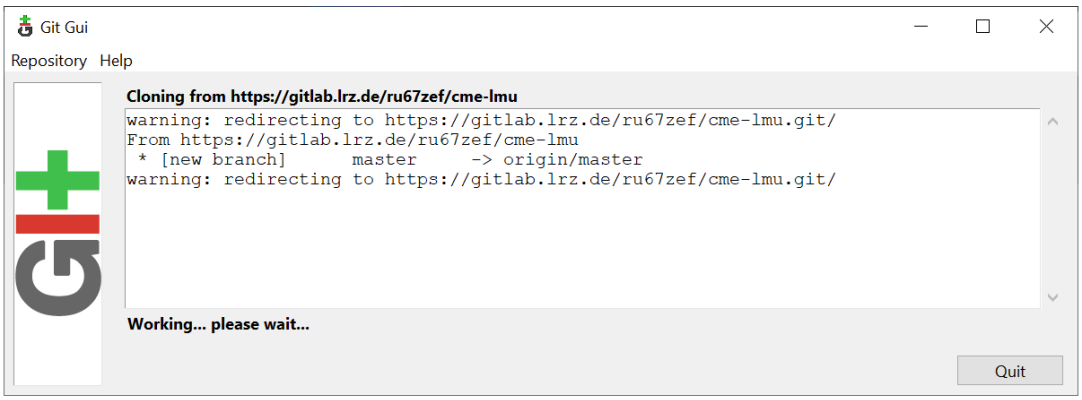
In [26]:
```
display.Image('git_gui5.png')
```

Out[26]:



In [27]:
```
display.Image('git_gui6.png')
```

Out[27]:

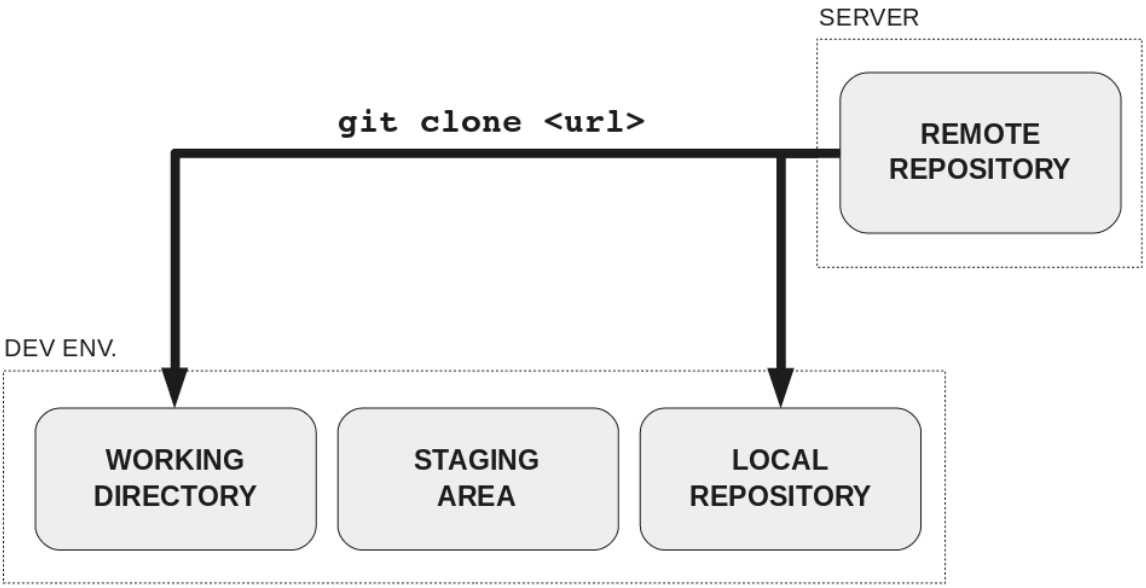


Cloning a remote repository essentially does two things:

- it downloads all the files into your working directory; and
- it initializes a *local* repository on your machine that also contains copies of all the files in the remote repository

In [28]:
```
display.Image('git_schema3.png', width = 600)
```

Out[28]:



Note that you have not only downloaded all the files in the repository. You can also access the commit history, using **git log** as seen above. This can be very handy when cloning a repository which contains files that are work in progress.

You can check whether or not a local repository is linked to a remote repository by typing **git remote**. It will display the name of the remote repository (or repositories). The default name of the first remote repository that it has been linked to is **origin**.

Pulling from a remote repository

Importantly, cloning a repository allows you to download everything at once, rather than each file individually, as you would do with the "conventional" way of downloading files manually.

While this is already a nice benefit, the real advantage of using Git and Gitlab is the following: suppose that I upload a new version of a notebook or another file after you have downloaded it, for example because I spot a typo or I adjust some of the content.

On Gitlab, I can easily indicate that I have made a change. But with the conventional way, you would have to check for each file individually whether you have the most recent version, and download the new version manually if necessary. This can be a bit annoying, plus you end up with several version of the same file on your local machine.

Using Git with Gitlab makes your workflow much more pleasant and easy. If I change a file, I will commit the changes to the remote repository. By running **git pull origin master** (or the **pull** command in Git GUI) you can sync your local CME-LMU repository (and your working directory), i.e. set it to the same "state" as the repository that I manage on Gitlab.

In other words, your local files will be automatically updated to incorporate all the changes that I have made since the last time you "pulled the changes from the repository".

In [29]:
```
display.Image('git14.png')
```

Out[29]:

```
MINGW64:/c/Users/Schmitt/Documents/Computational_Methods/cme-lmu
Schmitt@SYN19-A-011 MINGW64 ~/Documents/Computational_Methods/cme-lmu (master)
$ git remote
origin

Schmitt@SYN19-A-011 MINGW64 ~/Documents/Computational_Methods/cme-lmu (master)
$ git pull origin master
From https://gitlab.lrz.de/ru67zef/cme-lmu
 * branch            master     -> FETCH_HEAD
Already up to date.

Schmitt@SYN19-A-011 MINGW64 ~/Documents/Computational_Methods/cme-lmu (master)
$
```

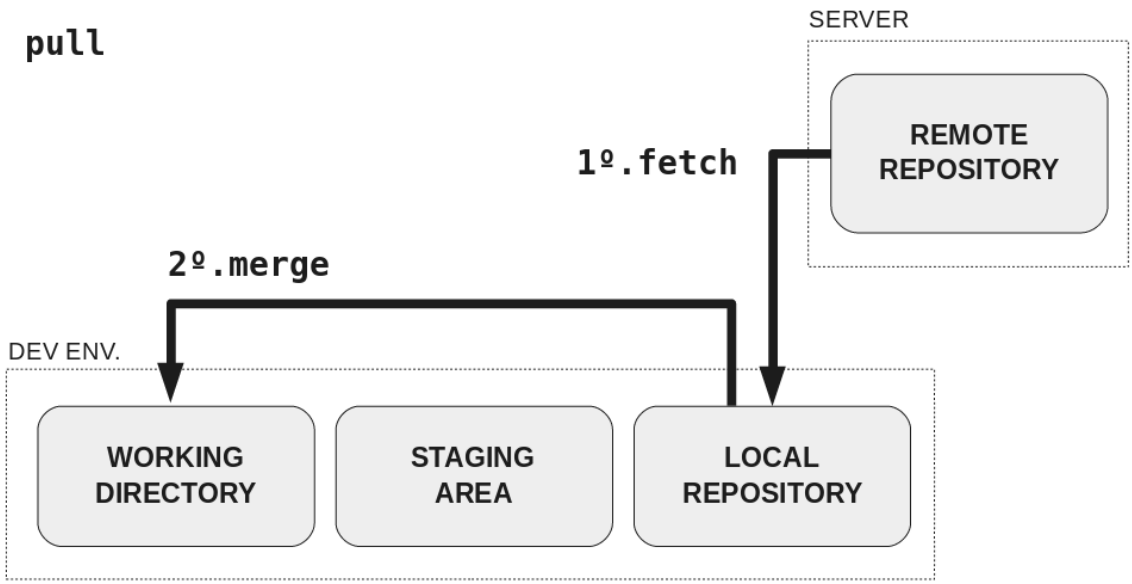
Here, I haven't made any changes to the remote repository, so pulling changes will not result in any updates.

Do not confuse **git clone** and **git pull**: the first one is used *only once*, to create a local version of a remote repository. The second one should be used in regular intervals to update the local repository.

Note also the following: in case you have already made changes to a file (e.g. inserted some text) and then pull an updated version from the remote, it will not replace your changes, but instead *merge the two versions*. Again, this will save you a lot of time managing your files!

In [30]:
```
display.Image('git_schema4.png', width = 600)
```

Out[30]:



For this class, I will not provide a complete repository of all the material right from the start. Instead, files will be added to the repository incrementally (and existing files will be updated), so using Git is the easiest way to ensure that you always have the most current version of a file.

Pushing to a remote repository

So far, we have seen how to get files from a remote repository to your local machine. If you were to use Gitlab only to download the course material, we would be done at this point.

However, I will also ask you to use Gitlab in the other direction, from your local machine to a remote repository.

