Practical Coding for Sustainability

Mary Chester-Kadwell
mec31@cam.ac.uk

Exercise 1: Version Control

Contents

License

Introduction

In this exercise we introduce a **simple day-to-day version control workflow** for working on your own computer and syncing with a remote repository, such as GitHub.

Photo by Adam Śmigielski on Unsplash

The content draws on materials from Software Carpentry and the online textbook *Research Software Engineering with Python*. These are very useful resources and go into more detail about certain areas we don't have time for in this CDH course. For our purposes, I have created this exercise using a relevant Digital Humanities example and in such a way that it flows into the other two exercises presented in this course.

Working with Git Locally

Throughout these exercises we will use the command-line to send commands to git. If you are on Mac or Linux open a **Terminal**. On Windows, open **Git Bash for Windows**.

Remember that in examples of git commands, the `$` at the beginning represents the command prompt. Do not type the `$`, just the command after it.

Setting up git

Exercise 1.0

Follow the instructions from Software Carpentry to make sure your git is set up properly: https://swcarpentry.github.io/git-novice/02-setup/index.html

After completing Exercise 1.0 you should have configured:

- A name and email address (the same as your GitHub account, or using a private GitHub email);
- Line endings as appropriate to your operating system;
- A text editor.

Create a local repository

Exercise 1.1

First, create a directory in your home directory (or elsewhere if you prefer) for the repository, and then move into that directory:

```
$ cd ~
```

```
$ mkdir best-practices-for-coding-in-dh
```

```
$ cd best-practices-for-coding-in-dh
```

Then tell git to turn this directory into a repository, i.e. **initialise** the repo.

```
$ git init
```

All subdirectories and files will be included in the new repo. There is no need to create separate repositories nested within the `best-practices-for-coding-in-dh` repository, whether subdirectories are present from the beginning or added later.

If we use `ls` to show the directory's contents, it appears that nothing has changed:

```
$ ls
```

But if we add the `-a` flag to show everything, we can see that git has created a hidden directory called `.git`:

```
$ ls -a
```

And if we list the contents of the special `.git` directory we can see a number of files and subdirectories that contain information used by git:

```
$ ls -a .git
```

These should look something like this:

```
. .. branches config description HEAD hooks info objects refs
```

If you ever delete the `.git` subdirectory, you will lose the project's history.

Check status of the local repository

Exercise 1.2

We can check that everything is set up correctly by asking git to tell us the status of our project:

```
$ git status
```

The output should look something like this:

```
On branch master
No commits yet
```

```
nothing to commit (create/copy files and use "git add" to track)
```

If you are using a different version of git, the exact wording of the output might be slightly different. In essence, the output means we have not committed any changes to the repository.

Racist language in software development

Note that git uses the name 'master' to refer to the default branch[1]. This language is widely deprecated in software development due to its racist connotations and GitHub has changed its default branch name to 'main'. However, git still has 'master' as the default branch name.

If you are using git version 2.28.0 or higher you can change the default branch name that is created when you initialise a repository:

```
$ git config --global init.defaultBranch main
```

Next time you create a new repository the default branch will be 'main'. If you wish, you can delete and recreate your new repository.

Make changes to the local repository

Project organisation

To organise our new project we will follow a template for organising small- and medium-sized data analysis projects (as described in Project Structure of the book *Research Software Engineering with Python*). This template is designed to help organise files and directories and document the progress of a research project. It was developed with bioinformatics in mind, but the principles are general and sound for any research coding project.

The template, unmodified, might not be a perfect fit for every Digital Humanities project, but it is an excellent starting point: "The core guiding principle is simple: Someone unfamiliar with your project should be able to look at your computer files and understand in detail what you did and why."[2]

Here is how we want our project structure to look by the end of the course:

---

[1] We do not cover branches in this course, but you are welcome to read more about branches here: https://merely-useful.tech/py-rse/git-advanced.html All you need to know for now is that a branch is an alternative history for the repo; each branch is a parallel timeline that holds independent changes. If you are collaborating with others you might work on separate branches to avoid interfering with one another, and later merge them into the 'main' branch. Generally, as an individual, you would work on the 'main' default branch unless you create any new branches.

