**Version Control with GitHub & Git**

So here is a small hand-out that I created on all the things we need to know (in theory, I haven’t fully tried it yet, but we can create fake pull requests later to see whether it works as we want it to) in order to use GitHub and Git (goes together).

**Which interface to use?**

There are several different interfaces for GitHub/Git, which might get confusing. The online <https://github.com> is necessary and has a nice interface, but shouldn’t be used on its own because it can’t help solving merging conflict. The only way to deal with conflicts is really to use command lines, at least partially, but no worries this is really simple and intuitive. There are about 10 simple git commands to know in order to be able to use GitHub properly, and it’s almost as if you had a click-button system!

**Getting started**

So to get started with the command line, you first have to create on account on GitHub online, and then install GitHub desktop on your computer/laptop, which you can download from <https://windows.github.com> or <https://mac.github.com> according to which OS you’re using. I’m personally using Windows, so the rest of the hand-out is meant for Windows-like interface, but I’m sure it is pretty similar under Mac (subject to few exceptions). Once it’s installed, DO NOT use the GitHub Desktop visual interface, it’s useless. Look for the command line version, handily called Git Shell, or add the PATH of Git command line (git.exe file) to the environment variables of your computer, so that you can use git commands directly through windows command line (see at the end for more information on how to do this).

**GitHub Online**

This is a more interactive view of your GitHub account and all your repositories. You can easily create a new repository, and it’s always better to automatically and systematically create a README.md file with instructions to people who will use this repository. This README file is the first thing you see when entering a repository. GitHub online interface is also where you can delete repositories and branches. And of course, this is where you share the files with others! So basically you work on your project with GitHub Command Line opened on your computer, but you have to push all your work to [github.com](http://www.github.com) anyways, so that the others can access updates and changes, and it’s best to pull all the updates down to your command line each time you start working, and regularly throughout time, to make sure you have all updates from others too.

When you go to Pull Requests on GitHub, you have an interactive view of all the changes and differences/conflicts (with colours, etc.), which makes it easier to figure out what’s going on in your repository and why the hell you get a merging conflict when you try to merge files in the command line. Once a pull request is submitted, you can have a conversation with other users whenever there are some changes to be made before the pull request is accepted for example. You can then add new commits with improvements to the existing pull request. You can even put comments directly on the line of the commit that needs changes before accepting the pull request. And you can also directly address a specific person in your comments with a “@” like on twitter: the concerned person will thus receive a kind of notification to say that you want her/him to have a look at your change or whatever.

**GitHub Command Line**

git provides a list of all the commands available with git, with a description of what they do. Not all of them are actually relevant to us (well, I still have to check though). The most useful commands are listed below.

git clone *<your\_repository\_url>* clones the repository you want to work on from [github.com](http://www.github.com) into your current working directory. This can be chosen with cd command line. To check whether the repository has indeed been added, type dir (if you’re using Windows) or ls (under Mac/Unix Shell). Your repository URL can be copied from [github.com](http://www.github.com) and pasted in the shell: when you’re in the online repository, look for the symbol that says “Copy to clipboard”.

git status summary of your working status, all files status, etc.

git add *<filename.ext>* adds *<filename.ext>* contents to the index.

\_\_\_ \_\_\_ .

add everything to the index.

\_\_\_ \_\_\_ -A

git rm removes a file or a directory.

git commit commits with a multi-line message. When done writing the message, use esc:wq. You can also directly type esc:wq in case you don’t want to submit a comment at all.

\_\_\_ \_\_\_\_\_\_ –m “*your\_comment*” commits with a single-line comment.

\_\_\_ \_\_\_\_\_\_ -v commits with the differences of your change in the editor so you can see exactly what changes you’re committing.

git push push data from your computer up to [github.com](http://www.github.com)

git pull pull data from [github.com](http://www.github.com) down to your computer

git branch lists all the branches in your repository.