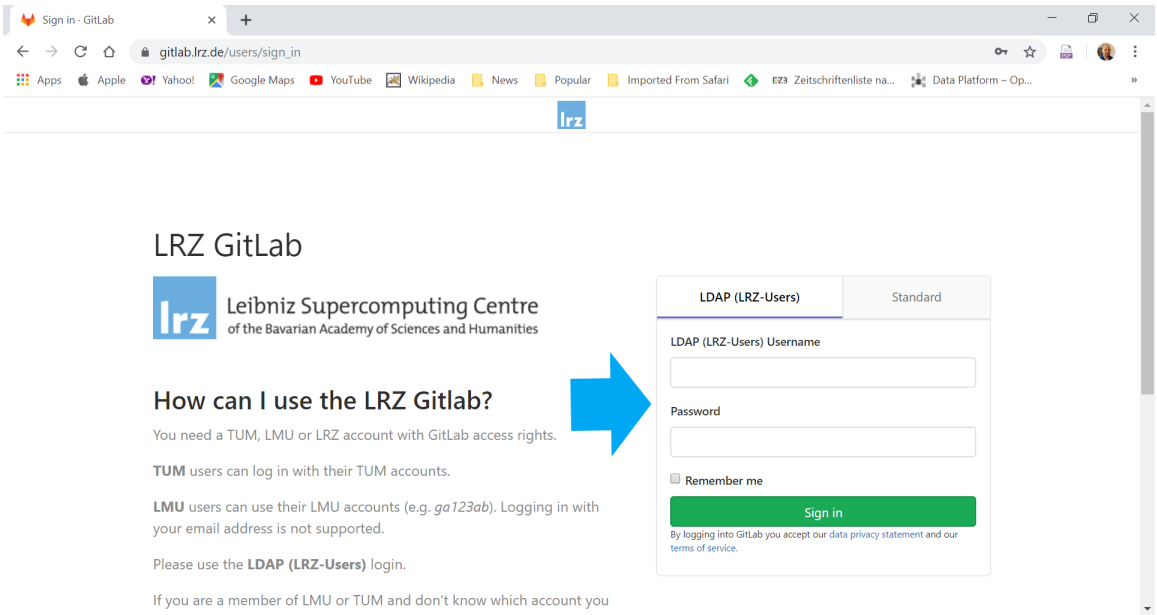
- If you want to submit solutions for the (optional) problem set, you will have to *"push"* your solution to a remote repository that only you and I have access to. This is voluntary, but highly encouraged, in particular since ...
- you *will have* to use this workflow for the exam: there will be a numerical problem that I will ask you to submit a solution for before the exam, by pushing it to a remote repository!

Before talking about "pushing", let's set up your remote repository.

1. Go to https://gitlab.lrz.de and log in with your LRZ-Kennung.

In [31]:
```
display.Image('lrz1.png', width = 1000)
```

Out[31]:



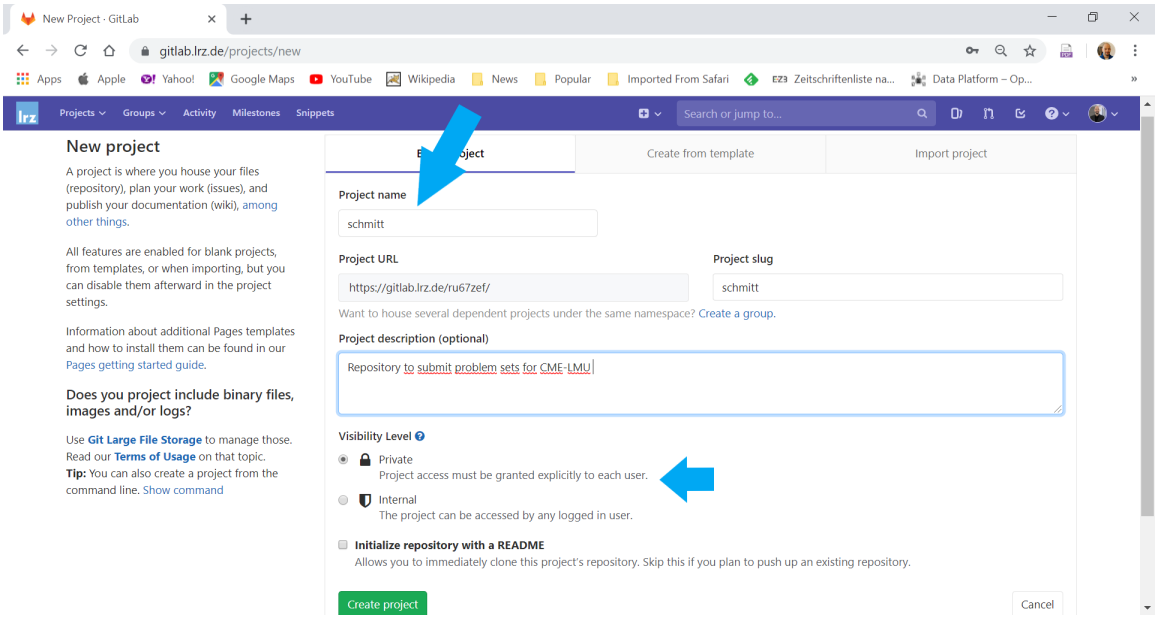
1. On the start page, click on **New Project** in the upper right corner. On the following page, you need to give your repository. Use *your last name*. In my case, the repository would then have the URL:

https://gitlab.lrz.de/ru67zef/schmitt

You can also add a description. Under **Visibility Level**, make your repository *private*. This means only you and people that you grant access can see your repository. Then, click on **Create project**.

In [32]:
```
display.Image('lrz2.png', width = 1000)
```

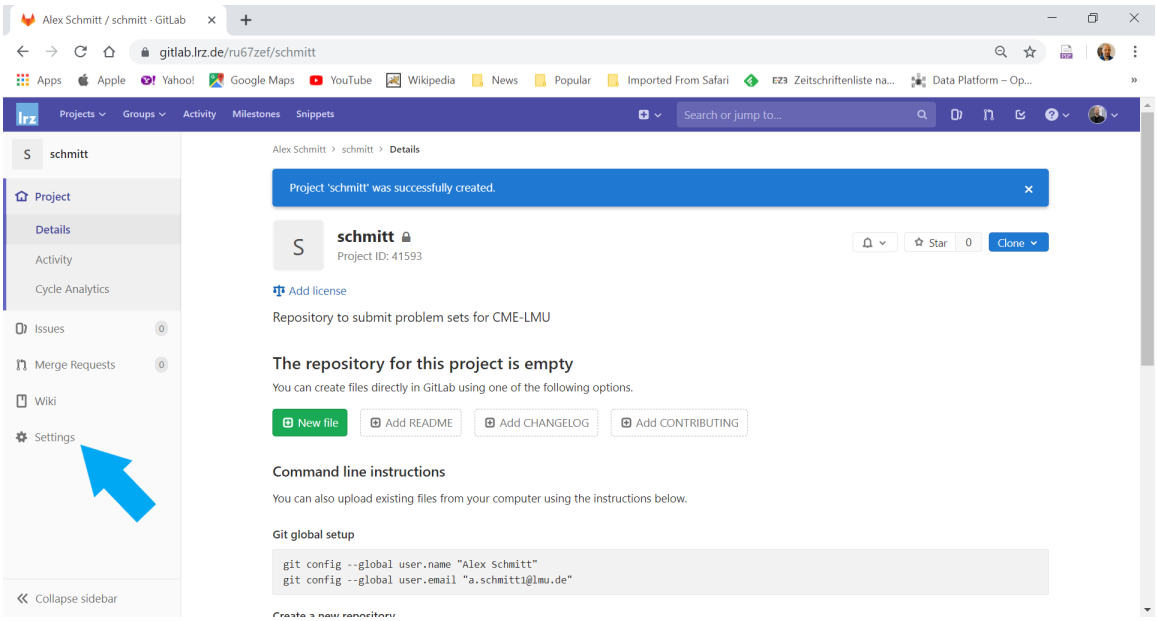
Out[32]:



1. On the next page, you should get the message that the project (i.e the repository) was created. It is empty. To give other people access, go to the **Settings** tab on the left hand side, and then to **Members**. Under **Invite member**, type my name (Alex Schmitt). Then click on the search result and then on **Add to project**.

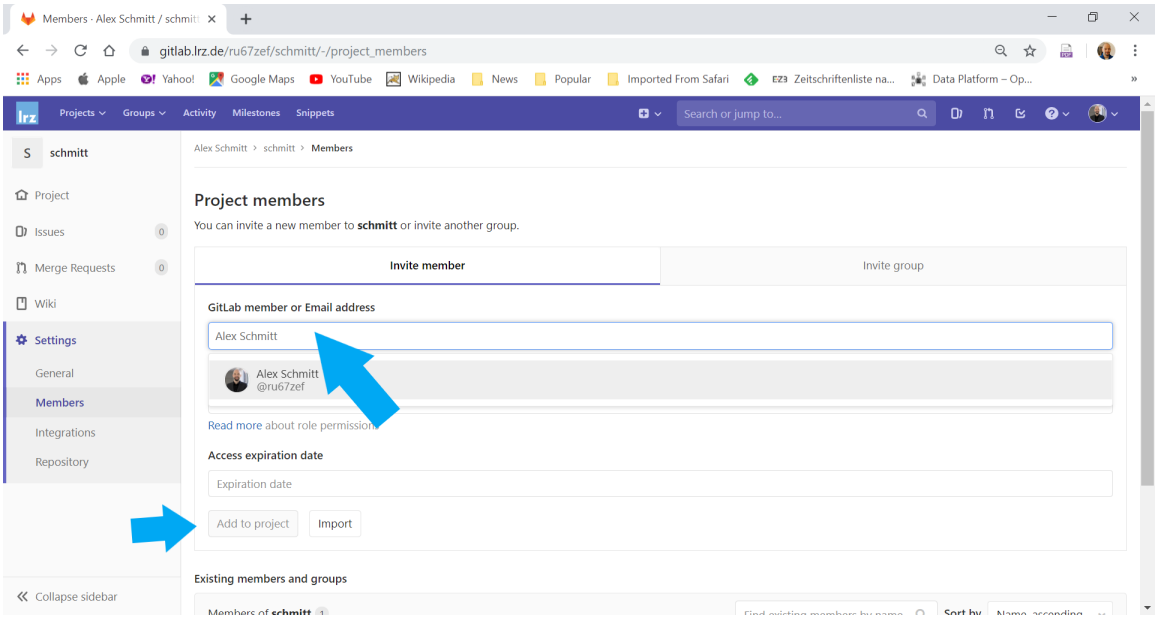
In [33]:
```
display.Image('lrz3.png', width = 1000)
```

