H62PEP Computing - Worksheet 2 - Git and Github

P. Evans, K. Cools

Collaborating with Git and GitHub

Exercise: Sign up for the Github student pack (https://education.github.com/pack). It takes some time to process, so better to do it right now. We will go into detail later on.

The problem

Naturally, code changes throughout the development process. Hopefully these changes will result in features being added and bugs being fixed. Unfortunately, despite our best efforts and the techniques for error detection we will discuss in the next term, it is impossible to avoid errors from now and then appearing in the code. This could happen for example when new or modified code fails to fulfil the silent assumptions that other code relies upon. This is just a single example of how bugs can be created without actually touching the relevant code, but many more, and much more mundane reasons exist.

All of this makes creating software a very hard exercise and sometimes there is no better solution than going back in time and inspect the code at the last working version. Even though most modern operating systems offer general and highly automated backup solutions, these solutions are not necessarily the most adequate for the management of source code: they give very little info when it comes to the actual content of the files, they do not allow to quickly retrieve old versions, and they do not store any meta-data that can give further insight in how the code base evolved over time.

Neither do these OS supplied systems provide branching, i.e. the ability to maintain and track several parallel versions of the code base (one for the next release, one for the latest bugfixes, one for an experimental feature, one for a paying customer who desired bespoke modifications,. . .). It goes without saying that an OS supplied backup system is not portable and thus cannot be used within the terms of reference of our project.

A related problem is that of collaboration. Indeed, assuming that you cannot remember all the modifications you made over time (and you cannot remember all modifications you made over time), working with several people is very similar to working with multiple versions of the software. In this case the goal is of course to, on a very regular basis, merge all existing versions into a single working copy that benefits from all the code you and your colleagues have been producing. No backup system I know about provides any help in this regard.

The solution

To solve the above problem we will use a tool called git to keep track of all code you will develop for this project. Git is a so-called code versioning system. Other systems are around (such as svn and mercurial). We will stick with git because it is fair to say it is the most widespread system used today and it offers the highest flexibility.

Exercise: figure out how to install git on your OS. Install git on your OS.

Git manages what are known as repositories. From a user point of view, these are simply directories containing code and their history / different versions. Git itself is a command line tool, but GUI interface tools for the various OSs exist. I strongly advise to use the CLI (command line interface) tool to begin with, because some of these GUI tools are pretty pro-active in guessing what you mean, even if you didn't mean what they guessed.

After installing git somewhere on your system you can invoke it as any other programme. That means that you can either supply the (usually very long) complete path to the executable, or you can add it to the system path. I recommend you persistently put the path to git.exe (on Windows) on the path.

Even if you are working on a Uni PC I would recommend not placing the repos on your Z drive. The reason is that git uses a lot of small files to keep track of history and that the small delay in accessing Z becomes very noticeable under these circumstances. As you will see this is not really a problem. All your work will be saved in the cloud anyway.

Creating a Repository

Exercise: Create a subdirectory that will contain your new repository. This repository will become your individual repository where you can create sub-directories for the worked examples you will be asked to complete in these worksheets.

Create a text file `README.md` using notepad. Place some text in this file, e.g. `hello world`.

We now create a repository by:

```
> cd repository_directory_name
> git init
Initialized empty Git repository in D:/repository_directory_name/.git/
```

The data git uses is stored in the hidden subdirectory .git. Users should not modify this directory. First we need to tell git to track the file we just created. This is done by:

```
>git add README.md
> git status
On branch master
No commits yet
Changes to be committed:
    (use "git rm --cached <file>..." to unstage)
        new file: README.md
```

Here we used the git status command to get a summary of the current state of our development directory. This provides us with some important pieces of information: * The branch we are working on. Roughly said this is the version of the code we are working on. More details later.

- A list of files that are modified but not staged to be committed.
- A list of files that are modified and whose modifications are staged to be committed.

This is something where a lot of novice git users get confused. Modifications first need to be staged to be committed. In the following commit statement, all changes that where staged up to that point will be the subject of the next commit.

The reason the system was designed this way is to provide the user with the opportunity to split a set of modifications done since the last commit over multiple new commits.

