An introduction to Git and GitHub

Prof. Andrew C.R. Martin, University College London

November, 2018

This self-paced tutorial will take you through the basic use of Git and GitHub. These are systems that allow you to maintain code and work that you are doing, track changes, recover old versions and collaborate with other people. There are many tutorials available on the Web or in books. I use **`this style`** for commands you type, and `this style` for filenames. In places I use ***`this style`*** or **`THIS STYLE`** for things that should be substituted with an appropriate filename or word.

Contents

1 Introduction — what are Git and GitHub?

How many times have you written a document — an essay perhaps — where you think you have got to your final version and called it `EssayFinal`, only to read through it again and spot some errors, so you rename it `EssayFinal2`. Perhaps it's a dissertation and you give it to your supervisor who has some comments so you end up with `EssayFinalRevised`. Perhaps you now realise it's too long so delete some stuff and create `EssayFinalRevisedCut`. Then you read it again and realise that you really need to add something back from an earlier version as it no longer makes sense — Oh! Wait...Did you keep that earlier version?

Git is designed to deal with exactly this sort of situation. Unfortunately it doesn't work that well with 'binary' files like the `.doc` or `.docx` files that Word uses, but for plain text files of the sort you use for writing programs and scripts, or files that you use with LaTeX (the typesetting system used to write these tutorials), it removes these

sorts of problems completely. What's more, you can synchronize git with GitHub on the web giving you a backup of what you have done, allowing you to collaborate with others and to publish what you have done to share it with the outside world. Some journals (such as F1000) are even requiring that people who write papers describing code deposit it in GitHub. If you write code to do your research, you can ensure that you tag your code in GitHub with a version number so that when you publish your research you can tie it to a specific version of the code allowing you (and others) to recover exactly the version of code that was used to create your results.

Git and GitHub are separate things, but linked. Git is the software that runs on your computer and manages your files. You don't need to use it with GitHub. GitHub is an online platform that allows you to synchronise your local Git reporitory onto the web. You can also use GitHub to browse other people's repositories and download code or documents without ever using Git.

2 Installing Git

2.1 If you are using Linux...

You probably have Git installed already — try typing
`git --version`
at the command line and see if it returns a version number. If it says the command is not found, use your package manager to install it. If you are using a RedHat style distribution (RedHat, CentOS or Fedora), type
`sudo yum -y install git`
or on the newer distributions
`sudo dnf -y install git`
If you are using a Debian-based distribution such as Ubuntu, then type
`sudo apt-get git`

2.2 If you are using a Mac...

Go to `git-scm.com/download/mac` and download and install the Git package. The download should start automatically.

2.3 If you are using Windows...

Go to `gitforwindows.org` and download and install the Git-bash package. You have probably already done this for the command line tutorial.

3 Getting started on GitHub

The first thing you need to do is create yourself a GitHub account. Since you are a student, you can create yourself a student account. GitHub is free to use providing you are happy to make all your repositories public. If you want private repositories you have to pay unless you are a student or an academic in which case you can have free private accounts.

Task:
Go to `www.github.com` and create an account. You probably want to select a fairly memorable (and sensible!) user name — ideally this is something that you will continue to use throughout your career.

As explained in the Introduction, one of the main things you will want to do with GitHub is synchronize your local repository with it. GitHub requires you to log in with your username and password. This would be rather a nuisance if everytime you wanted to synchronize your local Git repository with GitHub you had to specify your username and password. Fortunately a mechanism is provided to allow you to avoid this[1]. First you need to create yourself what is known as an SSH key:

1: Type the following:

```
cd
mkdir .ssh
cd .ssh
ssh-keygen
```

`ssh-keygen` will probably ask you where you want to save the key. Simply press the Return key to accept the default.

`ssh-keygen` will then ask you to enter a passphrase. Simply press the Return key so that no passphrase is generated. You will then have to press the Return key a second time to confirm your (empty) passphrase.

