Workshop on using Git and GitHub

Pieter Barendrecht

KAUST, November 12, 2020

An overview of Git

- Allows you to *track changes* in selected files by taking *snapshots* on demand

An overview of Git

- Allows you to *track changes* in selected files by taking *snapshots* on demand
- The directory containing the files is referred to as *the working directory*. The snapshots are collectively referred to as *the repository (repo)*

An overview of Git

- Allows you to *track changes* in selected files by taking *snapshots* on demand
- The directory containing the files is referred to as *the working directory*. The snapshots are collectively referred to as *the repository (repo)*
- A repository can be shared (online, external HDD, ...) to facilitate *collaboration* and/or to *back-up* your data

An overview of Git

- Allows you to *track changes* in selected files by taking *snapshots* on demand
- The directory containing the files is referred to as *the working directory*. The snapshots are collectively referred to as *the repository (repo)*
- A repository can be shared (online, external HDD, ...) to facilitate *collaboration* and/or to *back-up* your data
- Mainly used for source code (C++, Python, ...), but works well for most *plain-text file formats* (e.g. `.tex`, `.svg`)

Basic usage of Git

- Initialise a repository in an *existing directory* using `git init`
  - Or create a *new directory* and initialise a repository in it using `git init [directory]`

Basic usage of Git

- Initialise a repository in an *existing directory* using `git init`
  - Or create a *new directory* and initialise a repository in it using `git init [directory]`

- Prepare for a snapshot by adding selected files to *the staging area* using `git add`. From now on these files will be tracked
  - Examples: `git add script.py`, `git add *.cpp`, `git add .`

Basic usage of Git

- Initialise a repository in an *existing directory* using `git init`
  - Or create a *new directory* and initialise a repository in it using `git init [directory]`
- Prepare for a snapshot by adding selected files to *the staging area* using `git add`. From now on these files will be tracked
  - Examples: `git add script.py`, `git add *.cpp`, `git add .`
- Take a snapshot using `git commit`
  - Include a *meaningful* description of your updates/changes with every commit — use `git commit -m "message"` or a text editor, which defaults to `vi` (warning, non-standard interface)

Basic usage of Git

- Initialise a repository in an *existing directory* using `git init`
  - Or create a *new directory* and initialise a repository in it using `git init [directory]`
- Prepare for a snapshot by adding selected files to *the staging area* using `git add`. From now on these files will be tracked
  - Examples: `git add script.py`, `git add *.cpp`, `git add .`
- Take a snapshot using `git commit`
  - Include a *meaningful* description of your updates/changes with every commit — use `git commit -m "message"` or a text editor, which defaults to `vi` (warning, non-standard interface)
- Check the current status of your repository using `git status`

The interior of Git (somewhat technical, but essential!)

- Using `git add` stores each selected file as a *blob* (binary large object) in the directory `.git/objects/`
  - This is true for newly tracked *as well as* updated files (i.e. Git does *not* only store the differences)

The interior of Git (somewhat technical, but essential!)

- Using `git add` stores each selected file as a *blob* (binary large object) in the directory `.git/objects/`
  - This is true for newly tracked *as well as* updated files (i.e. Git does *not* only store the differences)

- Using `git commit` stores a snapshot of the files currently being tracked. Only those in the staging area are updated
  - A *tree* object represents a directory containing the tracked files and refers to blobs (files) and possibly to other tree objects (subdirectories)
  - A *commit* object represents the snapshot and refers to a *tree* object and (possibly multiple) parent commit objects

The interior of Git (somewhat technical, but essential!)

- Using `git add` stores each selected file as a *blob* (binary large object) in the directory `.git/objects/`
  - This is true for newly tracked *as well as* updated files (i.e. Git does *not* only store the differences)

- Using `git commit` stores a snapshot of the files currently being tracked. Only those in the staging area are updated
  - A *tree* object represents a directory containing the tracked files and refers to blobs (files) and possibly to other tree objects (subdirectories)
  - A *commit* object represents the snapshot and refers to a *tree* object and (possibly multiple) parent commit objects

- The filenames of these *blob*, *tree* and *commit* objects are SHA1-hashes based on their contents (40 hexadecimal digits). Note that fewer digits suffice to refer to a hash (4+)

Demo time!