Let's now actually commit the change:

```
>git commit -m "Initial commit"
[master (root-commit) 7267e00] Initial commit
    1 file changed, 1 insertion(+)
    create mode 100644 README.md
> git status
On branch master
nothing to commit, working tree clean
```

A commit needs an obligatory commit message. It is often challenging to enter something meaningful here, especially if you do a lot of commits (and you should do a lot of commits). Still take some time to make sure the message is representative of what you where working on at the time.

Immediately after doing a commit, the status shows that no modifications are present in the current working tree, as expected. Let's follow this initial commit with another one.

Exercise: create a new file hi_and_date.cmd with a command line script that prints a welcome message and the current time and date (see the data command). Change the test in README.md to "Read this very carefully". Add and commit the modifications.

Hint: The git add command followed by a directory name add all modified files in that directory and all its subdirectories to the stage. In particular git add . adds all modifications.

After finishing this we can query our two commit history:

```
> git log
commit 87746500427a636aa7b974697e205f465607c419 (HEAD -> master)
Author: Kristof Cools <krcools@gmail.com>
Date: Sun Aug 27 19:16:18 2017 +0100

    hi_and_date script added

    commit 7267e007958fbdb87013ecfba319ab505ae884ca

    Author: Kristof Cools <krcools@gmail.com>
Date: Sun Aug 27 18:53:40 2017 +0100
Initial commit
```

We can retrieve the modifications that were applied at every commit using the show command. The argument for the show command is the commit you want to visit. All commits have a hash code or SHA. This is the long hexa-decimal string you find next to commit in the output of the log command. Of course this is not a practical way of identifying the commit. Fortunately it suffices to enter the first couple of digits making up the SHA. As long as this uniquely determines the commit no further specification is required.

```
$ git show 87746
commit 87746500427a636aa7b974697e205f465607c419 (HEAD -> master)
Author: Kristof Cools <krcools@gmail.com>
Date: Sun Aug 27 19:16:18 2017 +0100

    hi_and_date script added

diff --git a/README.md b/README.md
index 1eeb140..baec71d 100644
--- a/README.md
+++ b/README.md
@@ -1 +1 @@
-Hello, World!
+Read this very carefully!
diff --git a/hi_and_date.cmd b/hi_and_date.cmd
new file mode 100644
index 0000000..4a7aab9
--- /dev/null
+++ b/hi_and_date.cmd
@@ -0,0 +1,3 @@
+@echo off
+date
+echo Welcome!
\ No newline at end of file
```

As you can tell from this output only the modifications are reported, not the entire file content. The show command allows deep customisation, but by default you get the commit in so called patch format. The + and - signs in front of the lines indicate which lines are removed and which ones are added. A modified line simply is recorded as a combination of a line removed and a line added.

There are a number of short hands to indicate commits, called references or refs for short. In particular, HEAD always refers to the most recent commit on the current branch. Entering a new commit will automatically advance the HEAD reference to point to the correct location. The most recent commit on any branch can be referred to using the branch name. In our current one branch scenario the following commands are synonymous (since 87747 is the ID of the previous commit, HEAD also points to the most recent commit, and master is the name of the default branch which is where the most recent commit was made):

```
>git show 87746
>git show HEAD
>git show master
```

The commit HEAD is pointing to just prior to submitting a new commit will be the parent of that commit. In general to refer to the parent of a commit, you can use the tilde notation. This means the following are synonymous:

```
>git show HEAD~
>git show HEAD~1
>git show master~1
>git show 7267e00
```