Now if you type **`ls`**, you should find that you have two files called `id_rsa` (your private key) and `id_rsa.pub` (your public key). Display your public key:

2: Type the following:

```
cat id_rsa.pub
```

[1]This relies on using a public/private key pair for authentication. This is the same sort of thing that is done for encrypting your bank details when you send them over the web. Take a look at `en.wikipedia.org/wiki/Public-key_cryptography` if you want to know more

Task:

- In GitHub, click the menu item at the top right (it may be a picture of you if you have added a profile picture) and choose **Settings** from the menu.
- Select **SSH and GPG keys** from the menu on the left.
- Click **New SSH key** towards the top right
- Enter a title for your SSH key in the box — this can be anything you like, but probably something that identifies the computer you are using (e.g. Andrew's Windows laptop)
- Cut and paste the whole of the public key that you displayed a moment ago (the content of `id_rsa.pub`) into the **Key** box.
- If you are using a Mac, use:

  ```
  pbcopy < ~/.ssh/id_rsa.pub
  ```

  to get the public key into your clipboard so you can paste it into GitHub.
- Click **Add SSH key** at the bottom of this section of the web page.

4 Configuring Git

You need to type two lines to configure Git and link it to GitHub.

In the following commands:

- Replace **`USERNAME`** with the username that you created on GitHub
- Replace **`EMAIL`** with your email address

3: Type the following:

```
git config --global user.name "USERNAME"
git config --global user.email "EMAIL"
```

5 Creating a Git repository

When you use Git, you create one repository for each program or project that you work on. I suggest that you create all your Git repositories under a single directory called `git`. Let's start by creating this directory:

4: Type the following:

```
mkdir ~/git
```

Command	Effect
Esc	Enter command mode
i	Insert text at cursor
A	Move to end of line and insert text
r	Replace a single character under the cursor
R	Overwrite characters
:w	Write the file
:q	Quit the editor
:wq	Write the file and quit the editor

Table 1: The most useful **vim** commands.

Now you need to create a directory for your project repository and enter that directory. For this tutorial we will call it `GitExercise`:

5: Type the following:

```
cd ~/git
mkdir GitExercise
cd GitExercise
```

Note that it is a good idea to avoid spaces in directory (and file) names since you will have to put the name in inverted commas or escape each of the spaces with a backslash which can be a real nuisance!

We now need to tell Git that this is a Git repository:

6: Type the following:

```
git init
```

6 Creating and editing files

A text editor is a program that allows you to create and edit a simple plain text file. Things like Word allow you to embed information about fonts, font-size, colour, bold, italics, etc. A text editor simply deals with the plain characters — some text editors may display text in colour while you are working on it (for example to highlight syntax while writing a computer program) but this information is not saved. Under Windows, you can use **NotePad** (ensure you save as a 'plain ASCII text file'), but other freely-available editors include **Atom** and **vim** (also known as **vi**). On the Mac, choices include **TextEdit** or **vim**.

vim is built into the git-bash environment, but is rather old-fashioned and not easy to use — its advantage is it is available in every unix-like environment and lots of prgrammers really like it because it is very fast to start and pretty powerful. It is also the default editor that Git will use if it expects you to provide information about a change you have made[2]. **vim** has a 'command mode' and a 'text-entry mode'. When **vim** starts, it will be in command mode. To add text, move around using the arrow keys. To start entering text at the cursor, press the **i** key (for insert) and start typing.

[2]You can change the editor that Git uses by giving the command
git config --global core.editor "*editor*"
replacing ***editor*** with the editor you wish to use.

When you are done press the **Esc** key to return to command mode. To exit the editor, press **Esc** to ensure you are in command mode, then type **:wq** Table 1 summarizes the most useful commands.

7 Adding files to your Git repository

Let's start by creating a file to contain some information about proteins.

Task:
Use a text editor to create a file in your ~/git/GitExercise/ directory called `proteins.txt` containing the following text:

```
Lysozyme         Chicken LYSC_CHICK P00698
Lysozyme         Human   LYSC_HUMAN P61626
Hemoglobin-alpha Human   HBA_HUMAN  P69905
Hemoglobin-beta  Human   HBB_HUMAN  P68871
```

Exit your editor and tell Git that you want to track this file:

7: Type the following:

```
git add proteins.txt
```

You only need to do this when you have a new file that you want to track with Git.

Having specified that the file should be tracked, we need to tell Git that we have made changes to the file that we want it to record:

8: Type the following:

```
git commit -a -m "Initial version"
```

Strictly in this case the **-a** isn't needed, but it will be whenever you do this in future so we will put it in[3]. The **-m** specifies the text that follows in double-inverted commas is a comment (or **m**essage) briefly explaining what changes were made in this version.

If you forget to supply a message with **-m** then you will enter the (**vim**) where you can enter text (See the notes above).