Overview of other Git commands we used (1)

- `git cat-file [hash]` to check the type (`-t`), size (`-s`) and contents (`-p`) of an object (blob, tree, commit)
- `git config section.variablename "value"` to configure Git (e.g. `user.name "Pieter Barendrecht"` or `core.editor "nano"`). Useful flags include `--global`
- `git diff` to check differences between working directory and latest commit (or differences between two commits)
- `git log` to look at the commit history. Useful flags include `--oneline` and `--graph`. Tip, create an alias using e.g. `git config alias.graph "log --all --oneline --graph"`

Overview of other Git commands we used (2)

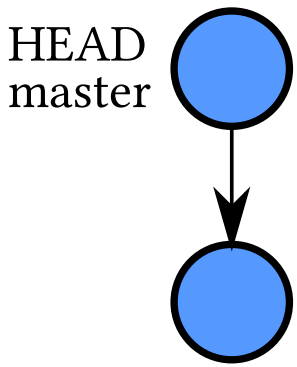
- `git branch` to list the current branches. A new branch can be created using `git branch [branchname]`
- `git checkout [branchname]` to switch to a different branch. Alternatively, use `git checkout -b [branchname]` to create a new branch and check it out directly
- `git merge [branchname]` to merge a branch into the *current* branch. It is a special type of commit, so use `-m` to add a description (why/what you are merging)
- `git help [command]` to get more information on using `[command]`, e.g. `git help merge`

Traversing the commit graph (1)

- Branch names (e.g. `master`) are merely *pointers to commits*
- `HEAD` points to the commit the current state of the working directory is based on, *usually* the tip of a branch

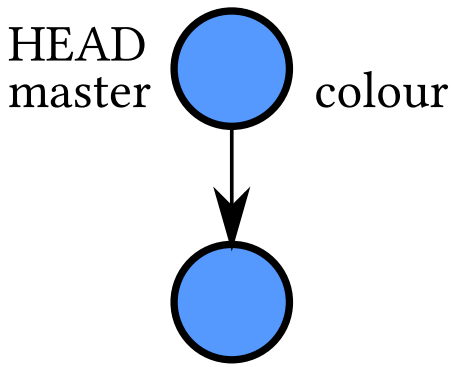
Traversing the commit graph (1)

- Branch names (e.g. `master`) are merely *pointers to commits*
- `HEAD` points to the commit the current state of the working directory is based on, *usually* the tip of a branch



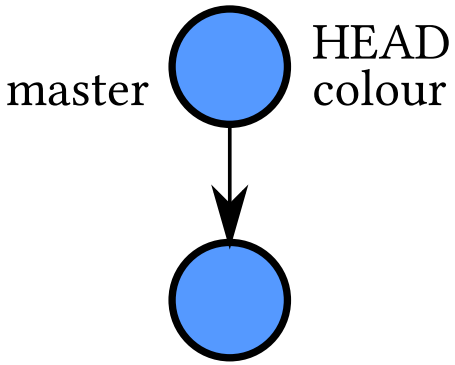
Traversing the commit graph (1)

- Branch names (e.g. `master`) are merely *pointers to commits*
- `HEAD` points to the commit the current state of the working directory is based on, *usually* the tip of a branch