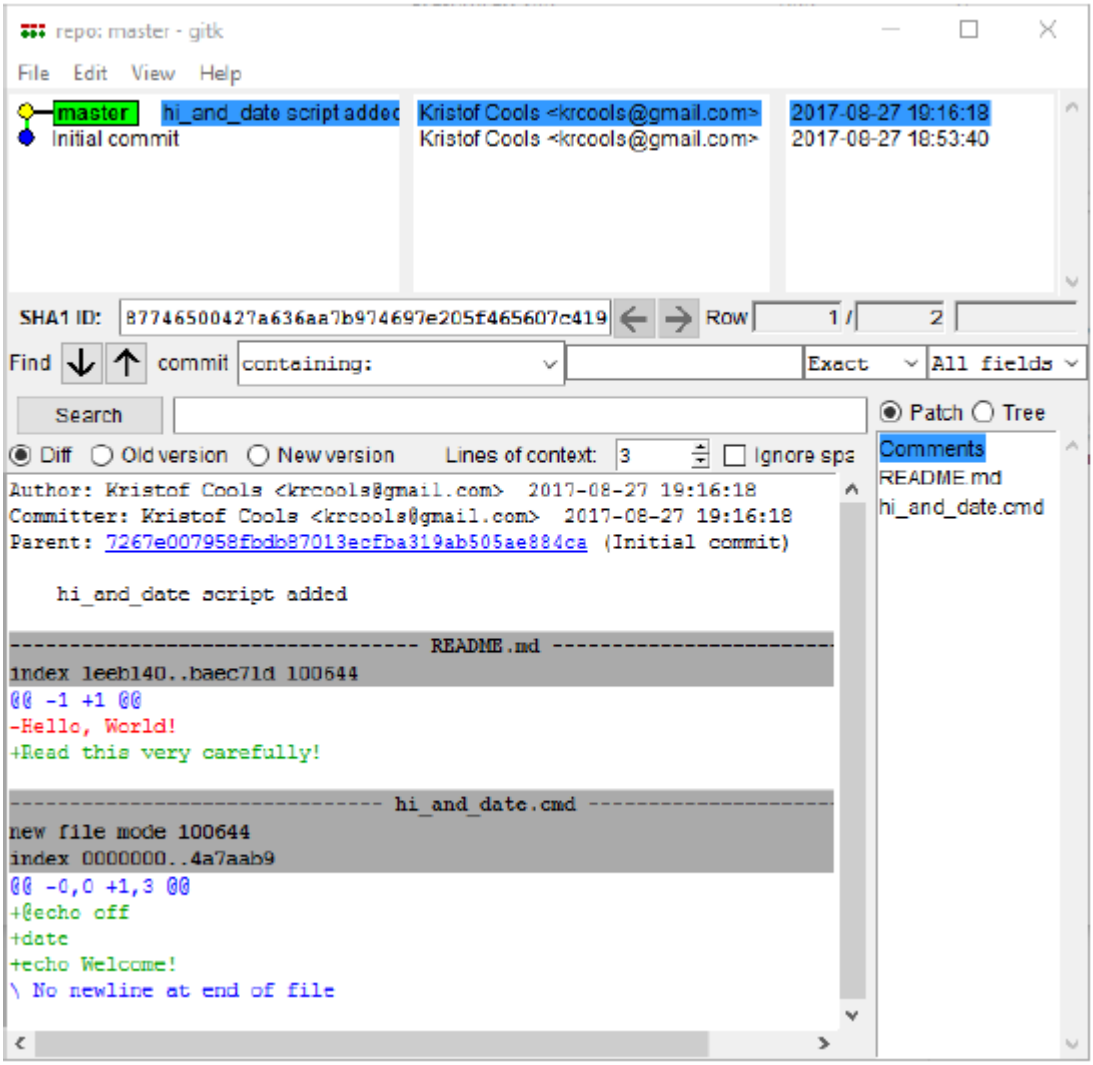
Git for Windows comes with a number of minimalistic but complete visual interfaces for adding, committing, and inspection of the commit history. You can launch a GUI window that enables you add a commit with:

```
>git gui
```

And view the commit history of a branch with:

```
>gitk master
```

If you want something more flashy, take a look at e.g. GitKraken. This tools offers in addition to the functionality of gitk also tools for working with remote repositories.



Exercise: Add a new script to the repo: last-commit.cmd. The content will be a single line git command that print in a single line the most important info for the last commit. Read the documentation of git show online https://git-scm.com/docs/git-show. The script should have the following effect:

```
>last-commit
    87746500427a636aa7b974697e205f465607c419 (HEAD -> master) hi_and_date script added
```

Note: When you've worked out the correct command to use you'll need to add it to a batch file. Any '%' characters in the command will need to be doubled-up (%%) otherwise the batch file will interpret them as variables!

Going back in time

In this section we will discuss four ways you can go back in time using git:

- Undoing a staging operation using git reset
- Disregarding changes in the working tree using checkout
- Checking out a previous version using checkout
- Undoing a commit using reset

Git is extremely conservative when it comes to overwriting you work. Most command will generate new commits or add to the history of your code development without actually overwriting data. From the four options above, only the second can lead to loss of data.

Git may seem very user unfriendly. It is true that the names of some commands do not really seem to make a lot

of sense or are used for several task that at first sight do not seem very much related at all. Only when you learn more about the philosphy behind the design of git you will see that there is actually a lot of consistency.