8 Finding out what's happening

To find out whether there are any files that are not being tracked by Git or whether there are files that are being tracked that haven't been committed:

9: Type the following:

```
git status
```

[3]The **-a** tells Git to add the changes in this file to those that are being tracked by **commit**. You already did that with **git add** which is always needed for a new file.

Having just committed your changes, you should receive the message:

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

Let's create another file that isn't being tracked just to see what happens:

10: Type the following:

```
touch foo.txt
git status
```

The **touch** command is used to set the date and time on a file to the current date and time. If the file doesn't exist, it creates it, so we are just using this to create an empty file.

This time the output from **git status** should look something like:

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        foo.txt

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

Git warns you that there are files present that aren't being tracked. Since we only created this file as a test, let's delete it again:

11: Type the following:

```
rm foo.txt
```

We can also list what commits have been made:

12: Type the following:

```
git log
```

This will list all the commits you have made with the comments that you specified with **-m**. The `commit` line in the output will be followed by a random string of characters — something like `b5526a8ddb40925e01620e751ecc97b735464444`. Don't worry about this for now; you will find out why it is useful in a minute.

9 Making changes and tracking them

Task:
Edit your `proteins.txt` file by adding a couple of extra lines to the end of the file. For example:

```
PFKM            Human   PFKAM_HUMAN P08237
Pfkm            Mouse   PFKAM_MOUSE P47857
```

Now if we check the status we will be told that one of the files has been modified:

13: Type the following:

```
git status
```

You should see something like:

```
On branch 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:   proteins.txt

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

We now have to commit the changes so that Git keeps track of them.

14: Type the following:

```
git commit -a -m "Added information on PFKM"
git log
```

Looking at the output from **`git log`** you should now see that there is a log message for your changes.

10 Synchronizing your local Git repository with GitHub

To synchronize your local repository with GitHub, proceed as follows:

- Go to the `GitHub.com` web site and make sure you are logged in,
- In the top right corner, you should find a + sign. Click this and then click **New repository**,
- Enter the name of your repository (`GitExercise`),
- Add a description of your repository — something like "Git Exercise",
- Do *not* select the option to "Initialize this repository with a README",
- Click the button to create the repository.

These steps have created an empty repository on GitHub; you now need to synchronize your local repository with this.

Replace **`USERNAME`** with your GitHub username in the following.

15: Type the following:

```
git remote add origin git@github.com:USERNAME/GitExercise.git
git push -u origin master
```

If you make a mistake in this command (such as not using the corrent username or repository name), you can delete the remote repository and start again with **`git remote remove origin`**

Task:
Refresh the GitHub web page — you should now find that your `proteins.txt` file is listed on the page. You should also see that there have been 2 commits reflecting what you did on the command line.

From now on, after you make and commit changes, all you need to do is **`git push`** (without the **`-u origin master`**) to 'push' your changes onto GitHub.

11 Deleting and renaming files

Normally you would delete a file with **`rm`** and rename or move a file using **`mv`**. However, we also need to tell Git that we have deleted, renamed or moved a file. To do that we have to remember to delete, rename or move the file via Git.

Do not do this now, but if, for example, we wished to rename our `proteins.txt` file to `proteininfo.txt` we would do:

```
git mv proteins.txt proteininfo.txt
```

Similarly to delete a file and stop Git from tracking it we would do:

```
git rm proteins.txt
```

12 Undoing changes

12.1 Before you commit...

Let's add something else to the `proteins.txt` file, but something that was clearly a mistake.

Task:
Edit the `proteins.txt` file and add the following line to the end of the file:

```
This is clearly a mistake!
```

If you now type **`git status`** you will be told that the file has been changed, but that you haven't committed the changes yet. At this stage, you can go back to the version without the changes:

16: Type the following:

```
git checkout -- proteins.txt
cat proteins.txt
```

As you will see, the file has reverted back to what it was before you added the lines.

If you have modified more than one file and want to reset everything to the last commit, you can also do
`git reset --hard`
to reset all the files.

12.2 After you commit...

Suppose however that you have committed a change and now want to undo that change and revert to the previous version. You will recall that when you use **`git log`** you are shown the comments, but also shown lines saying `commit` with some apparently random set of letters and numbers. This is an identifier for a particular commit. We can use **`git revert`** with one of these identifiers to remove the changhed that happened in that commit. In practice, you only need about the first 6 or 7 characters of the commit identifier to identify a particular commit uniquely.

Again add something to the `proteins.txt` file that we will clearly wish to undo.

Task:
Edit the `proteins.txt` file and add the following line to the end of the file:

```
This is clearly another mistake!
```

Now commit the change and look at the Git log:

17: Type the following:

```
git commit -a -m "I added a mistake on purpose"
git log
```

Task:
Record the first 6 or 7 characters of most recent `commit` message (at the top of the log) which is associated with the erroneous commit.

For example, my Git log looks something like:

```
commit adbd940940ba3f1584d3bd77c4b048e2b0fcda3f
Author: AndrewCRMartin <andrew@bioinf.org.uk>
Date:   Thu Sep 28 12:39:37 2017 +0100

    I added a mistake on purpose

commit 54f0e41a1c772dbde9002ab1252c634c6296ec77
Author: AndrewCRMartin <andrew@bioinf.org.uk>
```

```
Date:   Thu Sep 28 11:16:08 2017 +0100

    Added information on PFKM
```

I can now return to the previous version by 'reverting' the latest commit which contained the error:

18: Type the following:

```
git revert adbd940
```

(when you do this, replace **adbd940** with whatever is appropriate from your Git log). *Note that the commit number that you supply is the one that you want to undo, not the one you want to get back to!*

The screen will change to a message saying:

```
Revert "I added a mistake on purpose"

This reverts commit adbd940940ba3f1584d3bd77c4b048e2b0fcda3f.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Changes to be committed:
#       modified:   proteins.txt
#
```

You have entered the **vim** text editor (see Section 6 and Table 1.) where you can add other comments about the change, but we will simply accept the default message by pressing the **Esc** key and typing the characters: **:wq** (colon, 'w', and 'q')[4]

If you now look at the `proteins.txt` file, you should find that the changes have gone and we have returned to the file as it was.

Now push your changes to GitHub — you committed a change and then reverted it which is effectively another commit. Git tracks everything, so even your mistake is tracked and can be got back again if needed:

19: Type the following:

```
git push
```

NOTE! `git revert` can be used to undo any commit — not just the most recent one. You simply specify the relevant commit identifier.

12.3 Being more selective

You can also undo a change to an individual file rather than all the files that were changed in a commit. We won't look at an example here, but just explain what you need to do.

[4]Pressing the **Esc** key ensures you are in command mode and **:wq** means write the file and quit.

To obtain an older version of a particular file, you can use the command:
`git checkout` *`commitid -- filename`*
where ***`commitid`*** is the commit identifier of the version that we want to keep (i.e. the commit containing the last good version of the file) and ***`filename`*** is the filename of the file you want to obtain. Having done this you would also need to commit the changes in the usual way with **`git commit`**.

12.4 Retrieving an old version

To get back to an older version in the log history without actually undoing the changes between that version and the current version, we use the **`git checkout`** ***`commitid`*** command where ***`commitid`*** is the commit identifier of the version that you want to go back to. You can then create a new branch starting from that point (see Section 13, below).

To return to the latest commit, simply type **`git checkout master`**.

Task:
Give it a try. Go back to the original commit, take a look at the `proteins.txt` file and then return to the latest version.

13 Branching

Often we find that we have a version of a program (or script) that we are happy with, but we want to add some new feature(s). Whenever we make changes we risk breaking the existing version. Git allows us to create experimental 'branches' where we can develop new features before merging them back into the master repository.

This is also useful when we have multiple people working on a project. Each person creates a branch for the feature or bug-fix they are working on and when they are finished they merge it back into the master branch.

Suppose we want to modify our proteins file to include the length of each protein. Adding that extra field to the file might well break some program that relied on the file, so we will do it in a new branch called 'protlength'.

We create a new branch using
`git checkout -b` *`branchname`*
(where **`-b`** tells checkout to create a new branch):

20: Type the following:

```
git checkout -b protlength
```

This has created a new branch and switched to using that branch. You can confirm which branch you are currently using:

21: Type the following:

```
git branch
```

The 'master' branch is the default branch that we were using previously.

If we are using GitHub, we need to tell Git to link this new branch to GitHub. We do this with the command[5]:

22: Type the following:

```
git push -u origin protlength
```

We can now make our experimental changes to `proteins.txt`.

Task:
Edit `proteins.txt` to add the protein lengths:

```
Lysozyme          Chicken LYSC_CHICK  P00698 147
Lysozyme          Human   LYSC_HUMAN  P61626 148
Hemoglobin-alpha  Human   HBA_HUMAN   P69905 142
Hemoglobin-beta   Human   HBB_HUMAN   P68871 147
PFKM              Human   PFKAM_HUMAN P08237 780
Pfkm              Mouse   PFKAM_MOUSE P47857 780
```

Once we have made our changes, we commit them and push them to GitHub:

23: Type the following:

```
git commit -a -m "Added protein lengths"
git push
```

If we wish to return to the 'master' branch we can do so with:

24: Type the following:

```
git checkout master
```

Now take a look at `proteins.txt` — you will see that it no longer has the protein lengths. Return to the 'protlength' branch:

25: Type the following:

```
git checkout protlength
```

and verify that `proteins.txt` now has the protein length information.

13.1 Checking differences between branches

Just as we can use **`diff`** to look at the differences between files, we can use **`git diff`** to look at differences between branches or commits:

[5]You may remember that when you first synchronized with GitHub you had to type **`git push -u origin master`**. That was telling Git to link the 'master' branch with a server called 'origin' which is the GitHub web site (we specified that in Section 10 with **`git remote add origin`**; now we are synchronizing a different branch with 'origin'.

26: Type the following:

```
git diff master
```

This will show us the differences between the current ('protlength') branch and the 'master' branch.

If we use **`git diff`** by itself (with no following parameter), it will show us differences from the most recent commit.

We can also use **`git diff`** with a commit identifier to see the differences between the current version and a specified commit or with two commit identifiers to see the differences between two specific commits.

In any of these cases, if you are only interested in a particular file, just add the filename on the end as well.

13.2 Merging branches

Let's suppose we are now happy with our changes in the 'protlength' branch and want to merge it into the main ('master') branch. We simply check out the 'master' branch and merge in the 'protlength' branch:

27: Type the following:

```
git checkout master
git merge protlength
```

If you made a mistake somewhere and this doesn't work, you may need to do:

```
git add proteins.txt
git commit -m " "
```

The merge is automatically committed, but we need to push the merged 'master' branch back to GitHub:

28: Type the following:

```
git push
```

Again take a look at `proteins.txt` to confirm that it now conains the protein length information.

13.3 Deleting branches

Having developed our new feature in the 'protlength' branch and merged it into the main 'master' branch, we can delete 'protlength' branch. Quite often people will work with a development branch locally and not bother pushing it to GitHub since it is development stuff that they don't want to share. This isn't always the case because (as we will see later) they may be working across multiple machines (perhaps a laptop and a desktop machine) and using GitHub to keep the systems in synchronization.

To delete the 'protlength' branch on the local machine, all that's needed is:

29: Type the following:

```
git branch -d protlength
```

To delete the branch on GitHub as well, we need the following two extra commands:

30: Type the following:

```
git push origin --delete protlength
git remote prune origin
```

14 Tags and releases

We have already encountered the rather long-winded identifiers that are used to label particular commits. However it would be much nicer for particular versions — perhaps those that we release to other people or those that we have used for the analysis that we publish in a paper — to be able to tag them with a much simpler label, typically a version number.

When we have a particular version checked out, we can tag it as follows:

31: Type the following:

```
git tag v1.0
git push --tags
```

The first command specifies the tag that you wish to use for this version. You can choose whatever tag you like, but 'v1.0' is probably a sensible option. The second command makes sure that GitHub knows about the tag too.

On GitHub, we can then create a 'release' of our repository. A release provides a ZIP file and a gzipped tar file that people can download. To do this, follow these steps:

1. Visit your repository on GitHub
2. Once you have created a tag ('v1.0') and pushed it to GitHub as described above, you should see the text **1 release** above the list of filenames in the repository. Click this text.
3. You will be taken to a screen where it lists the v1.0 release and on the right you will see a button labelled **Draft a new release**. Click this button.
4. Click in the box with the faint text 'Tag version' and type the tag you have used — as soon as you start typing the letter 'v' it should list your tag and you can click on it.
5. Enter a title for your release in the 'Release title' box — something like "First release version"
6. In the box below you can type some information about this release. Typically for a first release this again will be something like "first release", but for subsequent versions you might provide a list of the changes since the previous version. A method for obtaining this is described below in Section 14.1.
7. Click the **Publish release** button at the bottom of the page.

GitHub will now return to the releases page which provides links where you can download the code as a ZIP file or gzipped tar file[6].

[6]Note that these are not Git repositories, they are simply copies of the files as present in that tagged commit.

14.1 Summarizing your changes

The **git shortlog** command gives you the commit messages (the comments you provided with the **-m** option to **git commit**) without all the other information:

32: Type the following:

```
git shortlog
```

You can also obtain only the commit messages between two particular commits:
git shortlog *commitid1..commitid2*
The version that you have currently checked out can be abbreviated as **HEAD** so to find the changes between a particular commit and the current version, you can use:
git shortlog *commitid*..HEAD

As usual tags are abbreviated ways of accessing commit identifiers, so if you have created a tag for a particular version, you can use the tag name instead of the commit identifier:
git shortlog *tag*..HEAD

Consequently, when you create a new release (say 'v1.1'), you can obtain the changes from the previous release (say 'v1.0') with:
git shortlog v1.0..HEAD

15 Downloading a repository

You are now going to download a second copy of your repository from GitHub. Normally this is unlikely to be something you would want to do on the same computer. However, it is quite possible that you might have a copy of the repository on your laptop and another on a desktop computer — actually this is a really useful way of synchronizing and tracking changes across multiple machines keeping a backup on the cloud at the same time. Personally I do this to synchronize my laptop, my desktop at home and my machine at UCL. It is also possible that you might want to collaborate with someone else such that you would have a copy and someone else would have a copy.

Let's start by creating a `git2` folder that we can put the copy in (again you wouldn't normally do this; we are just pretending this is on another machine).

Start by creating a directory for your second copy:

33: Type the following:

```
mkdir ~/git2
```

Now we will 'clone' the repository so we have a second local copy in this other directory. Replace **USERNAME** with your GitHub username in the following:

34: Type the following:

```
cd ~/git2
git clone git@github.com:USERNAME/GitExercise.git
```

You should find that you now have a `GitExercise` directory exactly as you had before.

16 Managing multiple copies of a repository

We now have a repository in our ~/git directory and a second copy in ~/git2 which we are pretending is on a different machine.

Let's go to the ~/git2 version and add another file — it doesn't matter what this is, we are just testing what happens. We will create a file, then add and commit it to Git, finally pushing it to GitHub:

35: Type the following:

```
cd ~/git2/GitExercise
echo "Hello world" > hello.txt
git add hello.txt
git commit -a -m "Added hello.txt"
git push
```

Take a look at the GitHub page (refresh it if necessary) and you will see the new file is present.

Now let's go to the original version of our repository in ~/git:

36: Type the following:

```
cd ~/git/GitExercise
ls
```

Of course the hello.txt file is missing from this directory.

Remembering that we sent changes to GitHub using **git push**, it is fairly obvious that to get changes from GitHub, we use **git pull**:

37: Type the following:

```
git pull
```

You can now check that the new file is present and look at the updated log of commits that have been made:

38: Type the following:

```
ls
git log
```

16.1 Dealing with simple conflicts

Suppose you have changed a file on both copies of your repository. How can Git deal with that? If you have made distinct changes (to different parts of the file), Git can sort out the conflicts automatically; if the changes are to the same part of the file (usually the same line), Git will show you where the confict occurs and you will have to sort it out manually.

Let's add a line to the *start* of our proteins.txt file in one repository and add another line to the *end* of proteins.txt in the other repository.

First we will add a line to the start of proteins.txt in the first repository:

39: Type the following:

```
cd ~/git/GitExercise
```

Task:
Edit the file `proteins.txt` and add the following line to the *start* of the file:

```
Lysozyme          Mouse    LYZ1_MOUSE P17897 148
```

Now commit the changes and push them to GitHub:

40: Type the following:

```
git commit -a -m "Added mouse lysozyme"
git push
```

Now we will add a line to the end of `proteins.txt` in the second repository:

41: Type the following:

```
cd ~/git2/GitExercise
```

Task:
Edit the file `proteins.txt` and add the following line to the *end* of the file:

```
PFKM              Rat      PFKAM_RAT    P47858 780
```

Now commit the changes and try to push them to GitHub:

42: Type the following:

```
git commit -a -m "Added rat PFK"
git push
```

You will find that Git refuses to push the changes to GitHub because the version of the repository on GitHub has changed since you last synchronized. You should see a message like:

```
To git@github.com:USERNAME/GitExercise.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:USERNAME/GitExercise.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Instead you must pull down the changes to merge them with what you have done before you can push:

43: Type the following:

```
git pull
```

This will automatically merge the changes on GitHub with the changes you have made locally. The screen will change to a message saying something like:

```
Merge branch 'master' of github.com:USERNAME/GitExercise

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

(Your message may be somewhat different on non-Linux systems.)

Again this has taken us to an editor where we can add information, but we will simply accept the default message by typing the characters: **`:wq`** (colon, 'w', and 'q').

Now you can take a look at the `proteins.txt` file and you should find both sets of changes have been included in the file. We should now be able to push the changes to GitHub:

44: Type the following:

```
git push
```

Before we continue, we will go to the other repository and pull down the merged changes:

45: Type the following:

```
cd ~/git/GitExercise
git pull
```

Both versions of the repository are now back in synchronization!

16.2 Dealing with more complex conflicts

As we have seen, Git sorts out conflicts when they are in different parts of a file. However, when two repositories have a different change to the same line of a file, Git cannot sort it out automatically. Let's try making a change to the same line in both versions of the file.

First we will change the file in `~/git`:

46: Type the following:

```
cd ~/git/GitExercise
```

Task:
Edit the `proteins.txt` file and change the line

```
Hemoglobin-alpha  Human    HBA_HUMAN    P69905 142
```

to read

```
Hemoglobin-alpha  HUMAN    HBA_HUMAN    P69905 142
```

(it doesn't really matter what change you make — just change something!)

Now we commit and push the change:

47: Type the following:

```
git commit -a -m "Changed the hemoglobin alpha line"
git push
```

Now we will go to the repository in ~/git2 and make a different change to the same line:

48: Type the following:

```
cd ~/git2/GitExercise
```

Task:
Edit the `proteins.txt` file and change the line

```
Hemoglobin-alpha  Human    HBA_HUMAN    P69905 142
```

to read

```
Hemoglobin-ALPHA  Human    HBA_HUMAN    P69905 142
```

(again it doesn't really matter what change you make — just make sure you change something different in the same line!)

Now we commit and push the change:

49: Type the following:

```
git commit -a -m "Changed the hemoglobin alpha line"
git push
```

As with the simple conflict earlier, you will find that the **`push`** has been rejected by GitHub. As before you need to **`pull`** the changes from the server:

50: Type the following:

```
git pull
```

This time you should get a message that looks something like this showing that the automatic merge has failed:

```
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:USERNAME/GitExercise
   a6105a1..931daf5  master     -> origin/master
Auto-merging proteins.txt
CONFLICT (content): Merge conflict in proteins.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Have a look at the `proteins.txt` file and you will find a section that looks something like:

```
<<<<<<< HEAD
Hemoglobin-ALPHA  Human    HBA_HUMAN    P69905 142
=======
Hemoglobin-alpha  HUMAN    HBA_HUMAN    P69905 142
>>>>>>> 931daf596e8ec94552889074ba767e87d3b578fe
```

This indicates the region of conflict — the first section shows how the line looks in this repository; the second section shows how it looks in the GitHub version.

Task:
Edit the `proteins.txt` file and fix the merge conflicts manually. You need to remove the lines that start with `<<<<<<<`, `=======` and `>>>>>>>` and then combine the remaining lines as you see fit; probably you will want a single line that looks like:

```
Hemoglobin-ALPHA  HUMAN    HBA_HUMAN    P69905 142
```

Now you need to commit the changes and push them back to GitHub:

51: Type the following:

```
git commit -a -m "Merged conflicts in Hemoglobin alpha"
git push
```

Finally we will return to the other copy of the repository and pull down the changes:

52: Type the following:

```
cd ~/git/GitExercise
git pull
```

17 Summary

This tutorial has given an overview of the most common things that you need to do with Git and GitHub. They are extremely powerful systems and you can do a lot more with them, but for most people this will cover almost all of your needs. Get into the habit of committing regularly — every half hour or so and whenever you have made any sort of significant change.