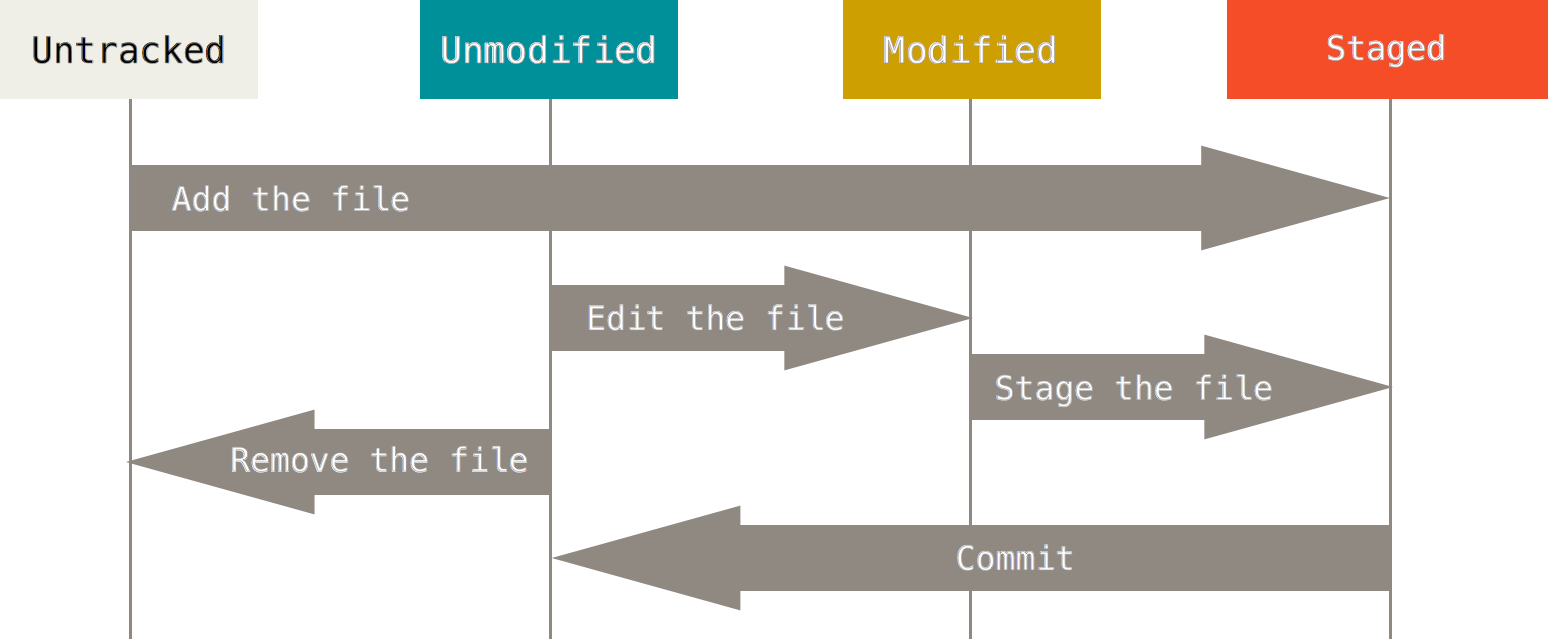
\_\_\_ \_\_\_\_\_\_ <*new\_branch>* adds <*new\_branch>* to your repository. This is a complete copy of master branch at the time it is created. You can check that it has been added with git branch.

git checkout <*branch>* switch to <*branch>*.

git merge *master* merge current working <*branch>* with <*master>* branch. A conflict might appear, which means that somewhere along the lines, somebody touched the same lines of a file as you. It has to be fixed manually directly into the file. GitHub shows you which lines are in conflict, and what are the different versions of these lines. If the conflict or GitHub description (might be confusing sometimes) doesn’t make sense, you can go to [github.com](http://www.github.com) and checkout each branch to have a better idea of what’s going on and how exactly did the conflict occur.

**GitHub Workflow**

*Step 1: Do your own thing on your own computer*



git commit

git add

git add

git rm

*Fig 1. The lifecycle of GitHub files status.*

Use git status to check the status of your files.

***NEW FILE***

If a new file has been created, it is listed as untracked. Untracked basically means that Git sees a file you didn’t have in the previous snapshot (commit); Git won’t start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don’t accidentally begin including generated binary files or other files that you did not mean to include.