Out[33]:



In [34]:
```
display.Image('lrz4.png', width = 1000)
```

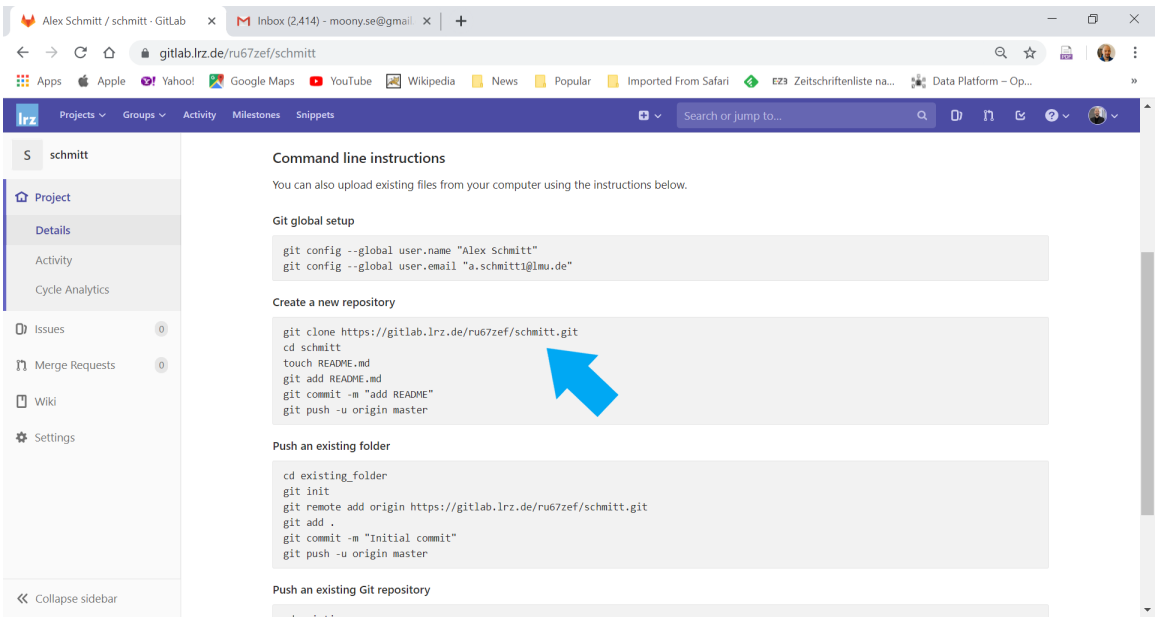
Out[34]:



1. Go back to the **Project/Details** tab. Scroll down to **Command line instructions** and then **Create a new repository**. Copy the line that starts with **git clone ...**.

In [35]:
```
display.Image('lrz5.png')
```

Out[35]:

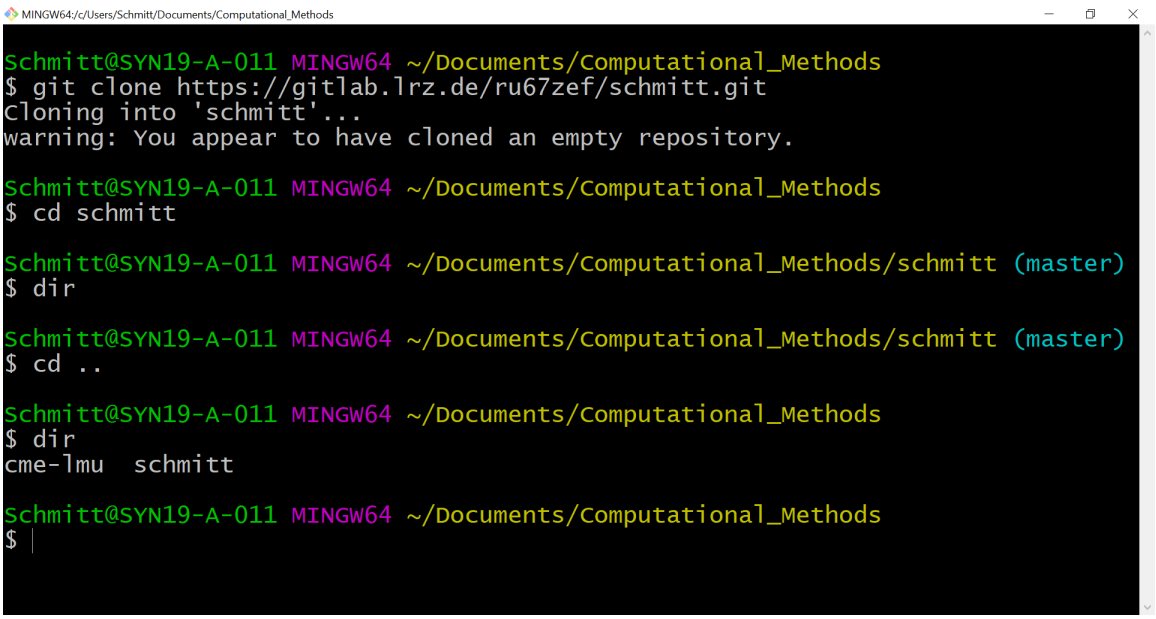


1. Open Git Bash and navigate to the directory in which you want to have the course material. It is recommended (but not mandatory) that you use the same directory in which you downloaded the **CME-LMU** repository earlier. In my case, this would be the directory called **Computational_Methods** in my **Documents** folder.

Word of warning: *do not clone your new repository in the **CME-LMU** folder. This should be a separate repository!*

In [36]:
```
display.Image('git15.png')
```

Out[36]:

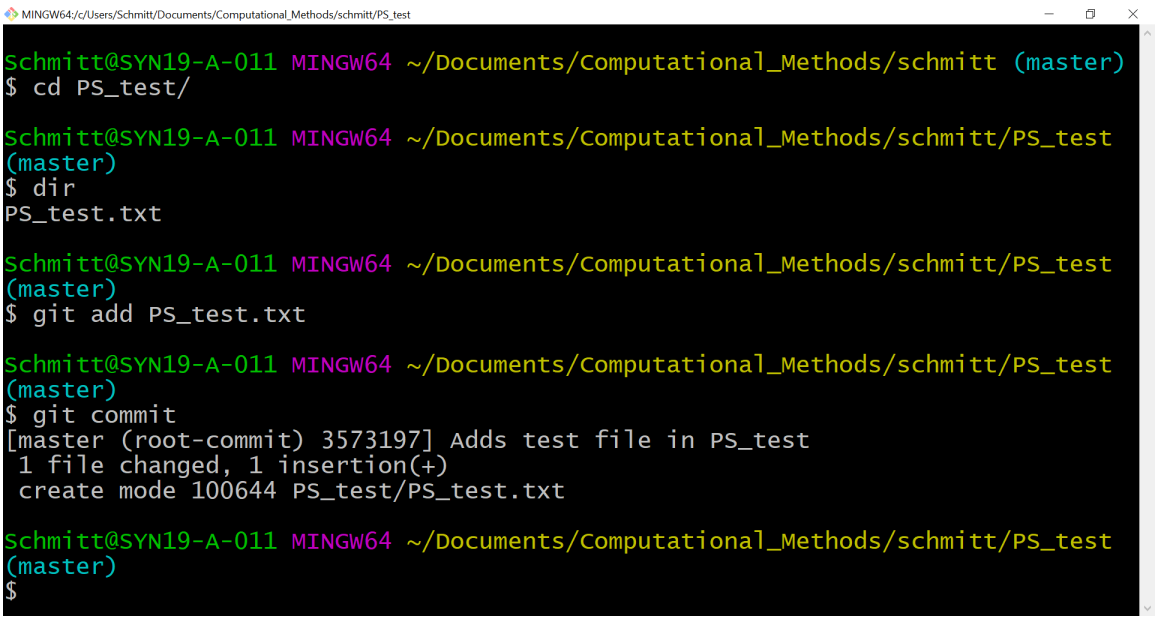


1. Finally, let's push something to your remote repository. In your working directory, create a folder called **PS_test**. Open a text editor, create a text file called **PS_test.txt** and save it in **PS_test**.

Use **git status** to confirm that the file is not yet tracked by Git. Next, use **git add** and **git commit** to commit the file to your local repository.