At a minimum, all you need to do having created a repository is to remember to **`git add`** new files, **`git commit`** changes and **`git push`** them to GitHub. If you are working by yourself, these three commands will cover 95% of your interaction with Git. The other things described here (like going back to old versions of files) will cover another 4%. The other 1% is used so rarely that most regular users of Git will have to Google for help!

18 Command summary

18.1 Creating a local git repository

- Create a directory for your repository:
  mkdir -p ~/git/*projectname*
- Initialize git for that repository:
  cd ~/git/*projectname*
  git init
- Copy or create files that you wish to track with Git
- Add these so that Git knows they are to be tracked
  git add *filename*
  git commit -a -m "*message*"

18.2 Synchronizing your local repository with GitHub

- Specify that the GitHub repository is the origin for this repository:
  git remote add origin git@github.com:*username/projectname*.git
- Push your local files to the GitHub repository:
  git push -u origin master

18.3 Changing and adding files

- If you change a file, then you must commit the change and push to GitHub:
  git commit -a -m "*message*"
  git push
- If you create a new file, then you must add it to Git, commit it and push to GitHub:
  git add *filename*
  git commit -a
  git push

18.4 Looking at differences

- Look at differences between current versions of files and the most recent commit:
  git diff
- Look at differences between the current version and a specified commit:
  git diff *commitid*
- Look at differences between two specified commits:
  git diff *commitid1 commitid2*
- Look at differences between the current version and another branch:
  git diff *branchname*

In any of these cases, if you are only interested in a particular file, just add the filename on the end as well.

18.5 Check out a repository from GitHub

- Create a clone of your repository from GitHub:
  `git clone git@github.com:`*`username/projectname`*`.git`
- Pull down changes that have been made to the GitHub repository: **`git pull`**
- If you change your copy and the copy on GitHub has changed in the mean time, you won't be able to **`push`** your changes. You will have to **`git pull`** the repository which will merge the changes with your local changes. This will happen automatically unless the changes are in the same part of the file. In this case you will have to resolve the conflicts manually.

18.6 Creating branches for developing and testing new features

- Create a new branch named ***`feature`*** and switch to it using:
  `git checkout -b `*`feature`
  (you only need the **`-b`** when creating the branch)
- Link the branch to GitHub:
  `git push -u origin `*`feature`
- Switch back to the 'master' branch:
  `git checkout master`
- Merge the branch into the main version:
  `git checkout master`
  `git merge `*`feature`
  `git push`
- Delete the *feature* branch
  `git branch -d `*`feature`
  `git push origin -delete `*`feature`

Note: if you pull a branch down on another machine, it may not be able to synchronize changes by just doing **`git pull`** and **`git push`**. If this is the case, simply type:
`git pull origin `*`feature`
`git push -u origin `*`feature`
Once you have done that you should just be able to do **`git pull`** and **`git push`**.

18.7 Undoing changes

- Undo an uncommitted change to one file:
  `git checkout -- `*`filename`
- Undo all uncommitted changes:
  `git reset --hard`

- Undo a commit:
  `git revert commitid`
  (where ***`commitid`*** is the commit identifier obtained from **`git log`**).

- Obtain an older version of a specific file:
  `git checkout commitid -- filename`
  (where ***`commitid`*** is the commit identifier obtained from **`git log`**).

18.8 Tags and releases

- Create a tag for your current code (e.g. a version number)
  `git tag taglabel`
  `git push --tags`
  (where ***`taglabel`*** is a label such as 'v1.0').

- Create a release (with a ZIP file and a gzipped tar file for download) on GitHub.

- Obtain a list of changes from the previous tagged version for use in the release information using:
  `git shortlog oldtaglabel..HEAD`
  (where ***`oldtaglabel`*** is a tag or commit identifier of an earlier version).

19 Other tutorials

You are also recommended to look at some of these for more information:

- `https://guides.github.com/activities/hello-world/`
  — A short overview from GitHub

- `https://try.github.io/`
  — An online interactive tutorial from GitHub.

- `https://www.tutorialspoint.com/git/`
  — An excellent introductory tutorial

- `https://www.codecademy.com/learn/learn-git`
  — Another excellent introductory tutorial

- `https://git-scm.com/docs/gittutorial`
  — A fairly long overview tutorial

- `http://www.vogella.com/tutorials/Git/article.html`
  — A fairly long detailed tutorial

Note that these tutorials may suggest different ways of doing things from what has been presented above. I have shown you what works for me!