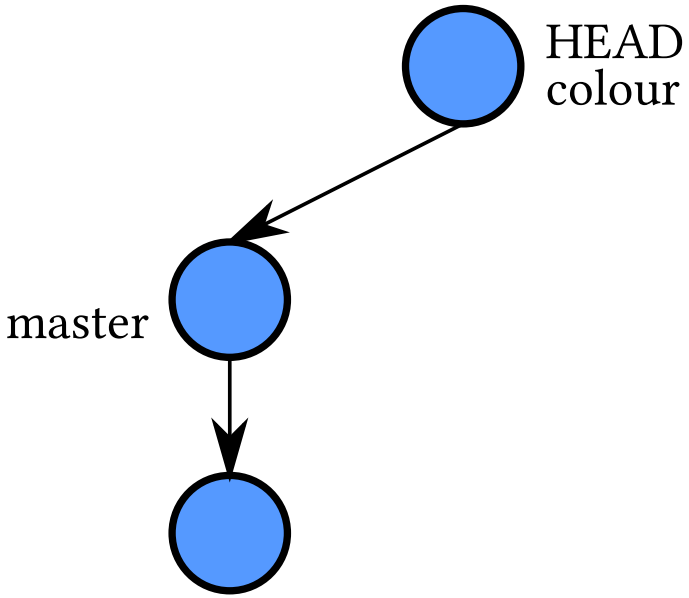
Traversing the commit graph (1)

- Branch names (e.g. `master`) are merely *pointers to commits*
- `HEAD` points to the commit the current state of the working directory is based on, *usually* the tip of a branch



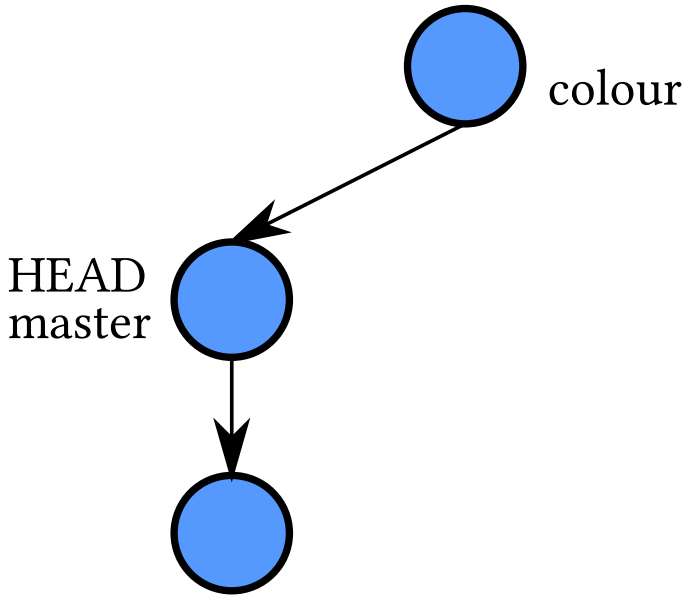
Traversing the commit graph (1)

- Branch names (e.g. `master`) are merely *pointers to commits*
- `HEAD` points to the commit the current state of the working directory is based on, *usually* the tip of a branch



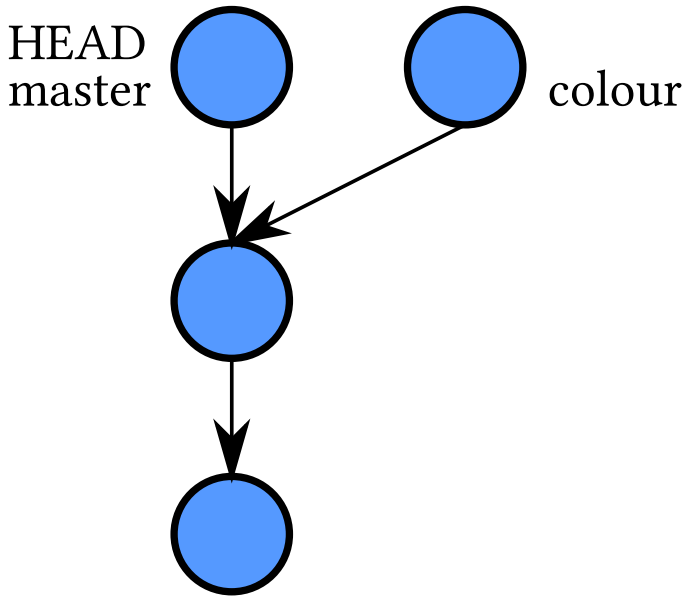
Traversing the commit graph (1)