In [37]:
```
display.Image('git16.png')
```

Out[37]:

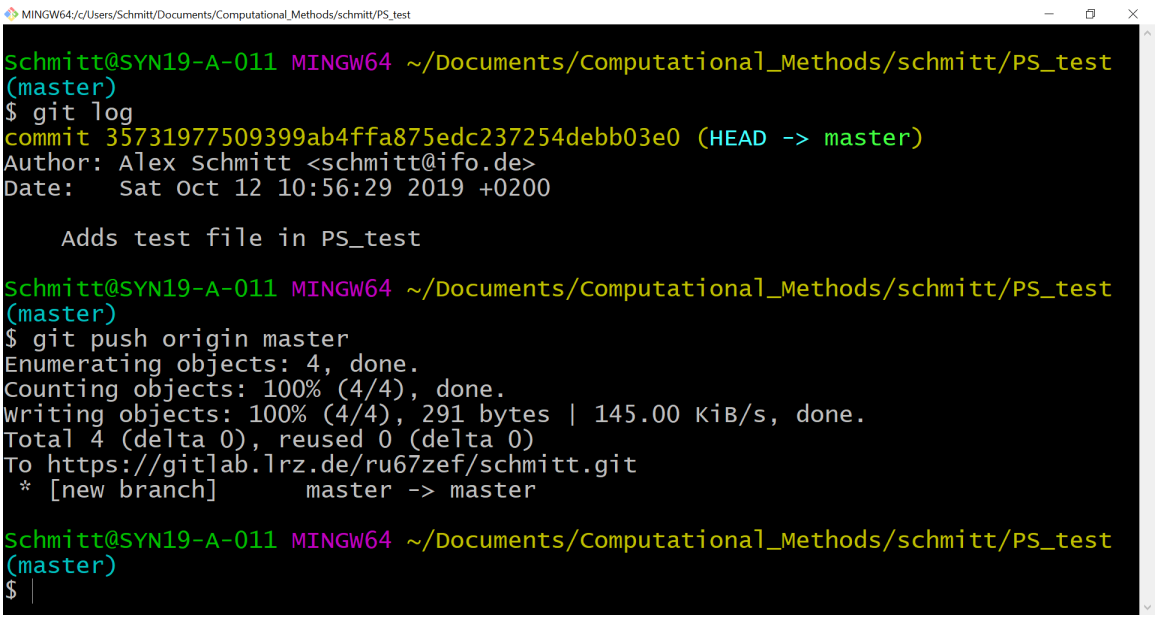


Type **git push origin master** to push the most recent commit to your remote repository (called **origin**). The message on your screen should look like on the screenshot below.

Finally, go back to your Gitlab repository and refresh the page. It should now show your commit.

In [38]:
```
display.Image('git17.png')
```

Out[38]:

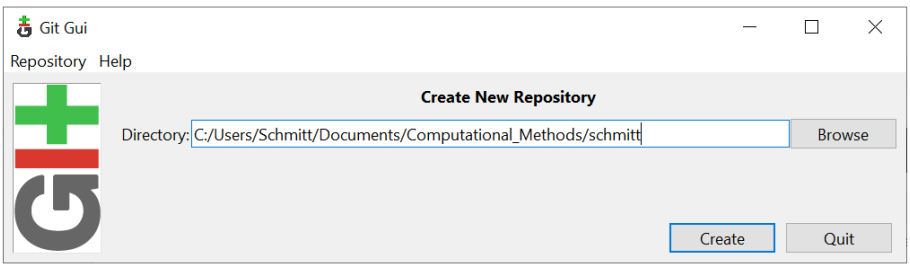


When using Git GUI, steps 1-5 above are of course the same. To clone your repository to your local machine, you could open Git GUI and use the **Clone Existing Repository** command, as above. However, for some reason that did not work on my machine (maybe because we are trying to clone an empty repository).

Instead, let's create a new local repository on your machine and then link it to your Gitlab repository. Use the **Create New Repository** option and browse to the target directory (again, using your *own name*!).

In [39]:
```
display.Image('git_gui7.png')
```

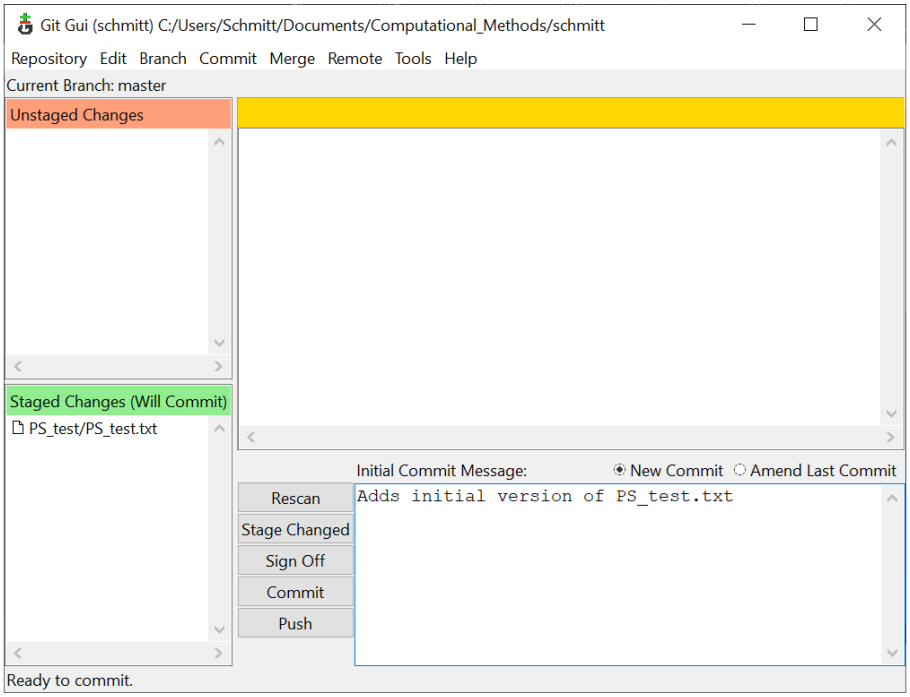
Out[39]:



As before, add a folder **PS_test** with a text file **PS_test.txt**, and commit it.

In [40]:
```
display.Image('git_gui8.png')
```

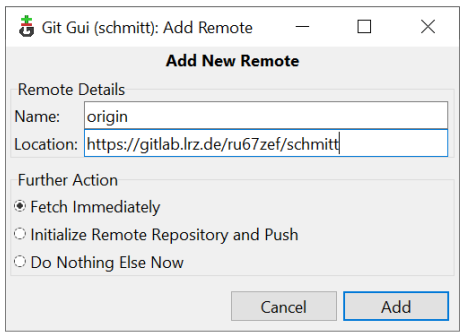
Out[40]:



Next, click on **Remote** in the menu bar and then on **Add..**. In the new window, type the name of your remote repository (**origin**) and the location, i.e. the Gitlab URL. When done, click on **Add**.

In [41]:
```
display.Image('git_gui9.png')
```

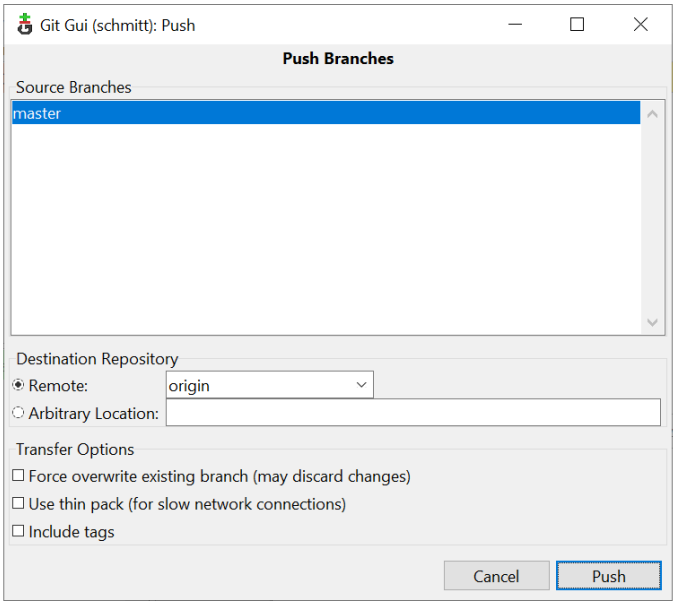
Out[41]:



Finally, click on **Remote** in the menu bar again, and then on **Push..** to push your file. On the next window, click on **Push**. Again, going back to your Gitlab repository and refreshing the page should show your commit.