Scenario 1: Undoing a stage operation

At mentioned above, the whole reason there is a staging area is to allow you to submit your changes in multiple commits. If you accidentally staged more changes than you originally were planning to commit you can undo these stage operations on a per file level using the following operation:

```
>git status
    On branch master

    Changes to be committed:
        (use "git reset HEAD <file>..." to unstage)
        modified: README.md

>git reset HEAD README.md
    Unstaged changes after reset:
    M README.md

>git status
    On branch master
    Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)
    modified: README.md
    no changes added to commit (use "git add" and/or "git commit -a")
```

Note that git actually tells you what this command is as part of the output of git status.

Exercise: Make changes to README.md. Add (but don't commit) the file and undo the staging using the reset command.

At this point you remove the changes from the staging area (incidentally, the staging area is sometimes also called the index). The changes you made however are still present in your working directory, as the output of git status will remind you of.

Scenario 2: Discarding changes made in the working tree

Discarding changes made in the working tree are undone by overwriting files with versions that were previously stored in the history managed by git. You can check out these versions using the - wait for it - `checkout` command.

```
>git checkout -- README.md
>git status
    On branch master
    nothing to commit, working tree clean
```

This is one of a few instances where git overwrites your work. Use git checkout in conjunction with file names very carefully! Gone is Gone!

Run this command and check to see that the last change you made to Readme.md has been undone.

Scenario 3: Checking out a previous version using checkout

Related to the previous but more common (and safe) is to checkout a different version of your code base. This does not delete anything from the history. It simply places an older version in your working directory (HEAD then points to this previous version).

```
>REM Checkout the version prior to the last commit

>git checkout HEAD~1
    Note: checking out 'HEAD~1'.
    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 performing another checkout.
    If you want to create a new branch to retain commits you create, you may
    do so (now or later) by using -b with the checkout command again. Example:
    git checkout -b <new-branch-name>

    HEAD is now at 8774650... hi_and_date script added

>git log
    commit 87746500427a636aa7b974697e205f465607c419 (HEAD)
    Author: Kristof Cools <krcools@gmail.com>
    Date: Sun Aug 27 19:16:18 2017 +0100

        hi_and_date script added

    commit 7267e007958fbdb87013ecfba319ab505ae884ca
    Author: Kristof Cools <krcools@gmail.com>
    Date: Sun Aug 27 18:53:40 2017 +0100

        Initial commit

>git checkout master
    Previous HEAD position was 8774650... hi_and_date script added
    Switched to branch 'master'

>git log
    commit e4de40830a89777e8c3f2cc1b5a64ce176d62b6f (HEAD -> master)
    Author: Kristof Cools <krcools@gmail.com>
    Date: Tue Aug 29 11:54:34 2017 +0100

        last-commit shows oneline summary for HEAD

    commit 87746500427a636aa7b974697e205f465607c419
    Author: Kristof Cools <krcools@gmail.com>
    Date: Sun Aug 27 19:16:18 2017 +0100

        hi_and_date script added

    commit 7267e007958fbdb87013ecfba319ab505ae884ca
    Author: Kristof Cools <krcools@gmail.com>
    Date: Sun Aug 27 18:53:40 2017 +0100

        Initial commit
```

`git log` will only give you information about the commit history up to the commit you currently have checked out (i.e. the commit pointed to by HEAD). In particular this means that our history seems to contain only two commits, not three.

`git checkout master` resets HEAD so that it points to the most recent commit again.

Scenario 4: Undoing a commit using reset

The git reset command changes where HEAD and the current branch name (e.g. master) refer to. By resetting it to refer to an older commit, you effectively forget about the newer commit. Note that it does not get deleted from disk, it just disappeared from git's history books (#fakenews).

```
> git reset master~1

> git log
    commit 87746500427a636aa7b974697e205f465607c419 (HEAD -> master)
    Author: Kristof Cools <krcools@gmail.com>
    Date: Sun Aug 27 19:16:18 2017 +0100

        hi_and_date script added

    commit 7267e007958fbdb87013ecfba319ab505ae884ca
    Author: Kristof Cools <krcools@gmail.com>
    Date: Sun Aug 27 18:53:40 2017 +0100

        Initial commit
```

This will have HEAD and the branch it points at (master) point at the commit prior to master. Even though this technically does not remove the latest commit, it is a bit tricky to undo the reset operation. Immediately after the reset command it can be undone by:

```
> git reset HEAD@{1}
```

You should read this as: set HEAD (and the branch it points at) to wherever HEAD was pointing at before the last time it moved.