- Branch names (e.g. `master`) are merely *pointers to commits*
- `HEAD` points to the commit the current state of the working directory is based on, *usually* the tip of a branch



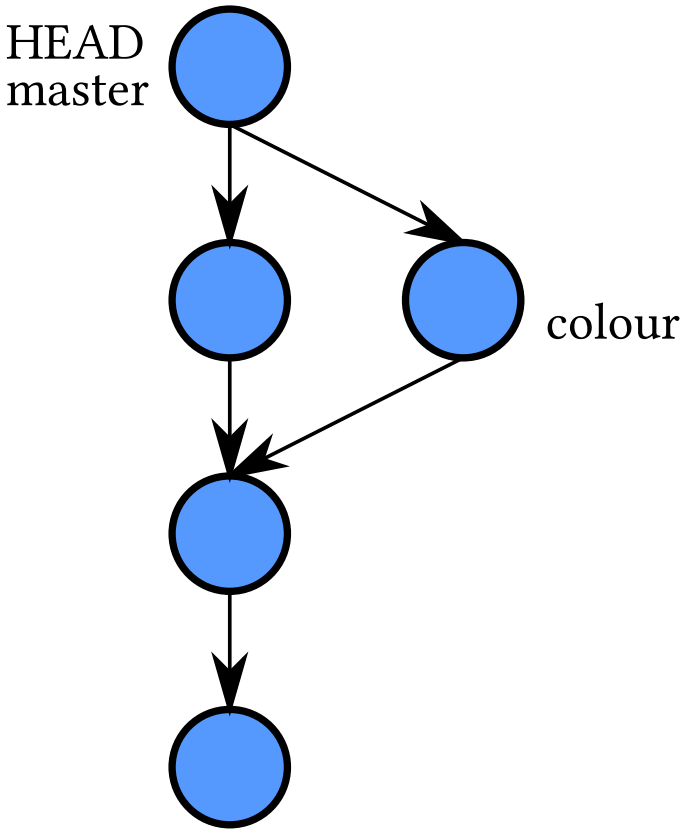
Traversing the commit graph (1)

- Branch names (e.g. `master`) are merely *pointers to commits*
- `HEAD` points to the commit the current state of the working directory is based on, *usually* the tip of a branch



Traversing the commit graph (1)

- Branch names (e.g. `master`) are merely *pointers to commits*
- `HEAD` points to the commit the current state of the working directory is based on, *usually* the tip of a branch



Traversing the commit graph (2)

- Use `git checkout [hash]` to check out any commit. This only updates the `HEAD` pointer. If the commit is not a branch tip, it results in a so-called *detached HEAD state*

Traversing the commit graph (2)

- Use `git checkout [hash]` to check out any commit. This only updates the `HEAD` pointer. If the commit is not a branch tip, it results in a so-called *detached HEAD state*

- Use `git reset --hard [hash]` to check out any commit. This updates both the `[branchname]` and the `HEAD` pointer
  - More recent commits on this branch are no longer referenced (and garbage-collected in due time)

Traversing the commit graph (2)