[2] Noble, William Stafford. 2009. "A Quick Guide to Organizing Computational Biology Projects." *PLoS Computational Biology* 5 (7): e1000424. https://doi.org/10.1371/journal.pcbi.1000424.

```
best-practices-for-coding-in-dh/
├── .gitignore
├── CITATION.md
├── CONDUCT.md
├── CONTRIBUTING.md
├── LICENSE.md
├── README.md
├── requirements.txt
├── bin
│   ├── named_entity_recognition.py
│   └── ...
├── data
│   ├── README.md
│   └── henslow
│       ├── letters_1.xml
│       └── ...
└── results
    ├── letters_1_ents.json
    ├── letters_1_viz.html
    └── ...
```

Exercise 1.3

Create the following empty files in the repository:

- `CITATION.md`
- `CONDUCT.md`
- `CONTRIBUTING.md`
- `LICENSE.md`
- `README.md`

You can do this any way you wish: in your text editor, in your file explorer or on the command-line using the `touch` command:

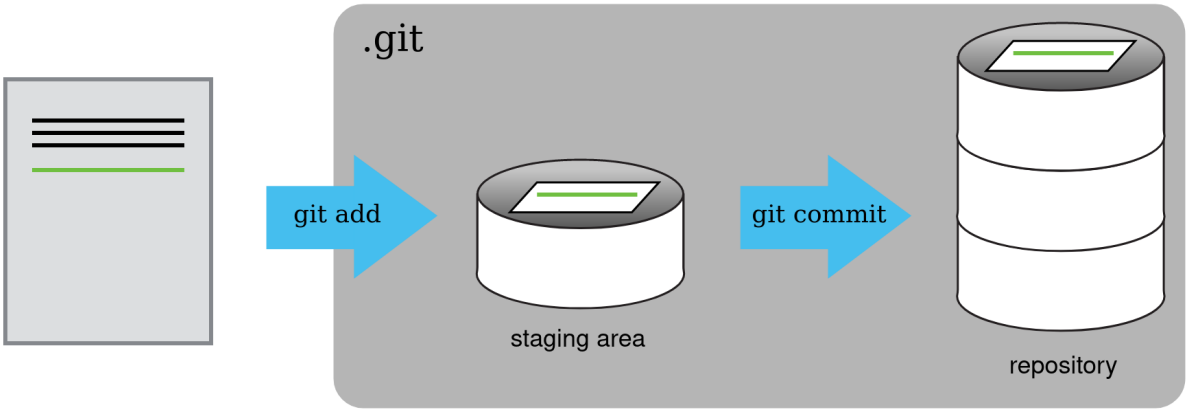
```
$ touch CITATION.md CONDUCT.md CONTRIBUTING.md LICENSE.md README.md
```

Now check that the files are present as expected:

```
$ ls
```

Track changed files

At this point what we have are **untracked files**. We have not yet told git that we want it to pay attention to these files. In order to track new files and changes to existing files we need to **add** them to the **staging area** with command **`git add`**.



The staging area is where we accumulate all the changes we want to make up the next **commit** to the repository (more on commits below). As you make changes, you should keep adding files to staging as you go. We will do that next.

Exercise 1.4

Before you start, make sure you are at the root of the repository's directory.

```
$ pwd
```

should output something like:

```
/home/mary/best-practices-for-coding-in-dh
```

Now check the status of the repository again:

```
$ git status
```

This time the output should look like this:

```
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    CITATION.md
    CONDUCT.md
    CONTRIBUTING.md
```

```
    LICENSE.md
    README.md
nothing added to commit but untracked files present (use "git
add" to track)
```

It lists the untracked files that we have created.

Now we add one of the files and check the status again:

```
$ git add README.md
```

```
$ git status
```

Output:

```
On branch main
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    CITATION.md
    CONDUCT.md
    CONTRIBUTING.md
    LICENSE.md
```

By adding just one file, only one file is now tracked. The rest are still untracked.