Creating a Branch

A branch in git should be though of as a separate version. It is implemented as a reference to a commit. Being on a branch simply means that every time you commit changes, the branch moves to the new commit, i.e. it always points at the most recent work done in that version of the code. Branches are used to separate work done on features or bug fixes, or to enforce a code freeze for an upcoming release, whilst allowing development to continue on the development branch.

Usually we want to immediately start working on a newly created branch and so the checkout command allows also for the creation of branches using the -b flag:

```
> git checkout -b coolstuff
Switched to a new branch 'coolstuff'

ezzkc@luin63348 MINGW64 /d/Users/ezzkc/OneDrive/Temp/repo (coolstuff)
> git branch -v
* coolstuff e4de408 last-commit shows oneline summary for HEAD
  master e4de408 last-commit shows oneline summary for HEAD
```

You can read off a number of things from this output: * Two branches exist and we are on the one marked by an asterisk * Both branches point (for now) to the same commit

Exercise: Add a file called `git-cheat-sheet.txt` summarising in your own words what each of git show log checkout init reset do. Add and commit it to the current branch (`coolstuff`).

Merging Branches

Say you are extremely happy with how your git-cheat-sheet.txt is getting along. In that case, you will probably want to include it in the main developing branch master so that it will be part of your next release. At this point

the history looks like this:

```
> git log --graph --oneline
* af4eee8 (HEAD -> coolstuff) git-cheat-sheet contains TMI
* e4de408 (master) last-commit shows oneline summary for HEAD
* 8774650 hi_and_date script added
* 7267e00 Initial commit
```

Note that `coolstuff` and `master` (remember that branches can be though of as references to commits) are pointing to different commits. This makes sense: we did some work on `coolstuff` that is not included in `master`. In order to merge it into master we first checkout that branch and than use the merge command:

```
> git checkout master
Switched to branch 'master'
```

Notice that after you execute the `checkout master` command the cheatsheet document disappears because it only exists on the coolstuff branch.

```
> git merge coolstuff
  Updating e4de408..af4eee8
  Fast-forward
      git-cheat-sheet.txt | 3 +++
      1 file changed, 3 insertions(+)
      create mode 100644 git-cheat-sheet.txt

  > git log --graph --oneline
      * af4eee8 (HEAD -> master, coolstuff) git-cheat-sheet contains TMI
      * e4de408 last-commit shows oneline summary for HEAD
      * 8774650 hi_and_date script added
      * 7267e00 Initial commit
```

Now master and coolstuff are pointing at the same commit. Not only does this mean the share the same content, but they also have the same history. Even though merge seems to imply some very smart algortihm was employed to commensurate the two version, all that really happened was that the master reference was moved to point at the same commit as the coolstuff reference. Hence the name fast-forward merge as reported by git.

Exercise: Switch to coolstuff. Add a line saying *Priority: cool stuff* to `README.md` Switch back to `master` and and a line saying *Priority: code stability* to `README.md`. Inspect the log (provide the `--all` flag to get the history of all branches in one view) as above, does it look more complicated? Switch to `master` and merge `coolstuff` into the master branch. What happens?

Resolving Conflicts

The log command you used just now show a splitting of the history. This is a result of work being done on the two branches after they were created and before we merged them. Merging the branches by fast-forwarding is not an option in this case. Instead git uses a more advanced algorithm to try to combine the work done in both branches. When changes were made in the exact same location, this algorithm gives up and asks for our help. From our point of view, a conflict has emerged.

If you open a file that has a conflict in it, you see conflict markers like below:

```
>git merge coolstuff
    Auto-merging README.md
    CONFLICT (content): Merge conflict in README.md
    Automatic merge failed; fix conflicts and then commit the result.

    Read this very carefully!
    <<<<<<< HEAD
    Priority: code stability
```

```
=======
Priority: cool stuff!
>>>>>>> coolstuff
```

You get the two versions encountered in the two branches included in the same file and enclosed by conflict markers. It is your job to choose how to proceed: keep one of the version (or merge them yourself in a way you see fit), and remove the conflict markers.