To tell Git to start tracking a new file, you can run the command git add. It is now ready to be committed (*i.e.* “Staged” in Fig. 1). You can tell that it’s staged because it’s under the “Changes to be committed” heading in the git status output. The git add command takes a path name for either a file or a directory; if it’s a directory, the command adds all the files in that directory recursively.

***ALREADY EXISTING FILE***

If you change a previously tracked file and then run git status command again, you get something like:

C:\Users\Claire\Documents\GitHub\Bats [master +0 ~1 -0]> git status

On branch master

Your branch is up-to-date with 'origin/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: githubtest.py

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

… Which is pretty cool because it basically tells you what you should do with your modified file: just git add your file already! Once you’ve done this, it is staged and ready for committing! git status will then indicate that you have files with changes to be committed.

***COMMIT***

Now you just have to run git commit. Every time you perform a commit, you’re recording a snapshot of your project that you can revert to or compare to later. ***WARNING:*** Git stages a file exactly as it is when you run the git add command. If you commit now, the version of your file as it was when you last ran the git add command is how it will go into the commit, not the version of the file as it looks in your working directory when you run git commit. If you modify a file after you run git add, you have to run git add again to stage the latest version of the file. Otherwise your file will be listed as both staged and unstaged, and only the staged changes will be committed.

Although it can be useful for crafting commits exactly how you want them, the staging area is sometimes a bit more complex than needed in the workflow. If you want to skip the staging area, add the -a option to the git commit command. It makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part. It’s convenient and faster alright, but I think we should keep the git add step as an automatism in order to avoid including unwanted changes by accident.

***MOVE AND REMOVE***

Git doesn’t explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact. And yet it has a move command. Git considers a moved file as a renamed file, so you can also use git mv to simply rename a file, and it works perfectly fine. So the command line goes like this: git mv <*file\_from>* <*file\_to>*. I didn’t add it in the commands list because I don’t think it’s a very important command.

You can remove files from Git by removing them from your staging area and then commit with the command git rm. It also removes the file from your working directory so you don’t see it as an untracked file the next time around. If you simply remove the file from your working directory, it shows up under the “Changed but not updated” (that is, unstaged) area of your git status output, while git rm stages your file removal. If you modified the file and added it to the index already, you must force the removal with the -f option. This is a safety feature to prevent accidental removal of data that hasn’t yet been recorded in a snapshot and that can’t be recovered from Git.

***OTHER DETAILS***

If git status is too wordy for you, you can get the shorter version with git status –s or git status –-short. But I personally find git status (long version) way more comprehensive. Maybe I’ll start using git status –s when I’ll be more comfortable with GitHub. The nomenclature goes as follows:

?? untracked file.

A new file that has been added to the staging area.

M modified file.

There are 2 columns in the output: the left-hand side is for staging area status while the right-hand column indicates status in working tree. Thus, MM corresponds to a file that has been modified, staged, and modified again, so there are both staged and unstaged changes.

For more details on your changes, you can use git diff. It answers to the following questions: What have you changed but not yet staged? And what have you staged that you are about to commit? Although git status answers those questions very generally by listing the file names, git diff shows you the exact lines added and removed – the patch, as it were. To see what you’ve changed but not yet staged, type git diff with no other arguments. That command compares what is in your working directory with what is in your staging area. If you want to see what you’ve staged that will go into your next commit, you can use git diff –staged (you can also use git diff –cached to see what you’ve staged so far, as --staged and --cached are synonyms). This command compares your staged changes to your last commit. ***WARNING:*** git diff by itself doesn’t show all changes made since your last commit – only changes that are still unstaged. This can be confusing, because if you’ve staged all of your changes, git diff will give you no output.

.gitignore is for files that you don’t want Git to automatically add or even show you as being untracked, as in automatically generated files such as log files or files produced by your build system for instance. I just had a quick glance on this function, as I don’t think it essential for us to know it, at least for now. It’s just good to know that it exists, just in case I guess. See also git rm --cached.