When you have multiple files to add at once you can simply list them:

```
$ git add CITATION.md CONDUCT.md CONTRIBUTING.md LICENSE.md
```

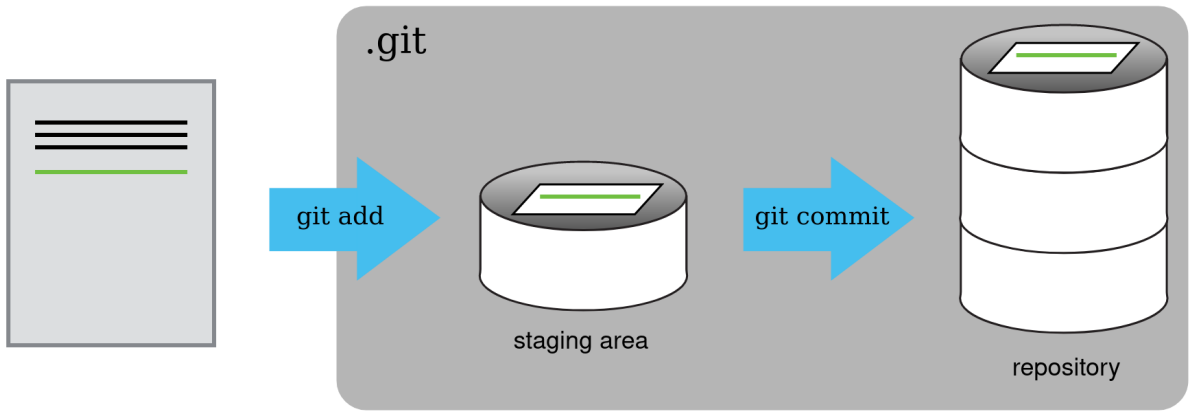
Alternatively, you can add *all* untracked files at once:

```
$ git add .
```

The `git add .` command is commonly used, but by adding all untracked files indiscriminately you may accidentally track changes you didn't mean to track. As a result, it is generally recommended to add files individually.

Commit all staged files

We are now ready to **commit** our tracked changes to the repository. This will create an individual snapshot of the changes and store a copy of the snapshot inside the special `.git` directory. Commits are also called **revisions**.



What should a commit contain?

Ideally, each revision you make to a repository should be relatively small and comprise a logical group of changes that cluster around a specific intention. You should be able to write a **commit message** in just one line that succinctly describes the purpose of a change (more on commit messages below). If you find it is very hard to distill the essence of your change, your commit may be too large and unfocused and might need splitting into multiple commits.

When you look back on your repository's **log** of commit messages you want to see a clear and atomic evolution. Your repository's log is a form of *documentation*, but it is also an *'undo'* list. If you make a mistake you can **revert** (undo) a commit, but if your commits are too large you may end up undoing a load of changes you wanted to make.

So as you are working on your code, think about how to group your changes into small and self-contained commits. This is a good practice to get into from the very first commit of your repository. Eventually, it becomes an ingrained habit that comes naturally (really!).

Exercise 1.5

The files we have added to the staging area are logically related: they are all standard documentation files fundamental to the structure of our project.

Now we will commit them to the repository:

```
$ git commit -m 'Add standard documentation'
```

Output:

```
[main (root-commit) a68d9c6] Add standard documentation
 5 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 CITATION.md
 create mode 100644 CONDUCT.md
 create mode 100644 CONTRIBUTING.md
 create mode 100644 LICENSE.md
 create mode 100644 README.md
```

We use the `-m` flag (for "message") to record a short, descriptive, and specific comment that will help us remember later on what we did and why. If we just run `git commit` without the `-m` option, git will launch whatever editor we configured as `core.editor` so that we can write a longer message[3].

Now we can check the status of the repository:

```
$ git status
```

Output:

```
On branch main
nothing to commit, working directory clean
```

We find that everything is 'clean' and there are no more changes to add.