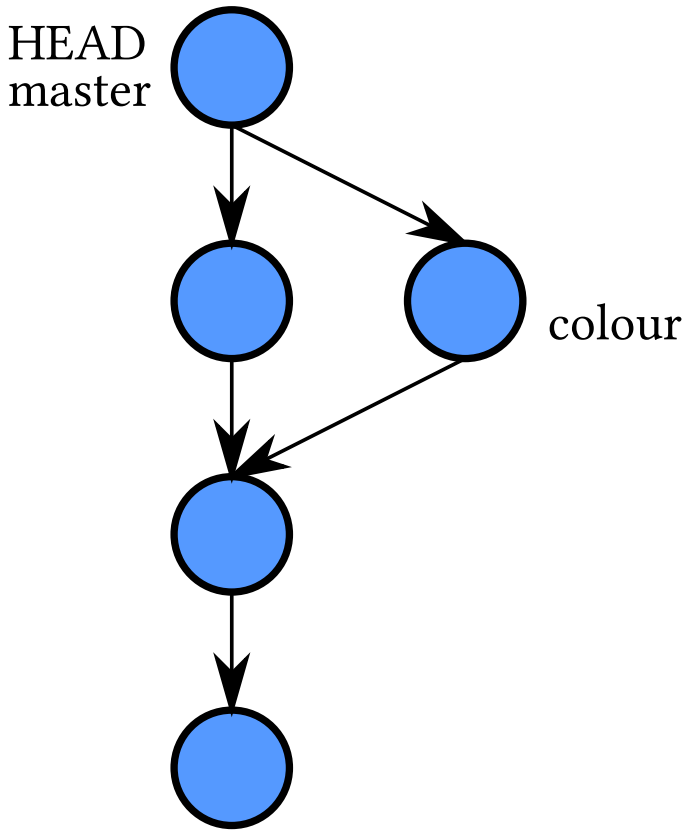
- Use `git checkout [hash]` to check out any commit. This only updates the `HEAD` pointer. If the commit is not a branch tip, it results in a so-called *detached HEAD state*
- Use `git reset --hard [hash]` to check out any commit. This updates both the `[branchname]` and the `HEAD` pointer
  - More recent commits on this branch are no longer referenced (and garbage-collected in due time)
- Use `git revert [hash]` to create *a new commit* reverting the changes since the selected commit

Traversing the commit graph (2)

- Use `git checkout [hash]` to check out any commit. This only updates the `HEAD` pointer. If the commit is not a branch tip, it results in a so-called *detached HEAD state*
- Use `git reset --hard [hash]` to check out any commit. This updates both the `[branchname]` and the `HEAD` pointer
  - More recent commits on this branch are no longer referenced (and garbage-collected in due time)
- Use `git revert [hash]` to create *a new commit* reverting the changes since the selected commit
- These are powerful commands. Experiment in a local setting, be careful in a shared setting! `git (ref)log` is your friend :)

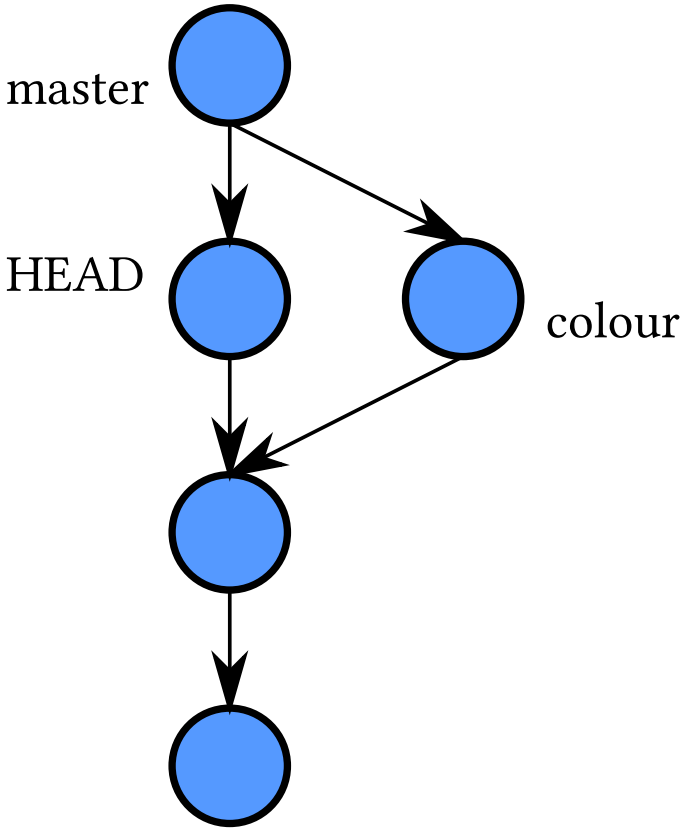
Traversing the commit graph (3)