To indicate to git that the conflict is resolved, add the file to the index and submit it to the history in a so called merge commit.

```
>git add .

>git commit -m "A balance between cool and stable was struck"

>git log --graph --all --oneline
    * 71a4726 (HEAD -> master) a balance between cool and stable was struck
    |\
    | * 8defc0c (coolstuff) stress importance of cool stuff
    * | 78c7d68 stress importance of stability
    |/
    * af4eee8 git-cheat-sheet contains TMI
    * e4de408 last-commit shows oneline summary for HEAD
    * 8774650 hi_and_date script added
    * 7267e00 Initial commit
```

And history is unified again! Being in conflict with yourself is fairly rare, but can be challenging nevertheless. Conflicts with other are more common as you often do not know which part of the code they are working on.

The .gitignore file

Git is well suited to keep track of source code and other simple text format based content. The things it is not good at are:

- Dealing with content where a simple semantic modification leads to file-wide changes in the representation of this information. E.g. don't try to have git manage your word or excell files.
- Large files. Git is a distributed versioning system. This means everyone needs to have a complete history. If large files are part of this history, this causes a lot of network traffic and corresponding delays.

This poses no serious restrictions on software development. In fact, when developing portable projects, you typically want to keep the entire shared code base free of any binary or system specific file types.

Keeping these system specific files out of reach of git is not always a simple task. For example while building the software (i.e. compiling, linking, . . .), many build systems pollute the code base with project files, and intermediate/final binaries.

Git has a simple but effective way to exclude these files. We can provide a file called .gitignore listing all types of files and names of subdirectories that should be exluded from version management. An example of such a file is:

```
# Contents of .gitignore
*.com
*.class
*.dll
*.exe
*.o
*.so
build/
```

Make sure you have a suitable .gitignore file(s) in your repositories. You should only be storing code on github, not binary files / executables.

Closing remarks on Git

This is a lot to take in. Git was designed to manage extremely large projects. Actually it was designed for the Linux kernel. You will learn git gradually. My advice is to create a repo just for messing around where you can experiment with what each command does.