How to write helpful commit messages

Good commit messages start with a brief (<50 characters) statement about the changes made in the commit. Generally, the message should complete the sentence "**If applied, this commit will**". If you want to go into more detail, add a blank line between the summary line and your additional notes. Use this additional space to explain why you made changes and/or what their impact will be[4].

Often, the main audience for your commit messages is *yourself*, in the future. Depending on your project, the audience may also be your collaborators, or other researchers who might inspect or reuse your code.

Commit messages should not just recapitulate *what* you did. The contents of the commit (i.e. the changes) are *what* you did. Commit messages are an opportunity to communicate *why* you did something for the benefit of future you or others.

[3] This paragraph is taken almost verbatim from Tracking Changes.
[4] This paragraph is taken verbatim from Tracking Changes.

Examples of bad commit messages

Bad message	Reason why it is bad
Fixed bug	Too unspecific. Which bug? What was the problem? Why did you fix it?
Polishing	Lazy! The message gives no idea what the commit is for at all.
Rework the edit function logic + refactor the dependent functions so that they are neater + use the refactored functions better in the places where they are used	Too verbose. Doesn't explain why the logic was reworked. Such a long message may also suggest the commit has too much unrelated contents and should be split into smaller commits.
The edit function has been removed	Just tells us *what* is in the commit. Doesn't say *why* it was done. Doesn't start with a verb to show an action.

Examples of good commit messages

Good message	Reason why it is good
Fix bug causing endless loop in printing results	Explains which bug it is and shows why it needs fixing by explaining the consequences (infinite loops are bad!).
Tidy notebook layout to improve look	Specifically states what sort of "polishing" is taking place and why (cosmetic changes).
Rework edit function logic for speed	Explains why the logic function was reworked (speed). The commit has been reduced to just this change; the other unrelated changes (see above) have been split into separate commits.
Remove edit function not used anywhere	Helpful reason provided why the function has been removed.

Tip: Start every commit message with a verb in the imperative, i.e. spoken as if you are giving a command.

It pays large dividends to get into good habits early on because your codebase will develop, over time, a comprehensive and well-documented history of progress and evolution in

thinking. Although it is tempting to write perfunctory commit messages just to get them out of the way, taking just a minute to craft a helpful commit message is best practice.

If you are interested, you can read more about How to Write a Commit Message in detail.

Inspect history of commits

Every repository includes a list of every commit (for every branch). This is often referred to as the repository **history** or **log**. It is helpful to think of your work as building up this history, commit by commit.

It is good practice to review your log regularly. Once you get into this habit, you will quickly realise the benefit of committing small, logical commits with clear messages.

Exercise 1.6

Let's inspect the history of your repository now:

```
$ git log
```

The log should look something like this:

```
commit a68d9c689ef4eff7826732f3892e32bc6f806695 (HEAD -> main)
Author: Mary Chester-Kadwell
<mchesterkadwell@users.noreply.github.com>
Date:   Thu Apr 8 15:58:41 2021 +0100

     Add standard documentation
```

You have only made one commit, so there is only one entry in the log. Note that the commit has a long unique ID number. We have seen this number before when we first made the commit (**Exercise 1.5**) except there it was shortened to the first seven characters: 'a68d9c6'. Typically, you will use the shortened ID to refer to individual commits.

If you have time, see Tracking Changes for more detail on `git log` and using `git diff` to inspect differences between your changes and the most recent commit.

Prevent files being tracked

Sometimes we don't want a file or files to be tracked by version control, for example, a local configuration file that contains configuration only relevant to your environment, or that contains passwords. Other examples might be logs, IDE project files, Jupyter Notebook

checkpoints and the virtual environment you are using. There are lots of reasons why you might want git to ignore certain files.

In this case we can add a file called **`.gitignore`.** Inside this file we list any specific file, file extension, directory, or pattern that matches files we want to ignore.