- Example of `git checkout [hash]` that does not point to a branch tip



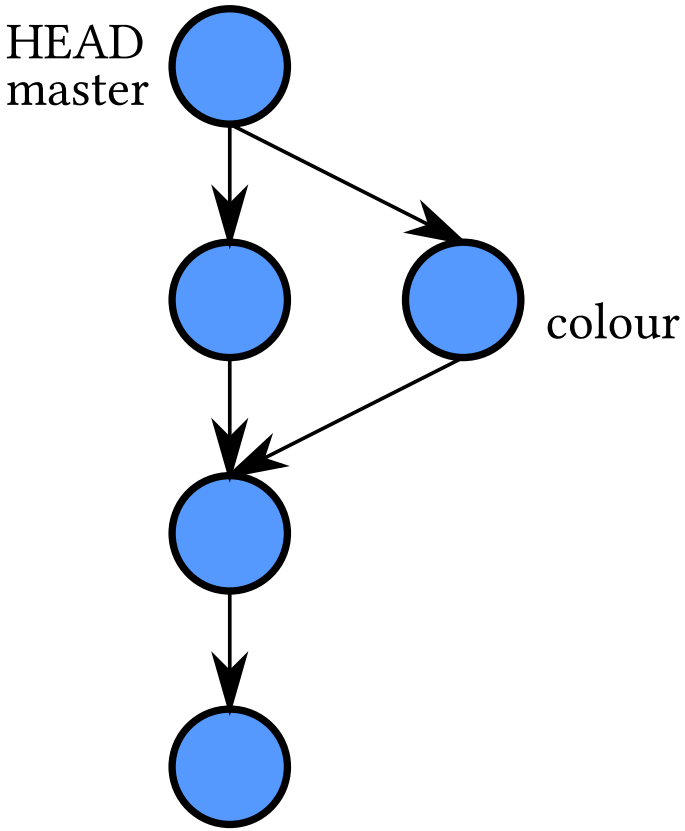
Traversing the commit graph (3)

- Example of `git checkout [hash]` that does not point to a branch tip



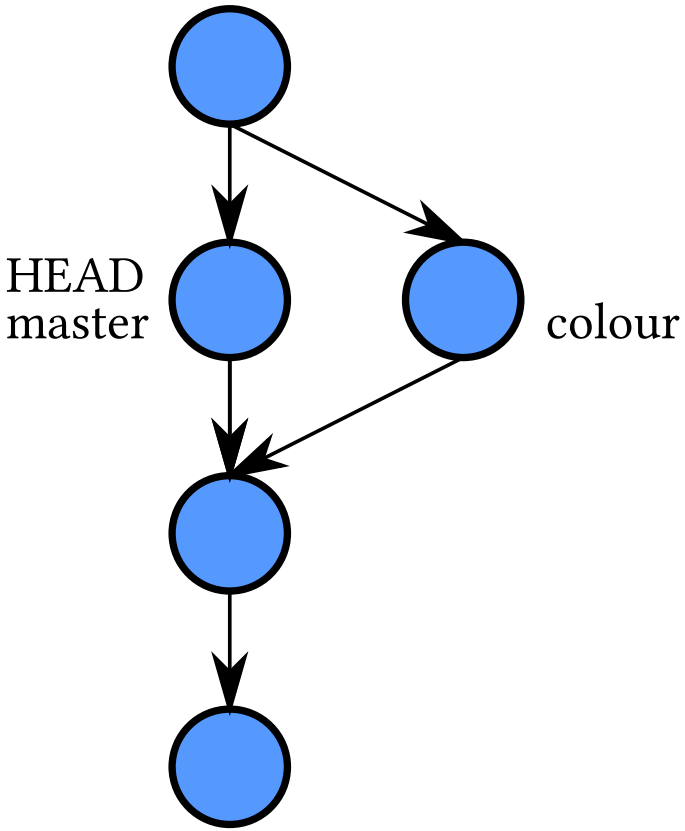
Traversing the commit graph (4)

- Example of `git reset --hard [hash]` while on the `master` branch



Traversing the commit graph (4)

- Example of `git reset --hard [hash]` while on the `master` branch



Merge conflicts

- If Git can not automatically merge (e.g. two different changes to the same part of a tracked file), you will have to *resolve the merge conflict manually*

