Last modified 12/02/2024

MACS 30500 Deep Dive into Git/GitHub

This tutorial demonstrates how to use Git directly from the terminal in R Workbench, as an alternative to the integrated Git GUI that we have been using so far in the course. Why so? While the Git GUI is convenient and user-friendly, using Git from the terminal provides greater flexibility and control, and it is generally a good skill to acquire beyond this course.

- For Mac users: these instructions should work seamlessly on your local R installation.
- For Windows users: to use Git from the terminal on your local R, you need to install GitBash first (see here for more info)

In this tutorial, we walk through a standard Git workflow commonly followed by researchers:

1. Create a local repository on your computer
2. Initialize it as a Git repository
3. Create a remote repository, and connect it to your local repository

Instead of relying on the Git GUI, we will perform these steps directly from the terminal.

Step 1. Create a local repo

In Workbench go to File > New Project > New Directory > New Project

- Under Directory name write: “css-materials-git”.
- You can create a project as a subdirectory of “lecture” or any other folder
- You have the option to create a git repository right away: *skip this step* for now to complete the tutorial (you can try this feature later).
- Check the box that says “Open in a new session”, then click “Create Project”
- Add one empty R script in that repo

A new R session with the project should open up as an empty folder with only the .Rproj file in it.

Alternative option. *Do not do this for this tutorial*, but here are the steps in case you need to create a folder first, and then track it with Git:

- Create a folder on your computer (e.g., your Desktop or your R Workbench)
- {lace all the files you want Git to track in that folder
- Then add an .Rproj to the folder: open R, go to File > New Project > Existing Directory > Browse and locate the folder you just created > Select it and click Create Project

Step 2. Initialize the folder as a Git repo, then stage-commit-push

We are going to learn the following commands:

- Terminal commands: `pwd, cd, ls` (these are not R or Git commands: they are terminal commands meaning they allow to interact with all programs in your machine)
- Git commands: `git init, git status, git add, git commit, git push`

In Workbench go to Tools > Terminal > New Terminal and type the following commands:

`pwd`
means "print the current working directory." Here it should be something like

```
[nardin@macss-r css-materials-git]$
```

This tells us that we are inside the folder we just created (this should be confirmed also by the R project name at the top right)

Now, close the project and start from the terminal home. We do that to demonstrate how to use the command `cd` to navigate to your desired directory. This is for practice, if you open up a project and you are already in the directory you need to be, there is no need to use `cd`

`cd`
means "change directory." Use it to move around your computer, and go to your desired directory (here we want to go back to the same project directory where we started)
Use `cd` + tab autocomplete or `cd ..` to go up one level

NB: To edit or cancel something you typed in the terminal: go at the end of the line and use backspace. You cannot cancel what you typed by highlighting it and pressing delete)

`ls`
Means "list all files." Use it check what is included in your current directory: it lists all visible content in your directory. For now, it's only the .Rproj file.

Now we are ready to learn some Git commands!

`git init` use it to initialize your project as a Git repository, so Git can track changes to it.

Whenever we use Git via the command line, we need to preface each command with `git` to tell the computer we are trying to get `git` to do something, rather than some other program. The output has a bunch of hints (to help the user with the next steps), but the main info is that it tells us that we have initialized an empty git repository in the given directory

`ls` use it to check again what is included in your current directory; now the directory should still look the same because we have not added any file

`ls -a` the -a flag stays for all content, including hidden content
this shows all files in your directory, including the new hidden `.git` and the hidden files associated with our `.Rproj`.
Flags are command line options that can be added to the commands. Be careful with spacing: `ls-a` is not the same as `ls -a`

`git status` to check the current state of the project
This is the most common git command. The output indicates that we are currently on the main branch and have some unsaved changes. To prepare these changes for a commit, we need to stage them first. This process is the same as what we've been doing using the Git GUI R

`git add .` to add to the Git staging area all new files or files that have been changed.
the dot means everything. You could also stage one file at a time, for example by typing `git add <file>` like so: `git add css-materials-git.Rproj` but often you want to add all files (especially when you create a new repository).

The `git add` command tells Git either:
- That there are new files (new to Git, meaning: untracked by Git) in your current directory and that you want Git to start tracking them
- Or that you have made changes to files already tracked by Git, and you want Git to take note of these changs

`git status` to check again the current state of the project! you should see a list of files in green for which Git says "Changes to be committed."