Often an IDE might suggest you add a `.gitignore` and even suggest some common things you might want to ignore in it. But if you are using a plain text editor then you may need to create the `.gitignore` by hand, as we will do now.

Exercise 1.7

Create a `.gitignore` file at the root of the repository.

Open the file, add the following lines, and save the file:

```
# Virtual environment
venv/

# Jupyter Notebook
.ipynb_checkpoints

# Python bytecode
__pycache__
```

Now add and commit the `.gitignore` to the repository with a suitable commit message. Inspect the log to see both the commits you have made. If you have forgotten, Exercises 1.4, 1.5 and 1.6 cover these steps.

Syncing Your Local Repository with a Remote Repository

We are going to sync your local repository to GitHub. By doing this you create a copy of the repository accessible online called a **remote repository**. This copy can act as a personal backup, a form of publication and/or an opportunity to share and collaborate your code with others.

Create a remote and configure local repository to use it

Exercise 1.8

Follow the instructions from Software Carpentry to create a remote repository and link it to your local repository: https://swcarpentry.github.io/git-novice/07-github/index.html

up to, but not including, the heading "The '-u' Flag".

Important: Make sure that you call your repository "best-practices-in-coding-for-dh" (instead of "planets" as in the tutorial). Your repository should end up called something like this:

`https://github.com/mchesterkadwell/best-practices-in-coding-for-dh`

with your GitHub account name instead of mine (`mchesterkadwell`).

After completing Exercise 1.8 you should have:

- Created a repository "best-practices-in-coding-for-dh" in your GitHub account;
- Added the GitHub repository as a remote on your local repository;
- Pushed your local repository to the GitHub repository.

Push changes to the remote repository

When you make commits to your local repository, they are not automatically uploaded to the remote copy. You have to do this manually.

Exercise 1.9

Create a new file in the `best-practices-in-coding-for-dh` repository named **`requirements.txt`**.

We will leave it empty for now. Add and commit the file to the local repository with the message ‘Add list of dependencies’.

Once you are satisfied that the commit has been added to the local repository history, push the changes to the remote repository:

```
$ git push origin main
```

If everything is successful, you should see something like this:

```
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 277 bytes | 277.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local
object.
To
https://github.com/mchesterkadwell/best-practices-in-coding-for-d
h.git
    6e7b5db..25dc35f  main -> main
```

Reusing Code

Cloning a remote repository

For **Exercises 2** and **3** we will be using some existing code to refactor into a command-line or GUI program. This presents us with an excellent opportunity to learn more about how to re-use others' code. The existing code is available in a repository on GitHub and you will make a copy of it on your local machine, in other words, **clone** the repository.

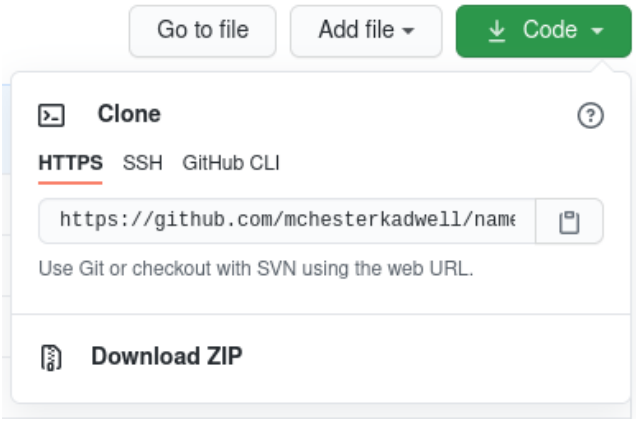
Photo by Zac Harris on Unsplash

This is a task you might do anytime you find some open-source code on GitHub and want to inspect or reuse it. You may also clone a repository that you are working on with other people so that you are all collaborating on the same codebase.

Exercise 1.10

Go to https://github.com/mchesterkadwell/named-entity-recognition .

Click on the green 'Code' button and inspect the options for creating a copy of the code.



You could just download a ZIP file with the contents of the repository, but then you would

lose all the git history and the files would retain no relationship with the original repository on GitHub.