In [42]:
```
display.Image('git_gui10.png')
```

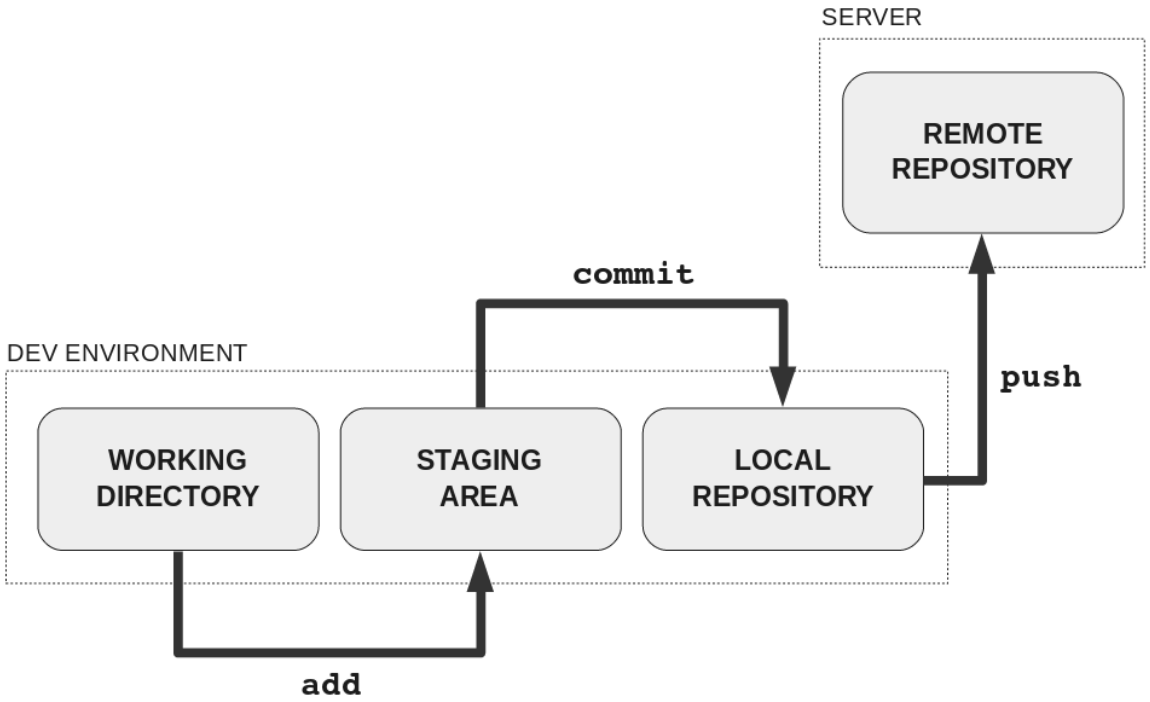
Out[42]:



The following picture summarizes this workflow:

In [43]:
```
display.Image('git_schema2.png', width = 600)
```

Out[43]:



For the problem sets, your workflow will be same as the one just illustrated: you save your solution (a Jupyter notebook) in a subfolder (whose name will relate to the name of the problem set) in your local repository, and then push it to your remote repository on Gitlab.

Since I have access to your remote repository, I can easily pull your solution. I will then use an autograder to mark your solution. Finally, I will push a feedback file (in **html**) back to your remote repository.

Further Topics

Branches

We saw earlier how we can go to previous versions of a file by checking out a commit, i.e. by running **git checkout** plus the commit ID. This is a bit tedious, and usually not how you should change between different versions of a file in practice.

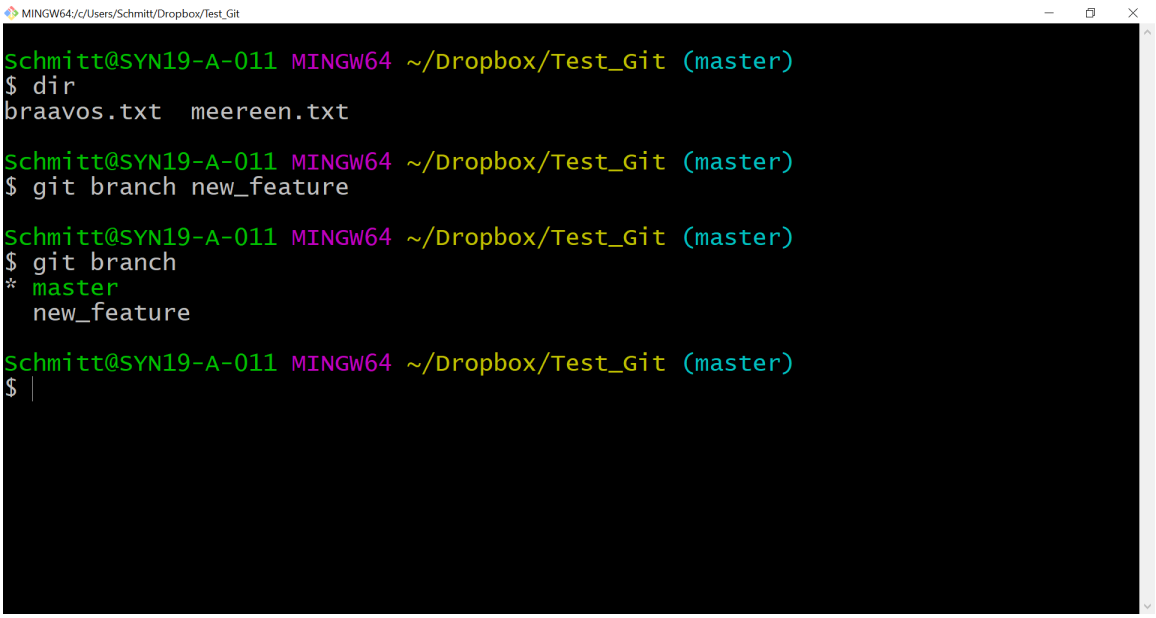
Instead, we work with branches. Say you have a version of a file, e.g. the implementation of a numerical model, that works and has all necessary features. After committing the file, it is by default on the **master** branch of your repository (that's where the word **master** in **git push origin master** comes from).

Now, you want to try out a new feature in the model, but you are not sure if it will work out. Instead of changing the code on the master branch, you can add a new branch, called for example **new_feature**, to the repository by running **git branch new_feature**.

Afterwards, you can type **git branch** to see all existing branches and verify that the new branch has been created.

In [44]:
```
display.Image('git18.png')
```

Out[44]:

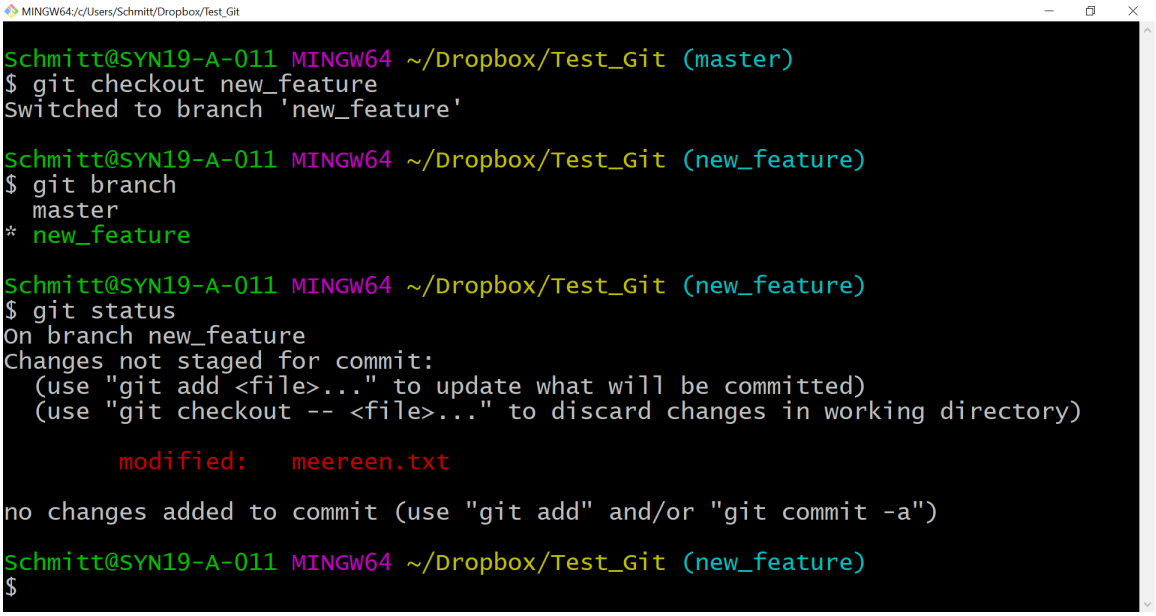


In order to switch to the new branch, type **git checkout new_feature**.

In the example, I then make change to the text file **meereen.txt**. Running **git status** confirms the modification.

In [45]:
```
display.Image('git19.png')
```

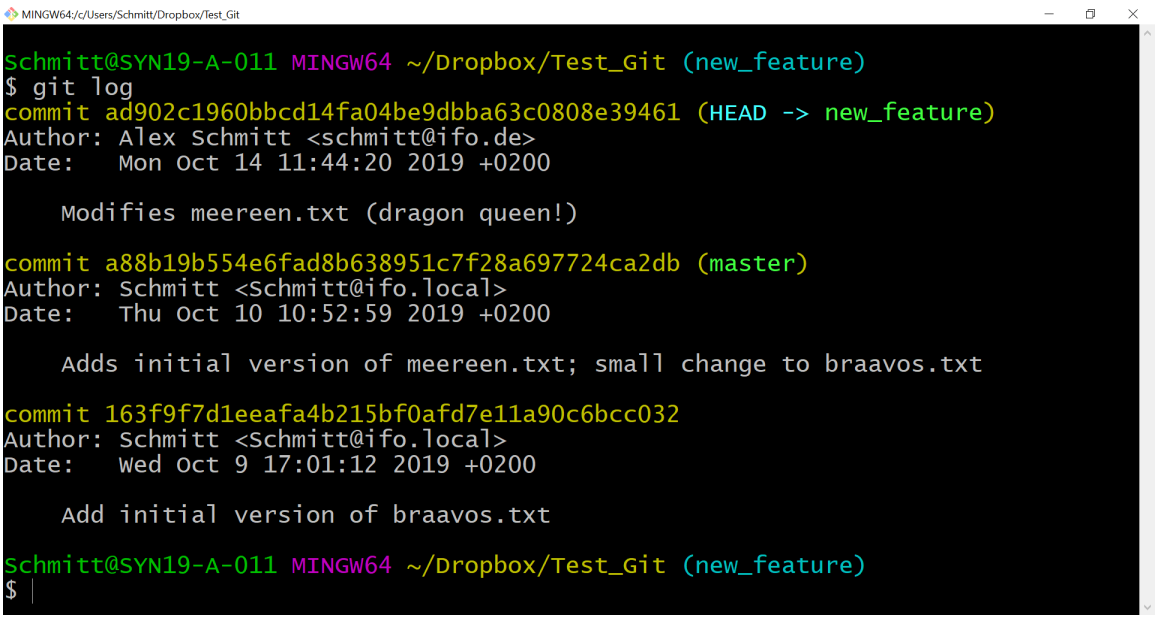
Out[45]:



Next, I add and commit the file as before. We can run **git log** to verify that the commit was successful.