Optional (do not run now): If you change your mind at this point (e.g. before committing the files) you can "unstage" all files or a single file. Type `git rm --cached <file>` to make Git stop tracking that specific file. To clear your entire Git cache, use the same command but with the -r option for recursive: `git rm -r --cached .`

`git commit -m "first commit"` this is the command to make your first commit!
- Committing is similar to "saving" a file to Git. However, compared to saving, a commit provides more information about our changes.
- The commit message is used to record a short, descriptive, and specific summary of what we did to help us remember later and without having to look at the actual changes; usually, the first commit message is something like "first commit' or "added all files" or "started tracking files"; it does not matter if you use upper or lower case

`git status` to check again the current state. You should see an output message that says: "On branch master nothing to commit, working tree clean"

To review these commands see: https://librarycarpentry.org/lc-git/02-getting-started.html

OK, we have now completed a few steps of this process: initializing the repo, staging files, and committing them. According to the workflow we learned in this course, the next step is to push our files… but where?

At the moment, we have nowhere to push these files! We must first create a GitHub remote repository, connect it to our local repository, and then push the files there. In fact, if we type the command `git push` now, it won't work and Git will tell us to create a remote first!

Step 3. Create a GitHub remote repo and connect it to your local repo

First, we create a new repository on GitHub.

Go to your own GitHub profile: https://github.com/brinasab and execute the following steps:

- Repositories > New or New Repository (green button)
- Owner is you and Repository name type: css-materials-git (you want to assign the exact same name as your local repository)
- Description (optional): type something like "Repo for in-class git practice"
- Leave it public (you can change this setting at any time)
- Do not add a README file and a .gitignore or license (they can be added at any time from your local repo and be then pushed to your remote GitHub repo). If you click on:
    - Add a README: it will just create an empty .md titled README
    - Add a gitignore: it will create a template gitignore file (you can select one for R)
- Click the green "Create repository" button

Second, **connect your newly created GitHub repository to your local repository** on your computer (here on Workbench). These two folders (repos) are completely disconnected and isolated from each other at the moment. We want to link them and synchronize them.

To do so, you now have two options, each with a set of suggestions provided by GitHub. We will do option 2 which is also the most common approach:

1. Create a new repository (if you do not have one already)
2. Push an existing repository: this is what we want here, because we have already created a local repository on R Workbench. Now we only need to connect it with the online repo, and push our local changes there. To do so:
    - Copy all commands provided on Github
    - Go back to Workbench and open the terminal. Paste these commands in there (if you use Windows to copy/paste `cntr+c` and `cntr+v` likely won't work on the terminal. Instead: right click and select Copy)

You can also copy these commands one at a time if you prefer. This is what they do:

```
git remote add origin git@github.com:brinasab/css-materials-git.git
```

- `git remote add` tells to your local Git repository that GitHub is the remote repository; a "remote repository" is another Git repository that contains the same content as your local repository, but is remote aka is not on your computer
- `origin` is the nickname or shortname that we are telling our local machine to use for the long web address that follows (this is the SSH authentication key!). After we enter this command, we can use simply origin to refer to this specific repo in GitHub instead of the long URL (do not use a different name: stick to origin!)
- If you want (not required) now type `git remote -v` to check if the setup is is correct: if you see the same line of code twice it means everything is alright

```
git branch -M main
```

This tells our local repository that GitHub is the main remote repository

```
git push -u origin main
```

This simply pushes our local changes to the GitHub repository!

- The nickname of our remote repository is "origin" and the default local branch name is "main". The `-u` flag tells Git to remember the parameters, so that next time we can simply run `git push` (without the rest) and Git will know what to do
- Side notes:
  - pushing local changes to the Github repository is sometimes referred to as "pushing changes upstream to Github". The word upstream comes from the flag we used earlier in the command `git push -u origin main`. The flag `-u` refers to `-set-upstream`, so when we say "pushing changes upstream", upstread refers to the remote repository.
  - You may be prompted to enter your GitHub username and password; if so, do it
    - If you have made a lot of changes or are adding a very large repository, it might take a bit for Git to push changes to GitHub

Before moving on, type `git status` to check the current state of the project (you want to type this command often to check what you're doing). You should see an output message that says `"Your branch is up to date with origin/main, nothing to commit, working tree clean"` This is your happy message! Refresh your GitHub repo and you should see the updates.