On the HTTPS tab click the clipboard icon to copy the web URL.

Open a Terminal or Git Bash for Windows and navigate to a directory where you would like to store the cloned repository. Then enter the command `git clone` followed by pasting the URL you just copied from GitHub.

```
$ git clone
https://github.com/mchesterkadwell/named-entity-recognition.git
```

After a short wait the repository will download and appear on your local machine under a directory called ‘named-entity-recognition’. Inspect the directory to view its contents.

```
$ cd named-entity-recognition
```

```
$ ls -a1
```

The output will look something like this (the exact order might be different):

```
.
..
0-introduction-to-python-and-text.ipynb
1-basic-text-mining-concepts.ipynb
2-named-entity-recognition-of-henslow-data.ipynb
3-principles-of-machine-learning-for-named-entities.ipynb
4-updating-the-model-on-henslow-data.ipynb
5-linking-named-entities.ipynb
assets
conftest.py
data
doccano
.git
.gitignore
html_notebooks
LICENSE
output
README.md
requirements.txt
tests
```

Forking a remote repository

Cloning a repository is fine when you just want to reuse code without changing it yourself, or if you are one of the collaborators on a repository and intend to work on it. Sometimes, though, you might want to change the code for your own purposes and yet keep those changes separately from the original repository. An example is if you are following a Jupyter Notebook tutorial and want to add your own notes or try it with your own data.

Photo by Rasmus Gundorff Sæderup on Unsplash

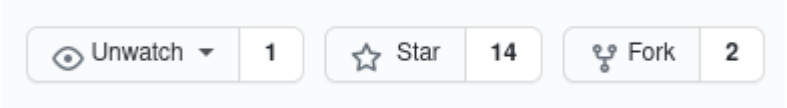
In this case you may want to **fork** the repository before you clone it. Forking makes a copy of the original repository, keeping a link to the original (on GitHub), but giving the new repository a new owner and new history. When you commit to your forked repository, the commits are only visible in your fork, and the original repository owner cannot control your fork[5].

It is important to check the license of the original repository before forking it so you know what you can and cannot do with the code.

Exercise 1.11

Go again to https://github.com/mchesterkadwell/named-entity-recognition .

In the top right-hand corner click on the 'fork' button:



GitHub will now 'photocopy' the original repository and create it as a new repository under your own account.

[5] You can, however, pull in changes from the original repository, say, if there is a security update or new function developed. You can also request that changes you made in your repository are pulled into the original ('upstream') repository.

The URL should look like this:

`https://github.com/youraccount/named-entity-recognition.git`

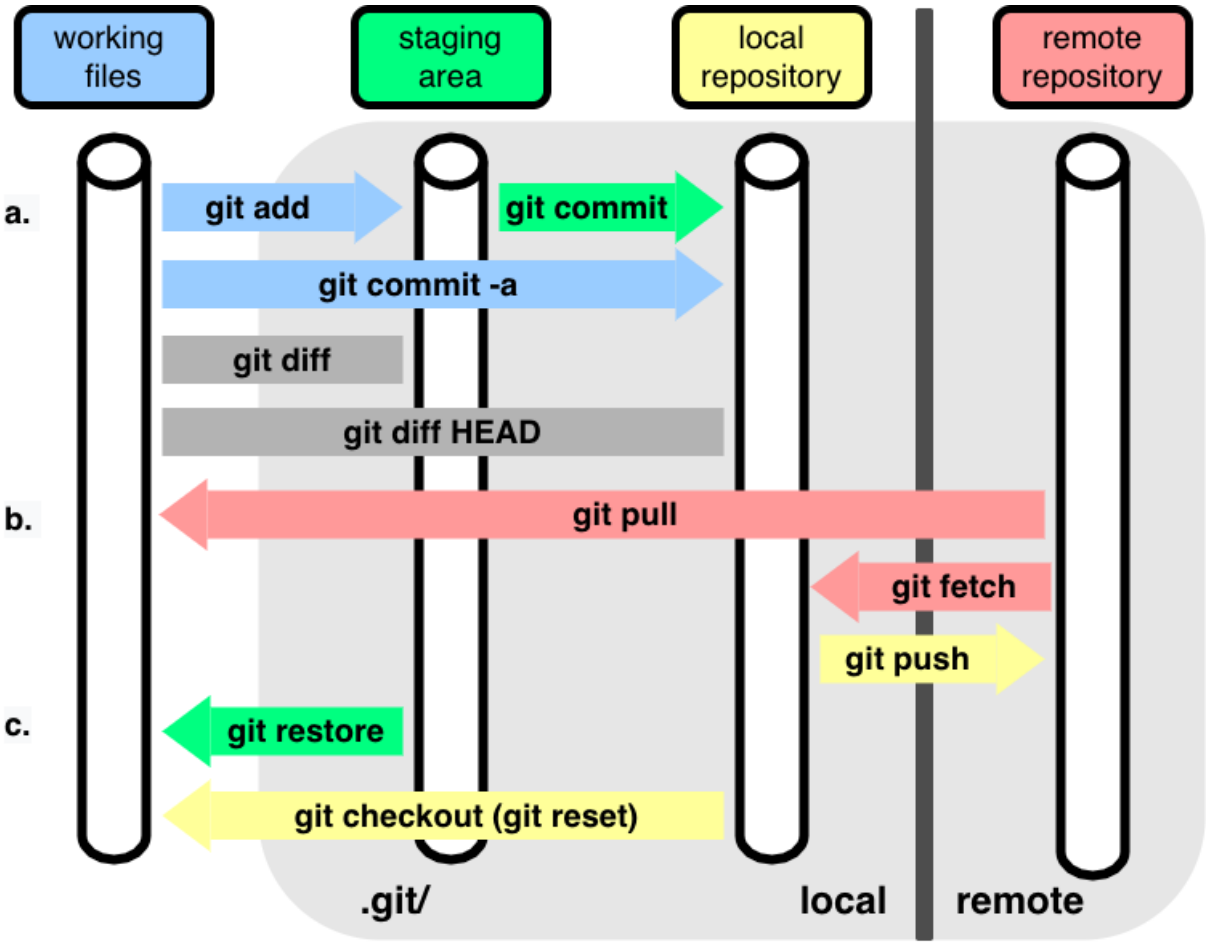
where ‘youraccount’ is your account name.

Now you could clone your new forked repository, the same way you did in **Exercise 1.9**, then make changes and commit in the usual way. (You don’t have to do this now. Just know that the process is the same.)

If you have time, you can follow the tutorial in the GitHub documentation on forking and syncing their example repository:
https://docs.github.com/en/github/getting-started-with-github/fork-a-repo

Further Resources

There is much more to learn about git than we can cover in this short course. After many years of using git I am still discovering new things about it all the time!



Commands included in basic Git workflows. Source

A useful approach to learning more about git is to first establish your simple day-to-day workflow. Then if you need to solve a particular problem you come across, look up the solution and spend a little extra time understanding the answer. This way you build up a practical knowledge base and avoid being overwhelmed by everything git has to offer.

If you do have some extra time and would like to do some further study about git I can recommend the following resources:

Software Carpentry: Version Control with Git
The full tutorial goes into much more detail than we have time for in this course.

Introduction to Git for Data Science (DataCamp)
Like all DataCamp tutorials this is an interactive hands-on introduction.

Head First Git (2022, early release) by Raju Gandhi (O'Reilly Learning, Access using your Raven account)
The 'Head First' series is well known for being engaging and funny. This book goes through an extended tutorial, explaining the concepts behind what is happening and not just giving you copy-and-paste commands to follow.

Complete Git Guide: Understand and Master Git and GitHub by Bogdan Stashchuk (O'Reilly Learning, Access using your Raven account)
If you like to learn with video tutorials, then this comprehensive series will provide you with an opportunity to learn about any aspect of git in some technical detail.