In [46]:
```
display.Image('git20.png')
```

Out[46]:

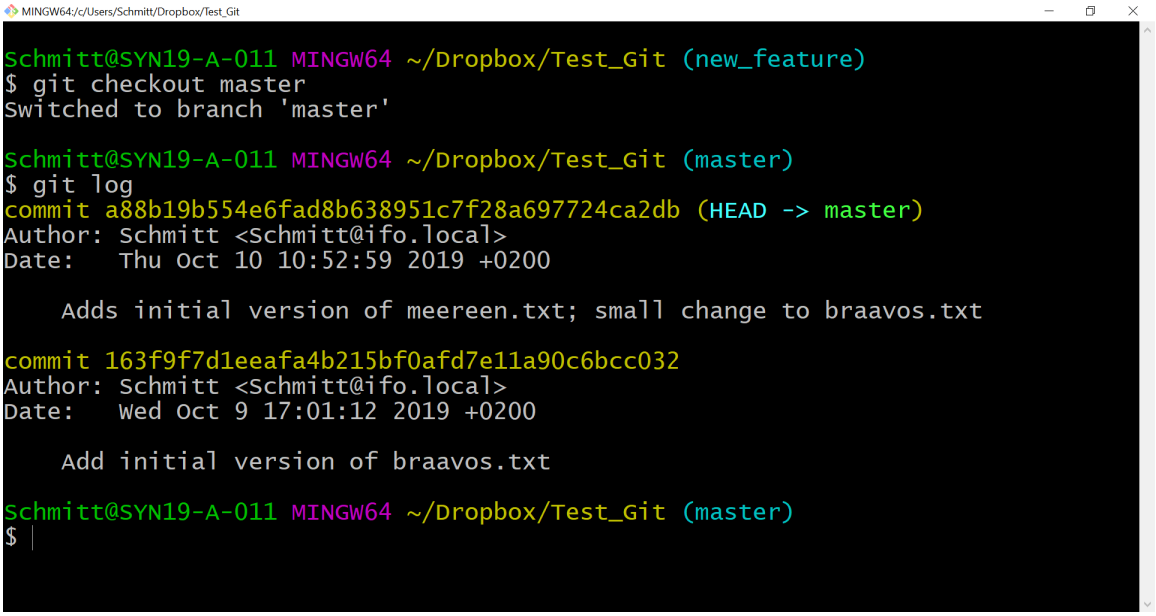


But note that we are on the branch **new feature**; hence, the change to **meereen.txt** has only been committed on this branch, but not on the master branch!

To see this, go back to the master branch by running **git checkout master** and check the commit history: the most recent commit is not there.

In [47]:
```
display.Image('git21.png')
```

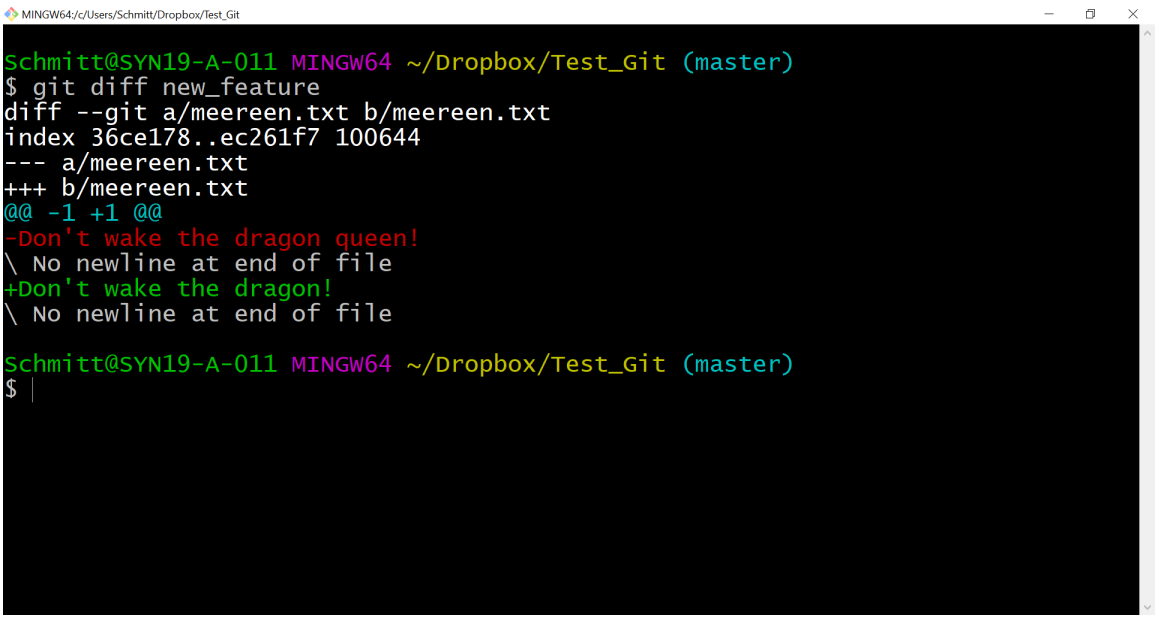
Out[47]:



We can also directly compare the two branches by running **git diff new_feature** (when on the **master branch**). It shows the files that are different between the two branches.

In [48]:
```
display.Image('git22.png')
```

Out[48]:



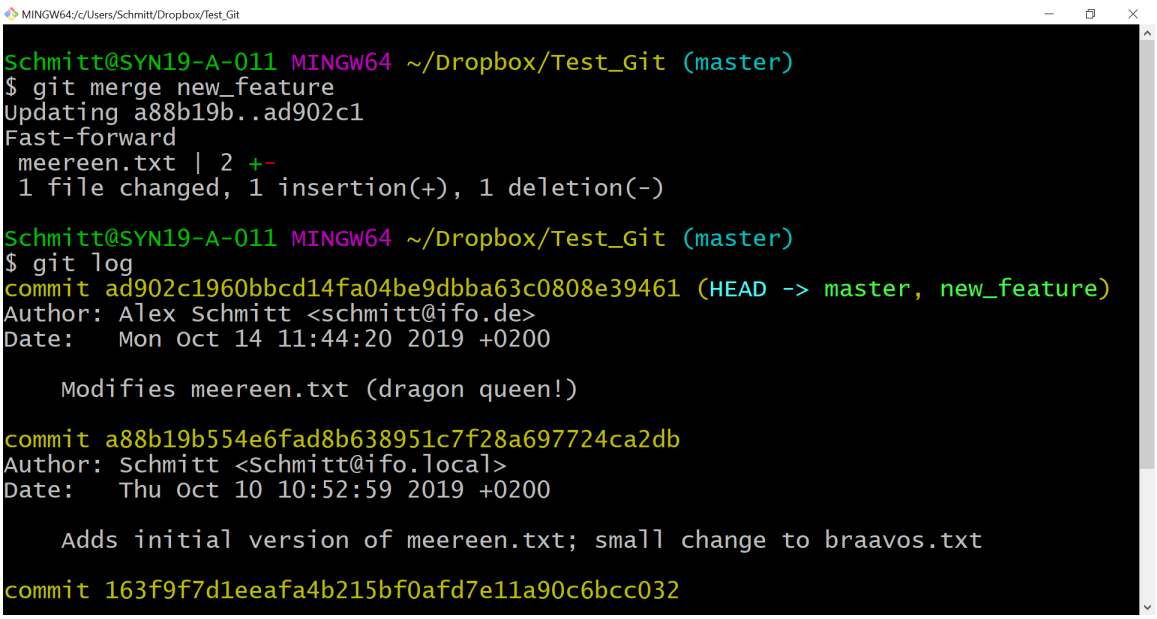
It is important to note that going from one branch to another changes the files in your working directory. In other words, opening the txt-file in the example in your explorer will give you a different version depending on the branch you have currently selected.

Finally, say you are satisfied with the new feature and want to include it in the "main" version of the code; i.e. you want to integrate the **new_feature** branch into the **master** branch. In Git terms, this is called *merging*. In the example above, checkout the **master** branch and run **git merge new_feature**.

Checking the commit history again, we can see that the most recent commit is now part of the **master** branch.

