**Git – Guidance Notes**

**Basic Git**

Overview

* Once changes have been made to a file such as new lines of code written, or an entirely new file added to the directory, the changes need to be added. This tells Git to track the file, and keep an eye on any further changes that are made.
* Once a set of files have been added, they need to be committed. This takes the changes that have been added, and commits them to the official history of the repository.
* We send it to the remote repository using a push function (covered later)
* To open a Git Bash terminal at your preferred location, navigate there using file explorer, then right click and click Git Bash here

|  |  |
| --- | --- |
| **Status**   * View the status of the repository | |
| git status | Shows the status of the repository, whether any changes have been made, whether there are any new files which haven’t yet been added, or whether any have been deleted or renamed. It’ll also tell you which branch you are on (more on this to come). |
| **Add**   * To add file modifications, or new files, the command git add is used. This needs to be followed by a parameter specifying what file or files need to be added. | |
| git add . | Add all created, modified, deleted or renamed files. This is the most commonly used action when starting out. |
| git add -u | Add all modified files - will not add any new files |
| git add 'preprocessing.py' | Add a specific file, 'preprocessing.py' |
| git add directoryname | Add a directory and its contents |
| **Commit**   * Once a set of files have been added, they can be committed. * This stores the changes as a permanent item in the history of the repo, with an associated message. It will only commit files that have been added, any untracked changes will not be included. * Adding a commit message is a crucial part in documenting your code. It allows others and your future self to understand what happened at a certain point in the development. * We should include useful information about the commit such as a description of the type of commit: * fix - a bug fix * feat - a new feature * doc - documentation changes | |
| git commit -m “string” | Indicates that the following string (in quotes) should be stored as the message for that commit. The message should be succinct and to the point, and state the changes in an impersonal manner made within that commit.  When doing git commit -m “messages” try to include the “branch: message” in there to help it make sense when you look back at it. |
| git commit | If you don’t include an -m “string” then Git Bash enters Vim, which is a text editor which allows for the editing of code within the Git Bash terminal. Avoid this when starting out. |
| **Log**   * To view previous commits, the repository log can be accessed. | |
| git log --oneline | --oneline is added to make the log more succinct - each commit gets added onto one line.  Without the modifer, each commit will be spread over about five lines, but carry far more detail about when it was made, and who it was made by. |

**Branches**

|  |  |
| --- | --- |
| **Checking current branches** | |
| git branch | Lists all local branches |
| git branch -a | Lists all local and remote branches |
| **Checking out (moving between branches)**   * If we want to make changes to a certain feature, we would checkout a new branch from the branch where that feature is being developed. * If we wanted to create an entirely new feature we could branch off from the develop branch. | |
| git checkout <branchname> | Takes you out of your existing branch and moves the repository to the branch specified |
| git checkout -b feature\_B | creates a new branch (feature\_B) and moves into it |
| git checkout --track origin/feature\_C | Create a local branch tracking `origin/feature\_C  NB. Origin refers to branches held on a remote server such as Github |
| git checkout 602a1ee | Checks out a specific commit rather than a branch |
| **Merging**   * Once work has been completed within a feature branch, it needs to be merged back in. * Before you merge a branch, you must move to the target branch by using git checkout targetbranch. * If there is a conflict when auto merging, then you will get a CONFLICT error message (development|MERGING) – these must be resolved by the user before the merge can be completed * Conflicting sections will be shown as follows:   <<<<<<< HEAD  dev code  can be found  in here  ...  =======  feature code  can be found  in here  ...  >>>>>>> feature\_B   * To resolve each conflict, edit the conflicting code to retain only the lines that you want, then delete the three identifying lines. * Repeat for each conflict - CTRL-F can be useful to find them * Once you have made these changes, add then commit them. The branch you are on in Git Bash will go from displaying (development|MERGING) to just (development). Your branches have now been merged. | |
| git merge mergingbranch targetbranch | Will take the commits from mergingbranch, and attempt to automatically merge them into targetbranch  NB. If the changes made within the mergingbranch do not conflict with any other commits within the targetbranch, the merge will be successful. |
| **Deletion**   * Branches should be deleted when they are no longer useful - normally after they have been merged successfully and since abandoned. * Care should be taken when removing branches - if they have not been merged successfully, their contents could be lost! | |
| git branch -d feature\_B | Delete branch feature\_B - Will only work if the branch has been merged! |
| git branch -D feature\_B | Delete branch feature\_B Will work regardless of whether the branch has been merge |

**Collaboration**

* Git can also be used with remote repositories, which are provided by online services such as Github or Gitlab. Which service you use is a matter of context and it’s always best to check with your organisation.
* Github is free to sign up to and considered the most user-friendly for people new to Git and version control.
* The ONS uses a system called GitLab, an open source alternative, which allows the department to host repos within the secure ONS ecosystem.