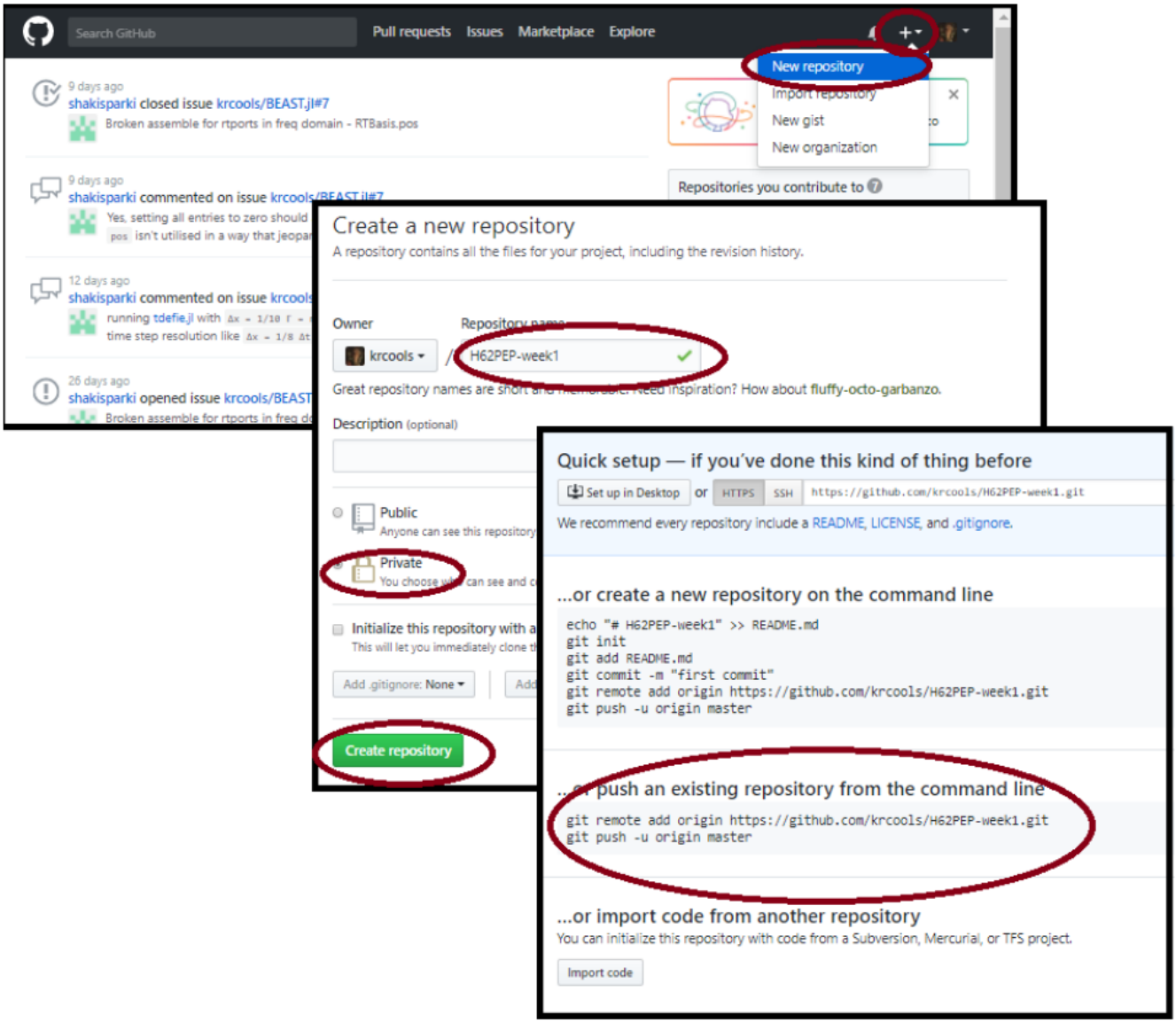
Remote Repositories and Github

To work together with others it is important that there is a central repository that is kept up to date and that is accessible to everyone on the team. In it most primitive form you could set up a shared drive and simply put the repository there.

Github is a free service (for open source projects and for academic use) that provides a place to put your communal repository but on top also offers many useful features:

- Issue trackers
- A Project website
- An in-browser visualisation of your code-base
- Pull request support
- Permission management
- An API that other services such as continuous integration tools can hook into

Your application for the student pack should ave come through by now. Log into Github and click the plus button to generate your first online repository.



As suggested by the dialog, we can connect our local repository to the Github repository by defining it as a so called remote:

```
>git remote add origin https://github.com/username/reponame.git

>git push -u origin master
    Counting objects: 22, done.
    Delta compression using up to 8 threads.
    Compressing objects: 100% (16/16), done.
    Writing objects: 100% (22/22), 2.21 KiB | 755.00 KiB/s, done.
    Total 22 (delta 3), reused 0 (delta 0)
    remote: Resolving deltas: 100% (3/3), done.
    To https://github.com/username/reponame.git
    * [new branch] master -> master
    Branch master set up to track remote branch master from origin.
```

It is possible you will be asked for your Github credentials. The name `origin` is arbitrary but it is common practice to use this name for the *upstream* repository. Upstream in this context means that you consider it the repository that all your work needs to end up in.

The `git push` command is used to actually copy over your commits so far to Github. The flag `-u` is only required

the first time and help git remember this is the upstream repository for the current branch. Later invocations of push do not require you to mention origin; it is understood that a push executed whilst on the master branch is to copy your data to `origin`.

Exercise: Let's pretend we are another user now (even though you will work under the same Github credentials). Make a directory otheruser and execute the following command from inside that directory:

```
>git clone https://github.com/username/reponame.git
```

You will see that the entire history of the repository is cloned or copied to this new directory.

Exercise: Verify using git log that indeed you have the entire history.

Exercise: Ask for a list of remote repositories by issuing git remote. Note that the repo you pulled from is automatically included, named origin, and set as the upstream destination for the master branch.

Exercise: Let's pretend to be user 1 again. Move into the user 1 repo. Make some changes to README.md (for example "User 1 says hello!"). Add, commit, and push to the Github based repository (a simple `git push` whilst on `master` sufffices). Move to the repo of user 2. Pull down the new work added by user 1 by issuing git fetch origin.