*Step 2: Share your work with your co-workers*

This means using the last commands in the list, such as git branch, git pull, git push and git merge. An important thing is to ***ALWAYS*** start your day by running git pull, just to make sure you have the latest update of the repository, including possible changes from your co-workers! It’s a priority, and even comes before having your morning coffee :p! And remember to regularly type git pull as a sync update, although we are only 2 people working on the project so I guess it’s not like the files are going to be changed all the time.

Then you can start working on your branch. To see you current branches, simply type git branch. The branch you are currently working in is indicated with an asterisk \* before and is coloured in green (at least that’s how it looks like in my working environment… Probably depends on your computer’s settings though). It is also written in between brackets at the beginning/heading of your command line, and designed with technicolour, so there’s no way you can get confused about the branch you’re using! You can create a new branch directly from the master branch, or from a sister branch (thus building embedded branches. I have no clue in which situation this might be useful, but good to know anyway!). So just go to the branch you want to copy, let’s call it <*original\_branch>*, using git checkout <*original\_branch>* and run git branch <*new\_branch>*. This will create a complete replica of your <*original\_branch>* at the time you copied it. Now this <*new\_branch>* appears when you run git branch.

You may then work on your file, do all the changes you want, and once you’re done, save them and commit them as described in Step 1: Do your own thing on your own computer. Once you think you have a code that works and you’d like to save it into the online repository, you supposedly just have to run git push. The first time I did this though, I had some bugs (see De-bugging below). But once I’ve figured it out, here’s how your workflow should look like:

git branch <feature>

git checkout <feature>

# make your changes on your file, save them.

git add <filename.ext>

git add .

git add -A

git commit -m “<comment>”

git checkout master

git pull

git checkout <feature>

git merge master

# Conflict: solve it manually, save changes.

git status

git add –A

git commit

esc:wq

git push

git push --set-upstream origin <feature>

After having pushed your updates up to GitHub Online, and if you consider that these changes are ready to be included in the master branch, go on [github.com](http://www.github.com) and create a pull request.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DE-BUGGING

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

14/04/2016

Okay so the first time I tried to run git push, I got the following error message:

fatal: The current branch commit has no upstream branch.

To push the current branch and set the remote as upstream, use

git push --set-upstream origin commit

So I tried to run the command below, as advised on a thread in Stack Overflow. Notice that it was a stupid thing to do, as I didn’t even fully understand the command. Note to myself: don’t do this ever again.

git push -u origin --all

Luckily it didn’t do anything too wrong, I just got this error message:

fatal: unable to access 'https://github.com/ClaireGuerin/Bats.git/': SSL read: error:00000000:lib(0):func(0):reason(0), errno 10054

Which very much looks like I cannot access GitHub Online! So as I didn’t know what the line was doing anyway, I started seeking help elsewhere on the amazing web, and turns out the original problem comes from the fact that the branch has not already been published, *i.e.* it is only on my computer… Which is exactly why I wanted to push it on the first place! Now I somehow ended up looking at the definitions of remote, local branches, local tracking branches, etc. (to have the full explanation on all this (very important as I found out), check out at the end of the document). Well, it seems like my branch commit is not set to track any other branch (which should ideally be master). You check this by running the command git branch –vv. I get the following output:

\* commit d462740 Merge pull request #2 from ClaireGuerin/claire

master ab39393 [origin/master: ahead 1] Let's see...

There’s no square brackets whatsoever on the commit branch line. This means that it’s a non-tracking local branch, which ultimately means that I cannot use git push for this branch. Great. Now I’m also starting to get the feeling that somehow my master branch ended up 1 change ahead from my commit branch, which might be another reason for why the push would not work. Indeed the “ahead 1” seems like a pretty strong clue (thanks Sherlock), as well as the “Let’s see…”, which is a comment I added after committing a change to my commit branch… Or so I thought! I am also wondering how did the “Merge pull request #2 from ClaireGuerin/claire” appear here, as on my desktop I am not able to see the branches we created previously other than master, such as claire… Another thing to figure out!