Merge conflicts

- If Git can not automatically merge (e.g. two different changes to the same part of a tracked file), you will have to *resolve the merge conflict manually*
- Make life easier by using `git mergetool`, which allows you to use specialised editor (e.g. `meld`) to resolve the conflict. Configure using `git config --global merge.tool meld`

Merge conflicts

- If Git can not automatically merge (e.g. two different changes to the same part of a tracked file), you will have to *resolve the merge conflict manually*
- Make life easier by using `git mergetool`, which allows you to use specialised editor (e.g. `meld`) to resolve the conflict. Configure using `git config --global merge.tool meld`
- There are other aspects of merging that can be configured (e.g. whether to save backups or not, use `mergetool.keepBackup false`)

Merge conflicts

- If Git can not automatically merge (e.g. two different changes to the same part of a tracked file), you will have to *resolve the merge conflict manually*
- Make life easier by using `git mergetool`, which allows you to use specialised editor (e.g. `meld`) to resolve the conflict. Configure using `git config --global merge.tool meld`
- There are other aspects of merging that can be configured (e.g. whether to save backups or not, use `mergetool.keepBackup false`)
- Related: there is also `git difftool`, which allows you to look at differences in an editor of your choice

Demo time!

Other tips and tricks

- Add a custom `.gitignore` file to the working directory to avoid tracking specific files or file types (e.g. `.o`, `.pyc`, `.pdf`)

Other tips and tricks

- Add a custom `.gitignore` file to the working directory to avoid tracking specific files or file types (e.g. `.o`, `.pyc`, `.pdf`)
- Remove a file from the staging area with `git reset [file]`. To remove from the staging area (not the working directory) *and* stop tracking it, use `git rm --cached [file]`

Other tips and tricks

- Add a custom `.gitignore` file to the working directory to avoid tracking specific files or file types (e.g. `.o`, `.pyc`, `.pdf`)
- Remove a file from the staging area with `git reset [file]`. To remove from the staging area (not the working directory) *and* stop tracking it, use `git rm --cached [file]`
- Use `git stash` to *temporarily* commit modified tracked files as well as staged files. Revert to the old situation using `git stash pop`

Other tips and tricks

- Add a custom `.gitignore` file to the working directory to avoid tracking specific files or file types (e.g. `.o`, `.pyc`, `.pdf`)
- Remove a file from the staging area with `git reset [file]`. To remove from the staging area (not the working directory) *and* stop tracking it, use `git rm --cached [file]`
- Use `git stash` to *temporarily* commit modified tracked files as well as staged files. Revert to the old situation using `git stash pop`
- Sometimes `git rebase [hash]` (while on `[branchname]`) can be a good alternative for `git merge [branchname]` (e.g. while on `master`). See (online) documentation/examples

Other tips and tricks

- Add a custom `.gitignore` file to the working directory to avoid tracking specific files or file types (e.g. `.o`, `.pyc`, `.pdf`)
- Remove a file from the staging area with `git reset [file]`. To remove from the staging area (not the working directory) *and* stop tracking it, use `git rm --cached [file]`
- Use `git stash` to *temporarily* commit modified tracked files as well as staged files. Revert to the old situation using `git stash pop`
- Sometimes `git rebase [hash]` (while on `[branchname]`) can be a good alternative for `git merge [branchname]` (e.g. while on `master`). See (online) documentation/examples
- Many other useful *commands* (e.g. `git cherry-pick [hash]`, `git tag [hash]`) and *flags*, see documentation!