The last command, `git fetch` will fetch all new commits from origin and submit that at the commit pointed to by the branch `origin/master`. This leads to two diverged branches: `master` and `origin master`:

```
* 3fb6cfb (origin/master, origin/HEAD) User 1 says hello
* 71a4726 (HEAD -> master) a balance between cool and stable was struck
|\
| * 8defc0c stress importance of cool stuff
* | 78c7d68 stress importance of stability
|/
* af4eee8 git-cheat-sheet contains TMI
* e4de408 last-commit shows oneline summary for HEAD
* 8774650 hi_and_date script added
* 7267e00 Initial commit
```

Fortunately we are in the simple situation where the work on origin/master can be merged into master by a simple fast forward. Fetching and merging with a remote upstream to a local branch is so common that there is a special command for it:

```
$ git pull
    Updating 71a4726..3fb6cfb
    Fast-forward
    README.md | 2 ++
    1 file changed, 2 insertions(+)
    ezzkc@luin63348 MINGW64 /d/Users/ezzkc/OneDrive/Temp/otheruser/H62PEP-week1 (master)

>git log --all --graph --oneline
    * 3fb6cfb (HEAD -> master, origin/master, origin/HEAD) User 1 says hello
    * 71a4726 a balance between cool and stable was struck
    |\
    | * 8defc0c stress importance of cool stuff
    * | 78c7d68 stress importance of stability
    |/
    * af4eee8 git-cheat-sheet contains TMI
    * e4de408 last-commit shows oneline summary for HEAD
    * 8774650 hi_and_date script added
    * 7267e00 Initial commit
```

Just like with a normal merge, this could potentially lead to conflicts. Investigate this in the following exercise:

Exercise: (Add,Commit,Push) as User1 a line in README.md saying "I want it this way". (Add,Commit) as User2 a line in README.md saying "I want it that way". Try pushing. Does this work? Why not? Follow the

instructions git gives you to resolve the situation.

Your history should look something like this now:

```
$ git log --all --graph --oneline
    * b450297 (HEAD -> master, origin/master, origin/HEAD) We found middle ground
    |\
    | * 1cac67e User 1 wants it this way
    * | 31bddb2 User 2 wants it that way
    |/
    * 3fb6cfb User 1 says hello
    * 71a4726 a balance between cool and stable was struck
    |\
    | * 8defc0c stress importance of cool stuff
    * | 78c7d68 stress importance of stability
    |/
    * af4eee8 git-cheat-sheet contains TMI
    * e4de408 last-commit shows oneline summary for HEAD
    * 8774650 hi_and_date script added
    * 7267e00 Initial commit
```

The fork, pull, push, pull request model

Choose a team leader. Team leader, create a new, empty online repo. This will be used to gold your group work for this semester. Your group will develop a model loading library in this repo so give it a sensible name like ModelLoader.

Exercise:

- Team leader, go to the Collaborators page (in settings) of your repo and add your minions to the list of users with read access.
- Minions, surf to the website of your leader's repo and it. What this does is cloning the repo to your Github account.
- Minions, clone your fork to a local directory. Add a file named .txt and add some content to it. Add, commit, and push. This push will be to your own fork, so no conflicts are possible.
- Minions, go to the github page of your own fork and click the New Pull Request button. Check the settings and submit the pull request.
- Leader, navigate to the **Pull Requests** tab on your github repo's site. Check that you are happy with the changes you are asked to pull in and confirm.

Note that from a minion's point of view, there are three repos involved: the github repo of the leader (upstream), the local repo, and the github repo created by forking the project (origin).

A pull request is essentially an automatic email send to the leader to perform a pull from one of the minions. Just like any merge operation, this could result in conflict if non-compatible work has been done in the origin.

It is typically considered the contributor (let's use that instead of minion) responsibility to resolve these conflicts. This is done easiest in the following manner:

- A minion (I changed my mind, fewer characters) defines the leader's repo as a remote, usually called upstream.
- Upon conflicts a minion pulls from origin. This results in conflicts, which can be resolved locally. A merge commit is pushed.
- The pull request is automatically updated and should now notify the absence of conflicts
- The leader can now confirm the pull request, which will be performed by simple fast forwarding

Exercise: Populate the root directory of the group repository with a `README.md`, `NEWS.md`, and `TEAM.md`. Give an overview of the project in README.md. Define your roles in TEAM.md. Get familiar with the pull request dynamics. Add `plevans` to the collaborator list for all of your repositories - remember I will need to see them to be able to mark them!