15/04/2016

Okay, so no panic, I deleted the Bats repository. As it was kind of a mess and we didn’t have anything important, I thought it would be easier to start all over again. In case we ever want to delete a repository again, it’s easy to do: you go on [github.com](http://www.github.com), under Settings in the repository you want to delete, go to Options, Delete this Repository (in the Danger Zone), confirm by entering the name of the repository you want to delete, and that’s it!

Still online, I created a new repository called “Learn-GitHub”, where we can play around a little bit and try to do things properly. There is a README file that explains the goal of the repository (although it is pretty obvious). Now I switch to Git Shell and see what happens.

Clone Learn-GitHub repository into my Desktop. Run ls to check that it has been added and… Surprise! Bats is still here. I probably have to update my Desktop with git pull. Here is what I learned from running this (might be trivial, be still I write it down just in case):

1. git pull is meant for unique repositories, you cannot pull data from all your repositories at the same time, but instead you have to enter the repository you want to update.
2. Bats repository is not found. So it has been deleted, but I still have the local directory on my computer.

So basically what I did is that I deleted the directory from my computer. It sounds very brutal and not ideal, I don’t think it’s the most subtle way to deal with the problem, but I couldn’t find any other way on the internet. So now I get:

C:\Users\Claire\Documents\GitHub> ls

Repository: C:\Users\Claire\Documents\GitHub

Mode LastWriteTime Length Name

---- ------------- ------ ----

d----- 15/04/2016 10:28 Learn-GitHub

So far so good. Let’s see how things are going on when we work around a little bit.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

$ git branch

\* master

$ ls

Repository: C:\Users\Claire\Documents\GitHub\Learn-GitHub

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 15/04/2016 10:28 574 README.md

Now I added a python script into Learn-GitHub directory, called fibo.py.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

fibo.py

nothing added to commit but untracked files present (use "git add" to track)

$ git add fibo.py

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: fibo.py

$ git commit -m "Added fibo.py file that contains a Fibonacci suite."

[master 436e6c3] Added fibo.py file that contains a Fibonacci suite.

1 file changed, 13 insertions(+)

create mode 100644 fibo.py

Now everything worked so far, we still need to push the file to the shared repository online… \*drum rolls\*…

$ git push

Counting objects: 3, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 557 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/ClaireGuerin/Learn-GitHub.git

a59f48a..436e6c3 master -> master

… YES! And fibo.py even appears on GitHub Online!!! Okay so when I work on master branch, everything seems fine, but when I start working with parallel branches, this is where everything goes south. If I understand well my mistake(s) from yesterday, I apparently created a sister branch of master, but still did my changes on the files in the master branch, leading to master being ahead of the sister branch, which is useless and stupid. So let’s see if I do it right this time:

git branch testing

git branch

\* master

testing

git checkout testing

Switched to branch 'testing'

ls

Repository : C:\Users\Claire\Documents\GitHub\Learn-GitHub

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 15/04/2016 11:53 393 fibo.py

-a---- 15/04/2016 10:28 574 README.md

$ git branch -vv

master 436e6c3 [origin/master] Added fibo.py file that contains a Fibonacci suite.

\* testing 436e6c3 Added fibo.py file that contains a Fibonacci suite.

And here my first mistake of the day! testing in not a tracking branch. Let’s see what happens anyway. Okay so now I am in testing branch. I have fibo.py opened in Spyder, I add a new line and save the change. Back to Git Shell:

$ git status

On branch testing

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: fibo.py

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

$ git add .

$ git commit -m "Number of parents also printed"

[testing b344485] Number of parents also printed

1 file changed, 1 insertion(+)

$ git checkout master

Switched to branch 'master'

Your branch is up-to-date with 'origin/master'.

$ git pull

Already up-to-date.

$ git checkout testing

Switched to branch 'testing'

$ git merge master

Already up-to-date.

$ git status

On branch testing

nothing to commit, working directory clean

$ git push

fatal: The current branch testing has no upstream branch.

To push the current branch and set the remote as upstream, use

git push --set-upstream origin testing

$ git push --set-upstream origin testing

Counting objects: 3, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 376 bytes | 0 bytes/s, done.