Note on authentication process:

- for this course, we have already authenticated (most of you using SSH credentials)
- one authentication for one machine: if after the class you set up R on your own computer, and you want to use Github, all you have to do is to go over the authentication process again (see instructions on the website)

Step 4. Make changes (add a new file)

Practice: Now follow the instructions below to add a new file in your repo and track it with Git

Add a new `.Rmd` file in your directory. You can do so in two ways (pick one):

1. The traditional way of interacting with R Studio: File > New File > R Markdown > Create Empty Document (so the Rmd is created without pre-filled info) then save it as `git.Rmd` and it should automatically be saved in your “css-materials-git” project.
2. Directly from the terminal, using the command touch: in your terminal, type `touch <filename.extension>` such as `touch git_touch.Rmd`

Then stage-commit-push the new file by typing these commands in the terminal (those in bold are the same from Step 2 of this tutorial, git status is to check after running each command):

```
git status
git add .
git status
git commit -m “your message”
git push
git status
```

Finally, go online and check that your GitHub repo has been updated. Repeat until you feel comfortable using these commands!

Additional Content (beyond class)

Learn novel Git commands: git diff, git log, git reset

Although 90% of the time, you will just use the same few commands (those listed at the top of this page), there are tons of Git commands! I'd like to introduce three commands that I use regularly: `git diff, git log, git reset`

To do so, we are going to modify one existing file. Open the `git.Rmd`, add the following YAML header (modify your name), and SAVE it

```
---
title: "Test Rmd for git practice"
name: "Sabrina Nardin"
output: "github_document"
—
```

Now run the following commands (the new commands are highlighted in bold):

`git status` to check
`git diff` to see changes you have made to this document
`git add .`
`git commit -m "Added YAML"` (or a similar informative message)
`git push`
`git status`

`git log`
If you have forgotten the changes you have made, use this command to look at what you have been doing with your Git repository (in reverse chronological order, with the very latest changes first). Then type `:q` or `wq` to exit (sometimes this does not work or is cumbersome on Workbench but if you have set up your local R with an editor, it works well)

Note on messages: use meaningful commit messages when you make changes; this is especially important when you are working with other people who might not be able to guess what your short cryptic messages mean. It is best practice to write commit messages using the imperative (e.g. 'add index.md' or 'added index.md' rather than 'adding index.md').

Often we change our mind or we make mistakes: we can fix them with Git! The most common command is **`git reset`** to reset the current branch to a specific commit or to unstage changes. See here for more but use this command **with caution**.

To delete pushed commits and their changes:

- `git log` to see all most recent commits

- `git reset --hard HEAD~2` to revert for example the last two commits
- At this moment your local tree differs from the remote because we removed one commit
- `git push -f origin main` to force the remote branch to take this push and remove the previous one
- `git status` to check origin and remote are aligned

To change message of your last unpushed commit:
- `git log` to see all most recent commits
- `git commit –amend`

Fix Git conflicts

A final common situation in which you might run is a so-called "git conflict."

A Git conflict occurs when different branches (for example, here, your local and remote) are not synchronized, that is one file is modified in one repo but not in the other, and Git is not able to determine which one is right (and there is no way Git should know: we have to tell it!).

Git conflicts happen more frequently than you expect, and the **R Git GUI cannot solve them!**

To solve them, you must use the terminal, navigate to the correct directory, and do the following:
1. Type `git status` to see what's going on
2. Manually solve your conflict
    a. Open the file that causes the problem in a text editor, and identify the problem (the areas that are different in one file compared to another are automatically marked by Git so we can find them: look for things like <<<<<<<, =======)
    b. Decide what changed to keep and save
3. Stage the file you modified or stage everything by typing `git add .` or `git add <filename>`
4. If you were in the middle of a merge or rebase operation when the conflict occur, complete it by typing `git merge --continue`
5. Type `git commit -m "Resolved"`
6. Type `git push`

Check this page (the official Github documentation) if you run into a conflict:
https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-using-the-command-line