|  |  |
| --- | --- |
| **Linking Remote to Local**   * Done through the use of a Personal Access Token ([Managing your personal access tokens - GitHub Docs](https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens)) | |
| **Create a repository**   * Follow this process to create a remote repository on Github ([Create a repo - GitHub Docs](https://docs.github.com/en/get-started/quickstart/create-a-repo)) | |
| **Clone the remote repository on your local machine**   * Follow this process to create a local version of a remote repository ([Cloning a repository - GitHub Docs](https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository)) | |
| git clone <the HTTPS string from your clipboard> <folder name> | This will clone the remote repo to the given folder name (using the https string that you have accessed using the clone function within your repo on Github) |
| **Fetch and Pull (**[Getting changes from a remote repository - GitHub Docs](https://docs.github.com/en/get-started/using-git/getting-changes-from-a-remote-repository))   * The user needs to manually ensure that changes on the remote repository are tracked by our device - we must explicitly ask git to check what changes have been made since we last updated our local repo. * This is achieved by calling the git fetch command. By fetching we ask the remote repo to inform our device of all the updates to the branches that we are tracking. * We can add in the additional argument after fetch that specifies which remote repo we are referring to. By default the remote repos are assumed to be called origin. * Note: This does not change our files on our branches, it just updates the information we have about the remote tracked branches. * To actually get the changes made on the remote repo to our device we use a different command, pull. * When we use the pull command we are merging our current branch with a certain branch on the remote repo. If the merge is successful our local branch will therefore be up-to-date with the remote repo branch. * You should ensure that all changes to your local repository branch have been committed (called a clean working copy) before you try to pull from the remote repo to avoid untracked changes. * As git pull merges the remote repo to your local repo merge conflicts can occur. These can be handled in the same way as discussed in the ‘Merges’ section. * Performing a pull will automatically perform a fetch command first. We use fetch on its own to understand what has changed on the remote repo. * We can check the state of our local repos in relation to the remote repository using the command we have already introduced status. | |
| git fetch origin | This fetches the information about our origin remote repo and shows us the changes at the remote end. |
| git pull origin dev | This will list all of the changes made to your local file, including insertions, deletions and files created/removed. These are dealt with as per merges above. |
| git status | Check the status of our local repos in relation to the remote repository. |
| **Tracked and Untracked Branches**   * We make the distinction between tracked and untracked branches. * Tracked branches are those which are connected explicitly to a remote repository branch. * An untracked branch is one which is just local to the device being used. * We can only pull/fetch and do other remote interacting operations with branches that are tracked. | |
| **Push** ([Pushing commits to a remote repository - GitHub Docs](https://docs.github.com/en/get-started/using-git/pushing-commits-to-a-remote-repository))   * To update the remote repo so it has our changes integrated is done by using the push command. * The push command allows you to merge a local branch to a remote branch. * When to Push - rule of thumb * Commit when a “unit” of work is done; such as a new function made or part of a script written. * Push when a complete feature is made. This feature shouldn’t contain any bugs and should work well with the rest of the code. | |
| git push <remote name> <branch name>  eg. git push origin dev | This would push our changes to the dev branch, within the remote repo called origin |
| **Working with Existing Projects**   1. Ensure that you have a Git area on your local machine. This is achieved using git init, you should have one area to store all your projects. 2. Create a repo on the remote service you are using. 3. Clone that repo into your git area such that a folder connected to the remote exists. 4. Move the files from the project you are working on into the new cloned folder. 5. add, commit and push the files up to the remote repo. | |
| **Merging (**[About pull requests - GitHub Docs](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests))   * We can now update remote repo branches with our local changes by using push. This allows us to alter the contents of a branch, but not to combine features from our different branches. For this we want to merge branches in the remote repo. * Note: GitHub’s terminology isn’t always very intuitive compared to other services (such as Gitlab). To create a merge between branches within the platform, you must use the “pull request” feature. The image below shows this feature. * Rather than use the Github webpage to carry out merges, we can also do this via the command line * First you would add then commit to the branch you are working on. * Then you would fetch/pull the target branch you would like to merge to. * You would then merge locally between your source and target branch, resolving any conflicts. * And finally you would push up to the remote target branch. | |

**Reverting Changes**

* git reset
  + This allows us to reset to a particular commit
    - eg. git reset db3c3ac
  + We have undone the commits but not removed the files
    - git status will show that the previous actions are now untracked and awaiting being added and committed
    - We can then either delete the files or change them in any way we want to, then re-add and re-commit
  + We can also do a hard reset
    - Eg. git reset db3c3ac --hard
    - This removes all changes completely – nuclear option
  + Warning
    - Never use “git reset” when you have pushed (uploaded, see below) changes to a remote repository that others are collaborating on.
    - They may have been working from the version you have just reset, which would lead to version (merge) conflicts.
    - Only use git reset –hard when you are only working on a local repository.
* git revert
  + Instead of “rewriting” the history of a branch, “git revert” actually creates a new commit that undoes the changes you made.
    - The history of the branch and repository remains intact, but we add an “undo commit”.
    - Much better option than “git reset” if you are working with a repository that others are collaborating on.
    - Eg. git revert b39b1aa
      * You need to use the commit reference for the commit you want to revert to, i.e. not the target one.
  + When you carry out a git revert action this opens the “vi” editor.
    - Allows users to edit the commit message for the new commit “git revert” is making for you.
    - To save and get out of this editor, press “:” once, and then type “wq” (“write and quit”), and press Enter. This takes you back to the CLI, and the commit is then completed for you.
  + Key difference to git reset
    - Instead of “rewinding” the commit history and removing things you have now undone changes by effectively adding a change to the history.
    - This is a better way of doing things as the history of the repository is retained; this is especially vital if you are collaborating with others.