Total 3 (delta 1), reused 0 (delta 0)

To https://github.com/ClaireGuerin/Learn-GitHub.git

\* [new branch] testing -> testing

Branch testing set up to track remote branch testing from origin.

Okay great! It worked, I just had to follow the working flow showed in the second lesson of the video tutorial (somehow I didn’t), and everything’s fine. On [github.com](http://www.github.com), I can see my testing branch, with the updated file ready for pull request if I wish, and the master branch not yet updated. I can keep on making changes on my file this way, and whenever I think the code is ready to be added to the master branch, I can create a pull request that will be checked by other contributors. And that’s it! I haven’t had to deal with a merge conflict yet though.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

END OF DE-BUGGING

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Useful links**

*LearnCode.academy* short tutorials on how to get a hand on git. Very well done, clear & goes directly to the point:

1. [First lesson](https://www.youtube.com/watch?v=0fKg7e37bQE): GitHub Tutorial for Beginners - GitHub Basics for Mac or Windows & Source Control Basics. ***18:53***
2. [Second (and last) lesson](https://www.youtube.com/watch?v=oFYyTZwMyAg&nohtml5=False): GitHub pull request, Branching, Merging & Team Workflow. ***11:20***

Another [tutorial](http://readwrite.com/2013/09/30/understanding-github-a-journey-for-beginners-part-1/#feed=/author/lauren-orsini&awesm=~omyHpouFOs03MN) (not a video). Haven’t read it yet but seems pretty good.

[This](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository) is GitHub help on the lifecycle status of your files, which I summarize in GitHub workflow – Step 1, but has more details and insights into the possibilities if you want to check it out… Or even if my summary seems too obscure/not understandable at all ha ha!

[Here](https://www.mutuallyhuman.com/blog/2012/06/22/a-git-walkthrough/) is the webpage that helped me figure out the git push problem. It actually treats other classical Git workflow issues, so interesting to check. I haven’t fully read it yet.

<https://help.github.com/articles/commit-exists-on-github-but-not-in-my-local-clone/>

<http://genomewiki.ucsc.edu/index.php/Working_with_branches_in_Git>

<http://www.gitguys.com/topics/creating-and-playing-with-branches/>

<http://stackoverflow.com/questions/7856416/view-a-file-in-a-different-git-branch-without-changing-branches>

**How to use git command lines with Windows cmd?**

1. Go to My Computer 🡪 Local Disk(C:) 🡪 Program Files(x86) 🡪 Git 🡪 cmd
2. Right Click the git 🡪 Select Properties
3. Under the location Copy the text *e.g.* - C:\Program Files (x86)\Git\cmd
4. Come back to the Desktop
5. Right-click My Computer
6. Select property
7. Open Advanced
8. Click Environment Variables
9. In the System variables Find the Variable call Path
10. Click the variable
11. Click the Edit Button
12. Select the Variable value Text Box
13. Go to the edge of the text and put semicolon (;)
14. Then Right-click and press Paste
15. Press Ok

**What is the difference between local branch, local tracking branch, remote and tracking remote?**

Here's the long answer (copied from a [Stack Overflow answer](http://stackoverflow.com/questions/16408300/what-is-the-differences-between-local-branch-local-tracking-branch-remote-bran)):

1. Remotes

If you're using Git collaboratively, you'll probably need to sync your commits with other machines or locations. Each machine or location is called a **remote**, in Git's terminology, and each one may have one or more branches. Most often, you'll just have one, named origin. To list all the remotes, run git remote. You can see which locations these remote names are shortcuts for, by running git remote -v:

$ git remote -v

origin git@github.com:Flimm/example.git (fetch)

origin git@github.com:Flimm/example.git (push)

Each remote has a directory under git/refs/remotes/:

$ ls -F .git/refs/remotes/

origin/

1. Branches on your machine

TLDR: on your local machine, you've got three types of branches: **local non-tracking branches**, **local tracking branches**, and **remote-tracking branches**. On a remote machine, you've just got one type of branch.

* 1. Local branches

There are two types of local branches on your machine: non-tracking local branches, and tracking local branches. You can view a list of all the local branches on your machine by running git branch. Each local branch has a file under .git/refs/heads/.

$ git branch

master

new-feature

$ ls -F .git/refs/heads/

master new-feature

* + 1. Non-tracking local branches

Non-tracking local branches are not associated with any other branch. You create one by running git branch <branchname>.

* + 1. Tracking local branches

Tracking local branches are associated with another branch, usually a remote-tracking branch. You create one by running git branch --track <branchname> [<start-point>]. You can view which one of your local branches are tracking branches using git branch -vv:

$ git branch -vv

master b31f87c85 [origin/master] Example commit message

new-feature b760e04ed Another example commit message

From this command's output, you can see that the local branch master is tracking the remote-tracking branch origin/master, and the local branch new-feature is not tracking anything. Another way to see which branches are tracking branches is by having a look at .git/config. ***Tracking local branches are useful. They allow you to run git pull and git push, without specifying which upstream branch to use.*** If the branch is not set up to track another branch, you'll get an error like this:

$ git checkout new-feature

$ git pull

There is no tracking information for the current branch.

Please specify which branch you want to merge with.

See git-pull(1) for details

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream new-feature <remote>/<branch>

* 1. Remote-tracking branches (still on your machine)

You can view a list of all the remote-tracking branches on your machine by running git branch –r. Each remote-tracking branch has a file under .git/refs/<remote>/.

$ git branch -r

bitbucket/master

origin/master

origin/new-branch

$ tree -F .git/refs/remotes/

.git/refs/remotes/

├── bitbucket/

│ └── master

└── origin/

├── master

└── new-branch

Think of your remote-tracking branches as your local cache for what the remote machines contain. You can update your remote-tracking branches using git fetch, which git pull uses behind the scenes. Even though all the data for a remote-tracking branch is stored locally on your machine (like a cache), it's still never called a local branch. (At least, I wouldn't call it that!) It's just called a remote-tracking branch.

1. Branches on a remote machine

You can view all the remote branches (that is, the branches on the remote machine), by running git remote show <remote>:

$ git remote show origin

\* remote origin

Fetch URL: git@github.com:Flimm/example.git

Push URL: git@github.com:Flimm/example.git

HEAD branch: master

Remote branches:

io-socket-ip new (next fetch will store in remotes/origin)

master tracked

new-branch tracked

Local ref configured for 'git pull':

master merges with remote master

new-branch merges with remote new-branch

Local ref configured for 'git push':

master pushes to master (up to date)

new-branch pushes to new-branch (fast-forwardable)

This git remote command queries the remote machine over the network about its branches. It does not update the remote-tracking branches on your local machine, use git fetch or git pull for that. From the output, you can see all the branches that exist on the remote machine by looking under the heading "Remote branches" (ignore lines marked as "stale"). If you could log in to the remote machine and find the repository in the filesystem, you could have a look at all its branches under refs/heads/.

1. Cheat sheet

To delete a local branch, whether tracking or non-tracking, safely: git branch -d <branchname>

To delete a local branch, whether tracking or non-tracking, forcefully: git branch -D <branchname>

To delete a remote-tracking branch: git branch -rd <remote>/<branchname>

To create a new local non-tracking branch: git branch <branchname> [<start-point>]

To create a new local tracking branch (Note that if <start-point> is specified and is a remote-tracking branch like origin/foobar, then the --track flag is automatically included):

git branch --track <branchname> [<start-point]

E.g.: git branch --track hello-kitty origin/hello-kitty

To delete a branch on a remote machine: git push --delete <remote> <branchname>

To delete all remote-tracking branches that are stale, that is, where the corresponding branches on the remote machine no longer exist: git remote prune <remote>

You may have noticed that in some commands, you use <remote>/<branch>, and other commands, <remote> <branch>. E.g.: git branch origin/hello-kitty and git push --delete origin hello-kitty. It may seem arbitrary, but there's a simple way to remember when to use a slash and when to use a space. When you're using a **slash**, you're referring to a **remote-tracking branch on your own machine**, whereas when you're using a **space**, you're actually dealing with a **branch on a remote machine over the network**.