In [49]:
```
display.Image('git23.png')
```

Out[49]:

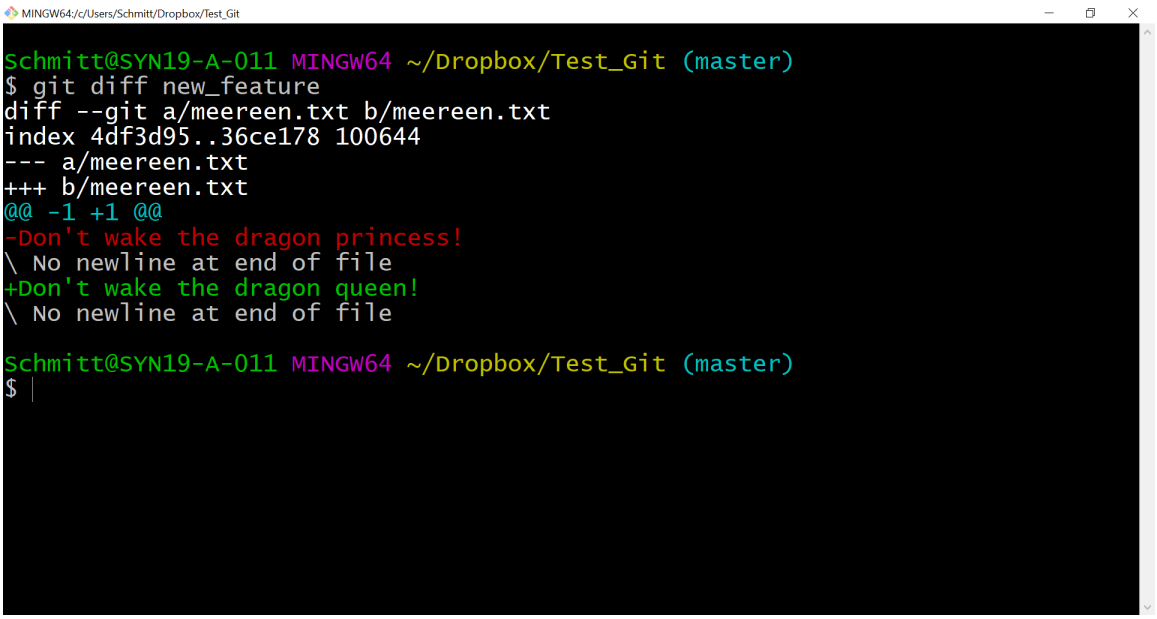


Merge Conflicts

In this very simple example, merging is straightforward, just adding a word to line. This is often not the case, in particular in shared repositories, where different collaborators may work in different branches. Trying to merge them may then lead to *merge conflicts*.

Here is a stylized example. Suppose you have two new branches, both originating from the master branch. The branch **new_feature** includes the following change:

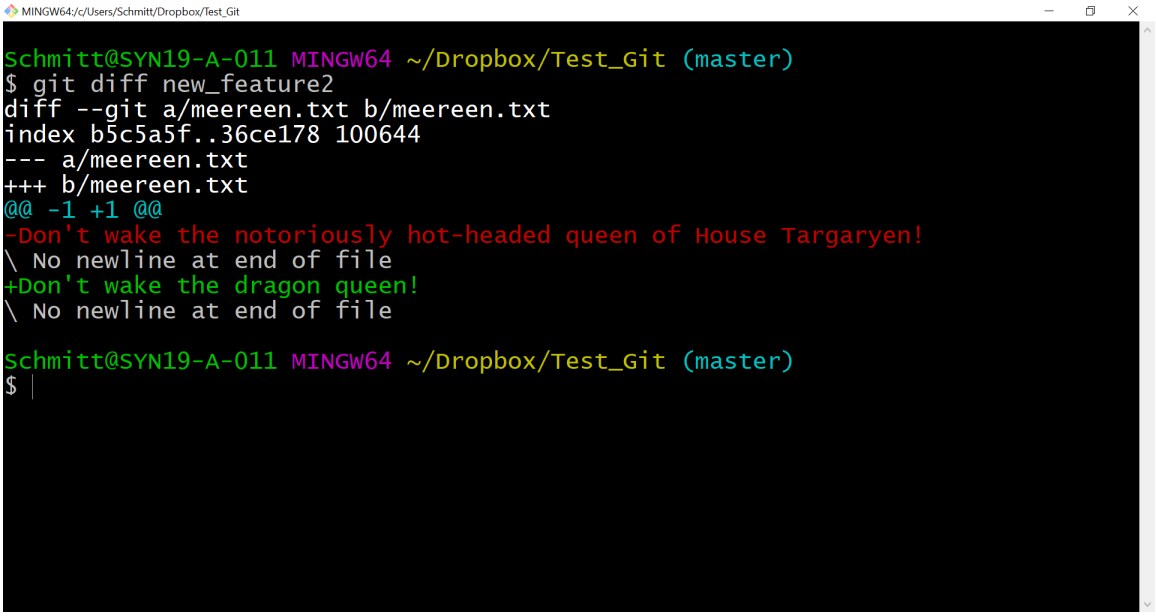
In [50]:
```
display.Image('git25.png')
```

Out[50]:


The branch **new_feature2** looks the following:

In [51]:
```
display.Image('git26.png')
```

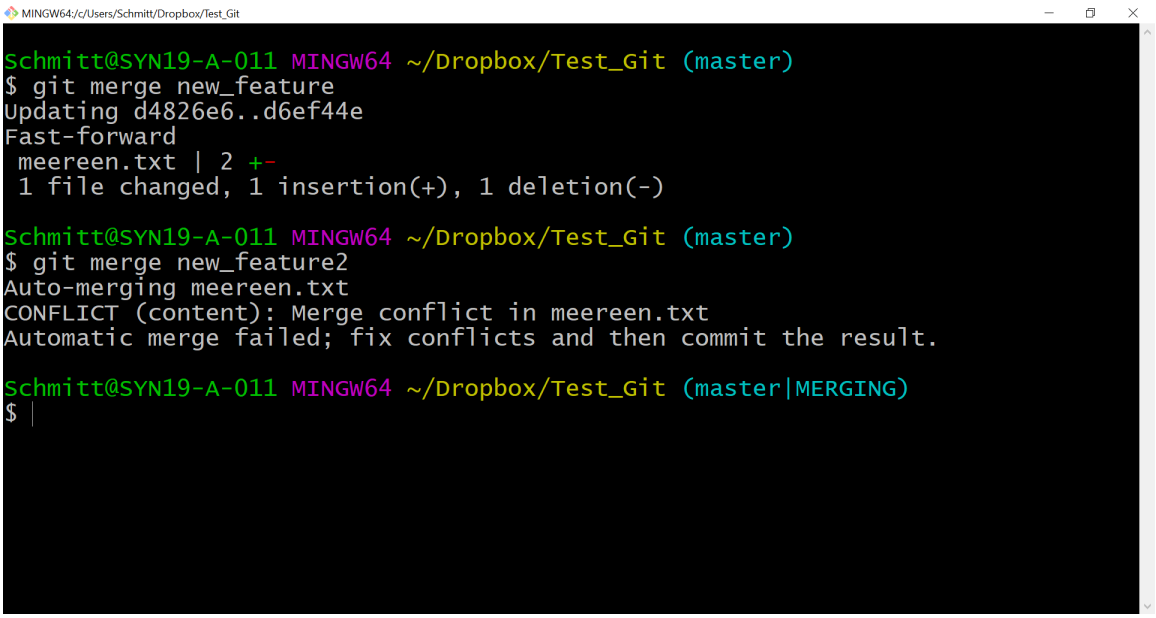
Out[51]:



We first merge **master** with **new_feature**, and then with **new_feature2**. The first merge is straightforward, but the second is not: Git does not know what parts to incorporate into the **master** branch, and hence cannot perform an automatic merge.

In [52]:
```
display.Image('git27.png')
```

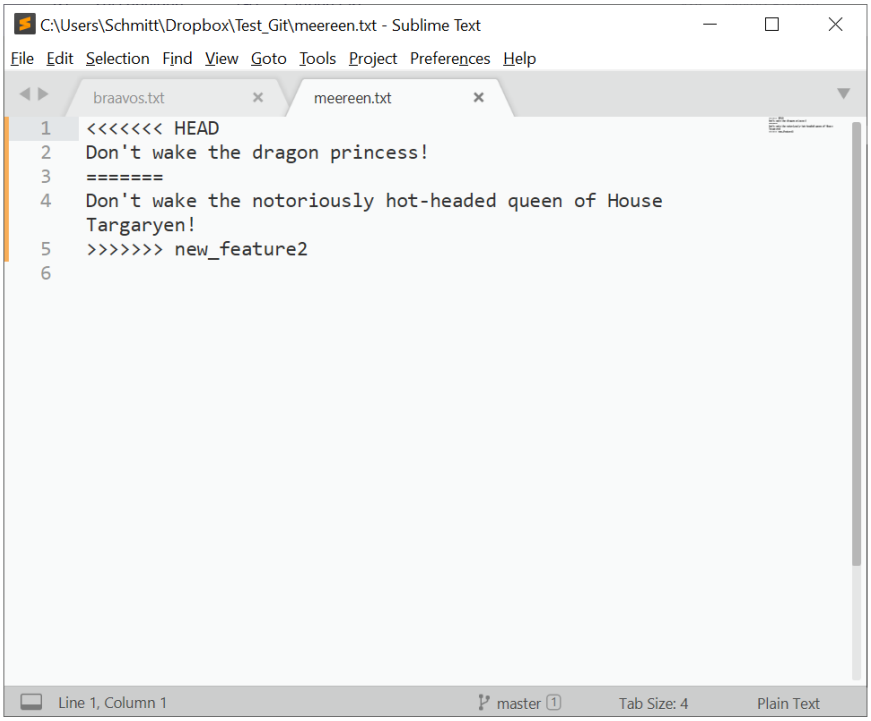
Out[52]:



You have to manually solve this conflict, by opening the file and manually editing the corresponding part(s). Git shows which parts cause a conflict by enclosing them by **<<<<<<<** and **>>>>>>>**.