Interacting with remote repositories: GitHub (1)

- To *share an existing local repository* to GitHub, start with creating a new repository on GitHub

Interacting with remote repositories: GitHub (1)

- To *share an existing local repository* to GitHub, start with creating a new repository on GitHub
- Run `git remote add [remote reference] [url]` to create a connection to this remote repository. Typically, `[remote reference]` is chosen to be `origin`, but I find this confusing and use `github` instead

Interacting with remote repositories: GitHub (1)

- To *share an existing local repository* to GitHub, start with creating a new repository on GitHub
- Run `git remote add [remote reference] [url]` to create a connection to this remote repository. Typically, `[remote reference]` is chosen to be `origin`, but I find this confusing and use `github` instead
- Synchronise a local branch to the remote repo using `git push [remote reference] [local branch]`. Adding the `-u` flag sets up a tracking reference (more on next slide)
  - Note that `git push (...)` requires your GitHub username and password. Tip, add an SSH key to your GitHub account

Interacting with remote repositories: GitHub (2)

- Receive updates from a branch on the remote repo using `git pull [remote reference] [remote branch]` and merge them
  - If `git push -u (...)` was used to synchronise a local branch to GitHub, you can omit the arguments of `git pull`

Interacting with remote repositories: GitHub (2)

- Receive updates from a branch on the remote repo using `git pull [remote reference] [remote branch]` and merge them
  - If `git push -u (...)` was used to synchronise a local branch to GitHub, you can omit the arguments of `git pull`
- Alternatively, use `git fetch [remote reference] [remote branch]` to receive the updates. Merge them manually using `git merge [remote reference]/[remote branch]`. *This approach is preferred* over `git pull` as you can look at the received updates before merging them

Interacting with remote repositories: GitHub (3)

- Obtain a connected copy of a repository on GitHub using `git clone [url] [local directory]`. The remote reference is `origin` by default, which makes sense in this setting

Interacting with remote repositories: GitHub (3)

- Obtain a connected copy of a repository on GitHub using `git clone [url] [local directory]`. The remote reference is `origin` by default, which makes sense in this setting

- `git push (...)` and `git fetch (...)` work as discussed previously
  - You can only push to a remote repo if you are a collaborator (or owner). Otherwise, you will have to *fork* the repository and submit *pull requests* (more info after the demo)

Interacting with remote repositories: GitHub (3)

- Obtain a connected copy of a repository on GitHub using `git clone [url] [local directory]`. The remote reference is `origin` by default, which makes sense in this setting
- `git push (...)` and `git fetch (...)` work as discussed previously
  - You can only push to a remote repo if you are a collaborator (or owner). Otherwise, you will have to *fork* the repository and submit *pull requests* (more info after the demo)
- List available local branches using `git branch`, use the `-r` flag to list the remote ones, or `-a` to list all of them

Demo time!

More on GitHub: issues

- If you spot a bug (or something else that could/should be improved) in someone's code shared on GitHub, you can *create an issue* to address it. Be clear and to-the-point

More on GitHub: issues

- If you spot a bug (or something else that could/should be improved) in someone's code shared on GitHub, you can *create an issue* to address it. Be clear and to-the-point

- It is then up to collaborators of that repository to respond to it (or perhaps ignore it altogether). They might assign a collaborator to fix/improve it, or even ask you to provide a solution. Eventually, the issue should be resolved and closed

More on GitHub: issues

- If you spot a bug (or something else that could/should be improved) in someone's code shared on GitHub, you can *create an issue* to address it. Be clear and to-the-point
- It is then up to collaborators of that repository to respond to it (or perhaps ignore it altogether). They might assign a collaborator to fix/improve it, or even ask you to provide a solution. Eventually, the issue should be resolved and closed
- You can refer to specific commits using their hash, and there are certain keywords (e.g. "fixes") that automatically handle aspects of an issue (e.g. closing it)

More on GitHub: pull requests

- If you want to contribute to a repository on GitHub, but you are not (yet) a collaborator, you can *fork* it on GitHub (i.e. become the owner of an online copy of the repo you can then clone and work on locally)

More on GitHub: pull requests

- If you want to contribute to a repository on GitHub, but you are not (yet) a collaborator, you can *fork* it on GitHub (i.e. become the owner of an online copy of the repo you can then clone and work on locally)

- Make your changes in a separate branch, which you then push to your fork. Afterwards, you can create a *pull request*, which allows the collaborators of the original repository to merge your changes in their code (if they decide to do so)

Demo time!