In [53]:
```
display.Image('git28.png')
```

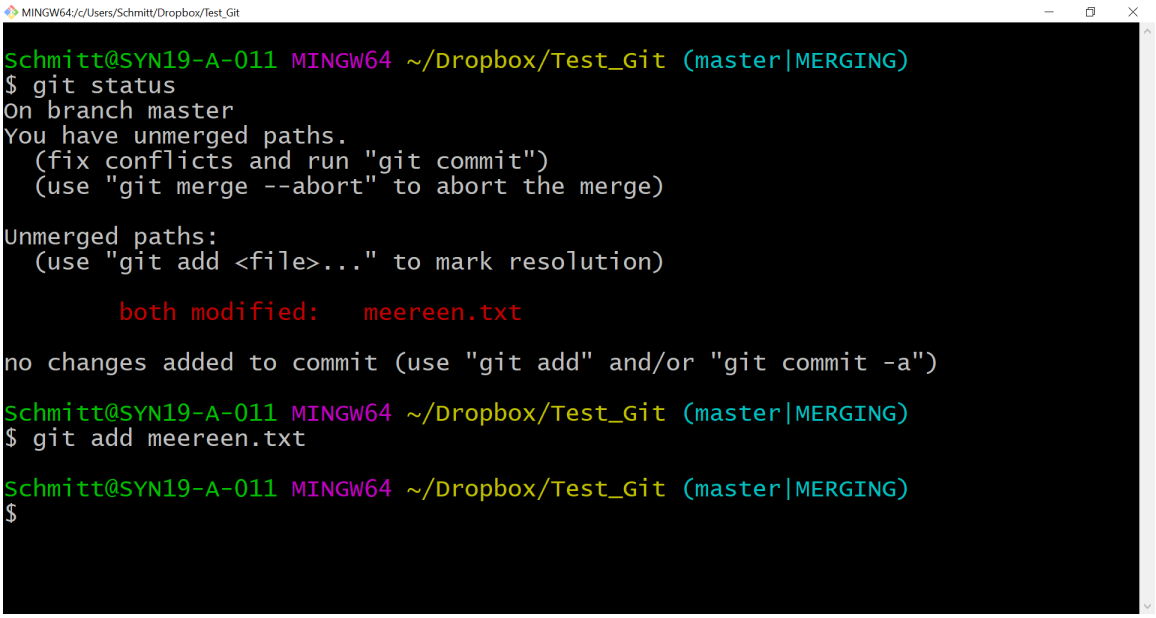
Out[53]:



Once you are happy with the version, you have to add and commit the file, otherwise it will not be tracked by Git.

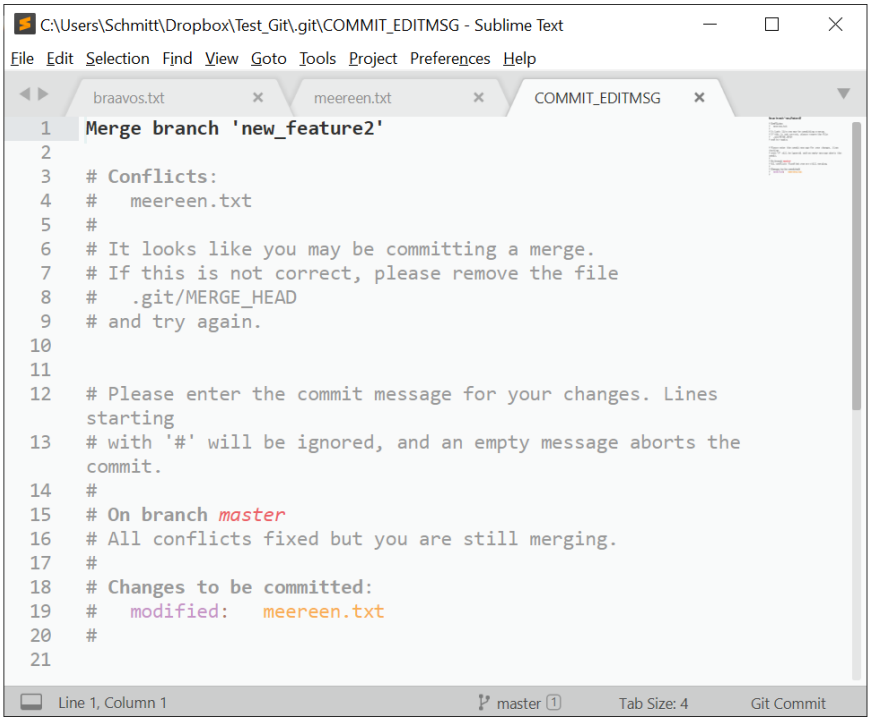
In [54]:
```
display.Image('git29.png')
```

Out[54]:



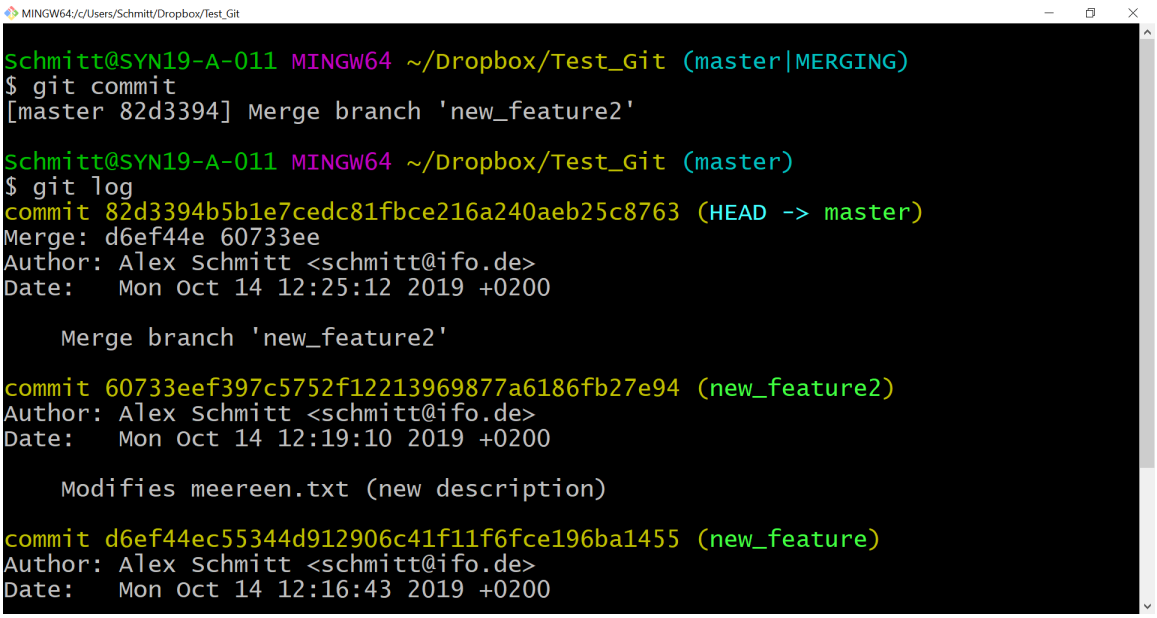
In [75]:
```
display.Image('git30.png')
```

Out[75]:



In [55]:
```
display.Image('git31.png')
```

Out[55]:



As a final remark, there is more to branches and merging than what was presented here. If you want to use these workflows in future work, it is highly recommended to spend some time reading up on them!

Moreover, I have only shown how to work with branches and merges in Git Bash. The conceptss are of course the same for Git GUI, but you'll have to play around a bit to see how they are implemented.

Appendix

Using a Text Editor with Git

If you use Git with the terminal/command line, it is convenient to use **a text editor for writing your commit messages** (rather than the default way in the terminal itself). To set this up, you need the path of your text editor on your machine.

For example, I'm using the text editor "Sublime Text 3", which is for free (at least as an unlimited demo version) and has a number on nice features: https://www.sublimetext.com/

On the Mac, it is located under "Applications". To link it to Git, open a terminal and type the following:

git config --global core.editor "'/Applications/Sublime Text 3.app/Contents/SharedSupport/bin/subl' -n -w"

On Windows, it's the same command, but with a differently named path, e.g.

git config --global core.editor "'C:/Program Files/Sublime Text 2/sublime_text.exe' -n -w"

Note that the precise paths may be different on your machine. If you use a different text editor, please use Google to find the path that you have to use.

You can find more information on how to customize your Git environment in the context of the Udacity course that I have linked above.

Setting up the Mac Terminal for Git

If you want the **Terminal** to display prompts with colors and show whether you have uncommitted changes in working directory, implement the following steps. Note that this is completely optional: you can use Git without this, but it will make your work experience more pleasant.

1. Start a terminal and type **cd ~** to get your home directory.
2. Here, look for the name of your Bash profile. Type **ls -a** to list all (also hidden files). Look for either **.profile** or **.bash-profile**.
3. In the terminal, type **nano** plus the name of your Bash profile. This will open Terminal's innate text editor and show your bash profile.
4. I have provided a file **git-in-bash-profile.txt** in the lecture repository. Copy its contents and paste it into the bash profile. Hit **Ctrl + X** to exit and hit **Y** to save your changes.
5. Finally, there are two more files in the repository, **git-completion.bash** and **git-prompt.sh**. Copy them to your home directory.
6. Exit and restart the terminal.

In [ ]: