Git for Philosophers

What is Git?

When software developers work on complex programming projects, they use something called a *revision control system*. A revision control system allows them to keep track of changes in their code – it stores a history of changes, and allows them to quickly and easily take back ("revert") changes that turn out to break things. It also makes is easy to collaborate with others: multiple contributors can edit and add code, and the revision control system automatically integrates changes when possible, and alerts contributors to conflicts when it isn't. Software code is just text, and revision control systems work just as well with LaTeX code, or Markdown text, as they do with Java or Haskell programs.

Git is such a revision control system. It is distributed, which means that the entire history of your code lives not just on a server somewhere, but on your own computer and any other computer that uses the same shared code. Thus, you do not have to be connected to the internet to use Git to make changes, add material, or revert to a previous version. In order to collaborate with others, it is of course necessary to make code maintained by Git available over the internet. There are a number of websites which provide this service, the most popular ones are GitHub and GitLab. Both are free. GitHub is bigger, but GitLab is open source and allows you to have private projects without paying.

You are no doubt familiar with the history function in Word, and with automatic backup services like Dropbox. Git is a little bit like these, and so it might be useful to compare them as we go along. You can use Git as a backup solution, but it is *not* an automatic tool.

How does Git work?

A collection of files managed by Git are called a *repository*. A repository may be a single text document, or an entire collection. It is essentially a folder (possibly with subfolders) for which Git keeps track of changes to (some of) the files in it.

A particular state of a repository is called a *revision*. In contrast to automatic versioning (e.g., track changes in Word, or automatic updates in Dropbox), a

Git revision must be explicitly created. If you have a file on your computer that is tracked by Git, you can make any changes you like to it, but you have to tell Git when you want your changes to “stick,” i.e., to count as a new revision. This is called *committing*. When you commit changes in a Git repository, Git creates a new revision of your repository. A revision may include changes only to a single file, or it may include changes to files, new files added to the repository index, files renamed, moved, or deleted. Each revision is assigned a unique 40-digit hexadecimal “hash” code (often abbreviated to a 10-digit code). When you commit changes to create a new revision, the identity of the committer (i.e., you) as well as a *commit message* describing the changes is recorded.

One of the differences between Git and, say, Dropbox, is that any new Git revision has to be created manually by committing, while Dropbox just checks if a file has changed on your disk, and makes a new revision whenever that happens. Another difference is that Git only tracks those files it has been asked to track, while Dropbox tracks every change in every file in the Dropbox folder. To ask Git to track a file, you *add* that file to the repository index.

A third difference is that Git works locally until you tell it to save or load changes from the cloud, whereas Dropbox not only records your changes automatically, it also saves those changes to the cloud, and any change in the cloud automatically is mirrored on your own computer without you having to do anything – but also without asking you first!

To save the revisions in your Git repository over the internet, your Git repository must be linked with a version of the repository on a server somewhere (e.g., on GitHub or GitLab). This server repository is called a *remote*. Think of it as the cloud copy of your local Dropbox folder. However, the remote repository is not automatically synced with your local version. You have to tell Git to copy the changes in your local repository to the remote. This is called *pushing* your changes. In the other directions, to incorporate changes to the remote copy into your local repository, you *pull* changes from the remote. It’s in this case, when the remote repository has been changed (by you, from a different computer, or by a collaborator), that Git is most useful. If a file is changed in your Dropbox folder at the same time you’re editing it on your own computer, Dropbox will make a new copy of the entire file and leave you to figure out if there are conflicting changes and what to do about them. If you pull changes from a remote Git repository, Git will try very hard to compare your local version with the remote version of each file. If it is possible to merge changes automatically, Git will do so. If not – e.g., when both you and your collaborator have changed the same sentence – Git will alert you to a conflict that has to be reconciled manually. In practice, you will find a version of the file with both lines marked; you delete the line you want don’t want to keep, save the file, and commit the change to create a revision that reflects both your and your collaborator’s change.

How do I use Git?

In order to use Git on your computer, you have to install the Git software. The bare-bones "command line" Git client is available for all operating systems, but there are a fair number of graphical interfaces available as well. The best place to start is on the Git download page. Git comes with two graphical tools, `git gui` and `gitk`. For Windows, there is TortoiseGit, which will make all the Git commands available via the Windows Explorer right-click menu. If your repositories are or will be hosted on GitHub, you can use GitHub's own graphical tools for Windows and Mac.

Once you have Git and possibly a graphical Git tool installed, you have to set up a repository. There are two ways to do this. The first is using the `git init` command: in the folder/directory you want to track using Git, just say `git init`. Most of the time, however, you want your repository to have a matching remote version on a server. Then it will be a lot easier to first create the repository on the server and create a local copy of that repository, which will then be linked to the remote. Creating such a local version of an existing repository is called 'cloning.'

So go to GitHub or GitLab and create a repository. In both GitHub and GitLab, when you are logged in, there is a little '+' next to your username in the top right corner which lets you add a new repository (GitLab calls them "projects"). Once your repository/project is created, you can clone it to a repository living on your computer using the clone URL at the bottom of the right sidebar in GitHub or at the top of the project page in GitLab. There are HTTPS and SSH versions of those URLs. HTTPS always works, but you will have to give your GitHub/GitLab user ID and password whenever you push or pull. SSH can be set up to avoid that, so it's preferable, but it is a bit of a hassle to set up on Windows.

You can of course also use Git to clone repositories not created by you. For instance, the Open Logic Text is a logic textbook project that uses Git, and you could use the clone URL on its GitHub page to download a copy using Git. This very document can be cloned from its GitHub page as well. In the same way, you can clone a repository set up by a collaborator for a document you are planning to work on together. However, only in the latter case will you have permissions to change the repository on the server. Instead of cloning, e.g., the Open Logic repository directly, you can first ask GitHub to make a copy of it, which you then own. This is called a *fork*. A fork of a repository is a snapshot of the original, which you have complete control over. In particular, you can clone it to your own computer, and push any changes you make back to GitHub or GitLab. Both GitHub and GitLab display a "fork" button in the top right corner of the repository page which allows you to do this.

Once you have a clone URL, you can create your local version of the repository with the command

```
git clone URL
```

To track your own work, you'll start with a new, empty repository and clone it. Suppose you are 'user' on GitHub and have started a repository 'project'. You make a local version of this repository using

```
git clone https://github.com/user/project.git
```

and this repository will live in the directory/folder 'project' on your local drive. You can add a new file to that folder, but to have it tracked you also have to add it explicitly to the Git index:

```
git add new-file.tex
```

Now Git knows that new-file.tex should be tracked. To create a new revision of your 'project' repository which records all the changes to your tracked files, say

```
git commit -a
```

The switch "-a" is for "all": Git will 'stage' all changes for all tracked files in the repository.

Then to sync your local changes to GitHub/GitLab, you say

```
git push
```

Collaborative Writing with Git

Collaborative writing presents similar issues as collaborative programming: different people making changes to the same document from different locations. Sending the document back and forth is inefficient: only one person can work on it at a time, and there is a risk of changes being made in parallel which are then either overlooked or which are hard to reconcile. Collaborative editing tools (e.g., Google Docs for Word documents, shared LaTeX editors like ShareLaTeX and Overleaf help if you're using those formats, but have their own drawbacks (e.g., they can't be used offline). Simply keeping your shared Document in a Dropbox folder is a partial solution, but the doesn't solve the issue of conflicting changes to your shared document. Dropbox assumes that you know what you're doing. So if you make a change to a document on your office desktop, and then make a change to the same document on your laptop, the later change will silently overwrite the former. If for some reason you didn't update the file on your laptop with the changes from your office desktop (say, if you were without an internet connection), these changes will be gone from the document when you save it the second time. The changes will be preserved in a previous version of the document, since Dropbox keeps every intermediate version – but it might take you a while to notice that your change got lost, and it might take you a while to find the most recent version of the file on Dropbox that still has it.

Dropbox is a bit more careful when two different users make changes to the same document in a shared Dropbox folder. If your collaborator has edited the file while you were away, you will see the changes automatically. But if you've had your file open in an editor while your collaborator has made changes – perhaps even with your laptop asleep! – Dropbox will realize that there may be a conflict between your version of the document and your collaborator's. But it won't know what to do. Rather than overwrite one of your changes, Dropbox will make a copy of the file you're working on simultaneously, and leave you to figure out which version is more up-to-date and how to reconcile the two copies into one. So if author A and author B are both working on a document, author A adds a sentence to the introduction and at the same time author B adds a sentence to the conclusion, your shared Dropbox folder will suddenly contain a second version of the document, `document (author B's conflicted copy).doc`, say. It will contain the additional sentence in the conclusion, but not author A's additional sentence in the introduction, and `document.doc` will contain author A's sentence but not author B's. Dropbox will leave it to you to figure out who has made what change where and how to reconcile them and integrate the two documents into a single one. This is annoying and complicated, especially if the conflict goes unnoticed for a while.

Using Git helps you avoid these problems to the extent it is possible to avoid them.

When you have set up a repository for your writing project (say, containing a LaTeX document plus a BibTeX file for references), you can edit the files, commit your changes to generate a new revision, and periodically push your revisions to the remote repository on GitHub or GitLab. If you are working on the project with someone else, you can give them access to the repository as well. If they have *push access*, they can send their own revisions to the shared repository just like you do.

Git keeps track of the state of your own local repository and the remote on GitHub/GitLab. The command

```
git status
```

will prompt Git to display the status of your repository: which branch you are on (typically, this is the *master* branch, but more about branches later), whether your local version of the repository is up to date with the remote or not, which files have changes that are waiting to be committed, and which files are untracked. If you have commits that you have not pushed to the remote yet, Git will report something like “Your branch is ahead of ‘origin/master’ by 1 commit.” The remote repository is usually called *origin*, and the remote branch that corresponds to your local *master* branch is then called *origin/master*. It might happen that your branch is *behind* the remote: if your collaborator has pushed changes to the shared remote, there will be commits on the remote that you don’t yet have in your local repository. Before you can push your changes, you will have to incorporate your collaborators’ changes into your own local version of your paper.

If your document is under Git control, you have to pull changes from the shared repository before you see what your co-author has done. By the same token, you and they have to remember to push new commits to the repository, or there won’t be any changes to pull. In the crucial case where you have both made changes at the same time, however, Git will handle the discrepancies gracefully.

In the best case scenario, you and your co-author have edited the same file, but you haven’t edited the same *part* of the file: say, they added a sentence to the introduction, you have cleaned up a passage in a middle section. If they have committed their changes and pushed to the shared remote, Git won’t let you push your changes. You’ll get an error message like

```
! [rejected]        master -> master (fetch first)
```

If your changes do not conflict (were not made to the same line of text), Git can fix this automatically. Just say

```
git pull
```

Git will then download (“fetch”) the changes from the remote and automatically merge them with your version of the repository, creating a new revision which includes both your changes and the older changes by your co-author. Your repository is now ahead of the remote by one commit (the act of merging your co-author’s changes with your own created a new revision), and you can push the combined changes to the remote.

If you did happen to both make changes that cannot be merged automatically, Git will alert you to this fact:

```
CONFLICT (content): Merge conflict in <filenames>
Automatic merge failed; fix conflicts and then commit the result.
```

Instead of letting you fend for yourself in figuring out what has changed and where, Git will tell you exactly what you have to fix. The files with editing conflicts will now contain the conflicting lines, indicating your and their changes, e.g.:

```
....preceding text...
<<<<<<< HEAD
what you wrote
=======
what your co-author wrote
>>>>>>> hash code of your co-author's commit
...following text
```

in the relevant place in the document. Fix up just that part of the document, e.g., delete the lines with <<<, ===, and >>> and have the file read:

```
....preceding text...
what you wrote, but also what your co-author wrote
...following text
```

Save the file, then say `git commit -a`. Your local repository now contains a conflict-free version of both your changes, which is ahead of the shared repository by one commit, and Git will again let you push to the remote. When your co-author returns to work and says `git pull`, they will have the merged, clean version of the document.

Note that your intervention is only required if both you and your co-author have made changes to the very same line of text, otherwise Git will merge the changes automatically. This includes the case where one of you adds a paragraph and the other one cuts text somewhere else in the file: after `git pull` the file will contain the new paragraph and the deleted text will be gone. For instance, suppose your file looks like this:

```
Introduction

Middle

Conclusion
```

Author A adds a sentence:

```
Introduction

A remark by author A

Middle

Conclusion
```

Author A commits this change and pushes to the repository.

Author B (you) adds a sentence to the conclusion:

```
Introduction

Middle

Conclusion

A concluding remark by author B
```

You commit your change, but `git push` results in a warning, asking you to "fetch first." So you say, `git pull`. Now your file looks like this:

```
Introduction

A remark by author A

Middle

Conclusion

A concluding remark by author B
```

In other words, your changes to the same file were merged automatically. If you now say `git push` the merged file will also be available to author A on the remote. No intervention in the file itself is needed in this case.

Forks and Pull Requests

If you have push access to a repository, you can sync your local clone with the remote on GitHub or GitLab directly. But many projects allow push access only to a select group of people to make sure no-one breaks anything. There are some larger collaborative writing projects like this, e.g., the Open Logic Text, the HoTT Book, and Alex Dunn's Markdown version of Walter Ott's Modern Philosophy text. You may wish to set up a project in such a way, to allow others to use, improve, and contribute to it while retaining control. Git is set up for this kind of scenario: it is called the *fork and pull* model of collaboration. In the fork and pull model, your collaborators each work on their own "fork" of the repository. Forking your repository is easy: both GitHub and GitLab display a fork button on the repository page. Your contributor clicks on that button, and a complete copy of the repository appears in their own GitHub/GitLab user space. If your repository lives at `github.com/author/project` then their fork will live at `github.com/collaborator/project`. Instead of cloning the `author/project` repository, and pushing to it directly, they clone the `collaborator/project` repository and work on that private copy.

The `collaborator/project` repository is a complete copy, but it is not automatically kept in sync with `author/project`. If your collaborator wants to have changes to your repository included also in their forks, they have to do this explicitly. If you have forked a repository, e.g., you have your own fork `collaborator/OpenLogic` of the `OpenLogicProject/OpenLogic`, you also have to explicitly merge changes into your fork from the original repository. You do this on your local clone on your own computer. Suppose you have cloned the `collaborator/OpenLogic` repository (your own fork) on your computer's hard-drive. It is already set up with a remote, namely your fork on GitHub. Pulling and pushing usually happen between your local clone and the remote fork `collaborator/OpenLogic`. To sync changes from the original `OpenLogicProject/OpenLogic` repository, you have to add it as another remote:

```
git remote add upstream https://github.com/OpenLogicProject/OpenLogic.git
```

Here `upstream` is the traditional name given to the original remote repository (in contrast to `origin` which is the remote repository on your own GitHub account, which happens to be a fork). Of course, the URL will point to whichever upstream repository you want to track.

Now you can tell Git to pull changes, not from the remote `origin` but from `upstream`, and merge them into your local repository using the command

```
git pull upstream master
```

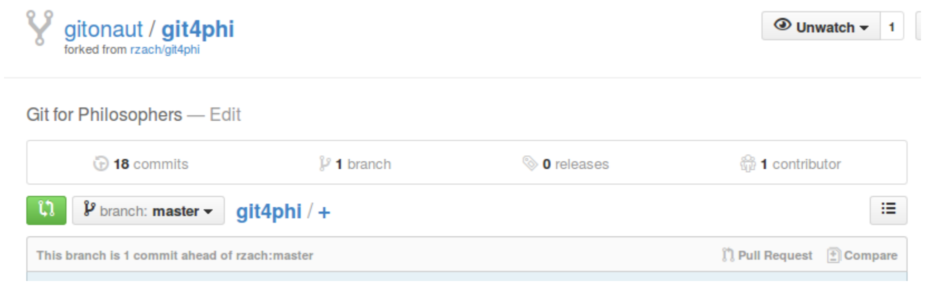
Your local clone of `reader/OpenLogic` contains all the changes from the upstream `OpenLogicProject/OpenLogic`. To update these changes also in your own fork on GitHub, simply say

```
git push
```

If changes to the upstream repository can be integrated automatically into your fork, it's also possible to do this on GitHub/GitLab by sending yourself a pull request. We'll have to discuss what that is first, though.

Having a copy of a repository, and keeping it up-to-date with the original, "upstream" version, is only half the fun. The real advantage of being able to edit your own *fork* – rather than just your own *copy* – is that your additions and corrections can be incorporated back into the source repository. For instance, you might want to correct a typo, help with some copy-editing, or, in a collaborative project, contribute additional material. The editors of the source repository may want to retain control over who gets to contribute what, and so they don't give push access to anyone. But you can work on your own fork, and make those corrections and additions there first.

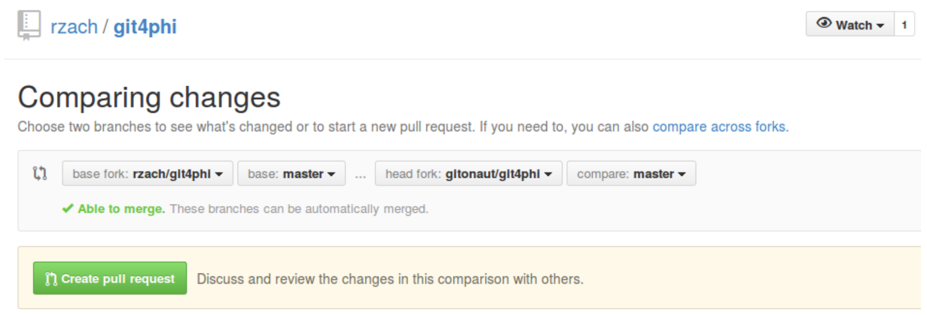
When you are done, you can send a /pull request/ to the owners of the source repository. GitHub/GitLab remember from which repository your fork was created, and will display if your fork contains changes not in the upstream source repository. Here's what user `gitonaut` sees after they've added some material to their fork of this file:



This branch is 1 commit ahead of rzach:master

Figure 1: GitHub fork view

When your fork is "ahead" of the original repository, you can send a *pull request*: alert the owners of the source repository that your fork contains changes which they might want to pull, i.e., merge into their repository.



Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

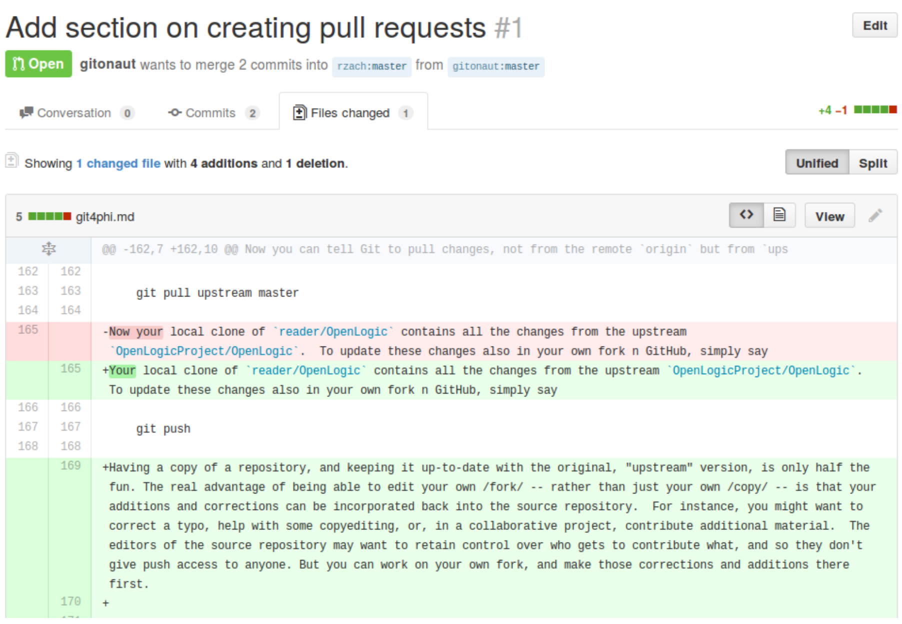
✔ Able to merge. These branches can be automatically merged.

Discuss and review the changes in this comparison with others.

Figure 2: Creating pull request

The pull request shows up as an "issue" in the GitHub view of the source repository. The owners of the original repository can see who sent the pull request, read the description, look at the line-by-line changes in all affected files. For instance, the pull request adding a paragraph above in this file by `gitonaut` looked like this:

The repository owners can respond to the pull request with a comment, e.g., asking for clarification or explaining why they can't merge the proposed



Add section on creating pull requests #1

Open **gitonaut** wants to merge 2 commits into `rzach:master` from `gitonaut:master`

Showing 1 changed file with **4 additions** and **1 deletion**.

5 git4phi.md

```
@@ -162,7 +162,10 @@ Now you can tell Git to pull changes, not from the remote `origin` but from `ups

     git pull upstream master

-Now your local clone of `reader/OpenLogic` contains all the changes from the upstream
 `OpenLogicProject/OpenLogic`.  To update these changes also in your own fork n GitHub, simply say
+Your local clone of `reader/OpenLogic` contains all the changes from the upstream `OpenLogicProject/OpenLogic`.
 To update these changes also in your own fork n GitHub, simply say

     git push

+Having a copy of a repository, and keeping it up-to-date with the original, "upstream" version, is only half the
 fun. The real advantage of being able to edit your own /fork/ -- rather than just your own /copy/ -- is that your
 additions and corrections can be incorporated back into the source repository.  For instance, you might want to
 correct a typo, help with some copyediting, or, in a collaborative project, contribute additional material.  The
 editors of the source repository may want to retain control over who gets to contribute what, and so they don't
 give push access to anyone. But you can work on your own fork, and make those corrections and additions there
 first.
+
```

Figure 3: Viewing changes in pull request

changes. It is also possible to add comments to individual lines in the commits making up the pull request. If the changes in the pull request are simple and do not conflict with any changes that have meanwhile been made to the source repository, GitHub will show a button that allows the repository owner to merge the changes automatically. Your contribution can be easily incorporated into the repository in this way. The history of the files you changed will contain a record of the change merged via the pull request, and you will be listed among the contributors to the files you changed. For instance, this file now lists `gitonaut` as a contributor, and the "blame" view of the file shows the contributions made by `gitonaut`.

Now if you are looking at your fork on GitHub, it may also be "behind" the original repository. You may be able to integrate updates to the original repository into your fork also directly on Github. To do this, proceed as before t ocreate a pull request. When GitHub shows you the changes after you click the "pull request" button, change the base and head fork. The base fork is the fork where you want to want changes to be applied, and the head fork is the fork where you want changes to come from. By default, the base fork is the original repository, which should receive the changes from the default head fork, your forked repository. If you switch these, the pull request will apply changes to the original repository in your fork.

Branches

Programmers are fond of using “branches” in their code. A code branch is a version of the entire project that shares it change history, but includes some changes the main branch (usually called “master”) does not (yet) contain. This is useful, for instance, when you are adding new functions to your program. The new functions may be implemented in a separate file that is added to the project, but it may also require changes in other, existing parts of the project. Until the new function is developed and tested, you don’t want to isolate these changes in a development area. After all, the other parts of the program are still used, and perhaps even continue to be developed. Adding possibly buggy code in the process of developing the new function shouldn’t interfere with development of the old, tried-and-true parts of the project. So to do that, a software project will add a “feature” branch where all the new development takes place, and when it’s complete the changes will be merged back into the “master” branch.

This is not a situation that a run-of-the-mill collaborative writing project will encounter. But it is useful to understand what branches are, because behind the scenes, Git’s functionality for dealing with branches is what’s behind most of the features discussed so far. For instance, pulling changes from a remote repository in fact uses this functionality. If your repository is linked to a remote, the content of the remote is actually a branch of your repository, “origin/master”. The `git fetch` command updates the content of this branch to the content of the remote. The `git merge` command allows you to merge a branch into another, e.g., `git merge origin/master` would merge the content of the “origin/master” branch into the current “master” branch. An that’s actually what `git pull` does: first `git fetch` to update the “origin/master” branch, then `git merge origin/master` to merge the remote changes into your local “master” branch. Similarly, merging a pull request is a combination of a `git fetch` the contents of your contributor’s repository into a, say, “gitonaut/master” branch, and then merging that branch into your own “master” branch. GitHub and GitLab do this behind the scenes when you click on the “merge pull request” button, you can also do it by hand. A situation where you might want to do that is if you want to adjust the changes made by your contributor before merging into the “master” branch.

There are other situations when branches might be helpful. One is analogous to the “feature” branch in a software project. Suppose you are working on a collaborative book project (such as the Open Logic Text), and you want to add a chapter on a new topic. You will star writing the chapter in a separate file, of course. But there are perhaps other areas of the book that will be affected, e.g., the table of contents, cross-references, the bibliography, the index. While your new chapter is not yet read for prime time, you may still want to exclude any mention of the new material. Someone printing the text shouldn’t have to print your half-baked ideas and placeholder bibliography entries. At the same time, /you/ do want to be able to see what the book will look like with all the new cross-references etc. The solution is to work on your new chapter in a branch. When your new chapter is ready, you can then merge your branch into "master". Before it is ready, any changes to the master branch can also be merged into your "feature" branch, so the copy of the project in which you're working on the new chapter will also include all the changes you or other people make to other parts of the text.

Another possible application of branches is when you want to develop a slightly different version of your project. For instance, say you are co-authoring a paper using git, and you're ready to send it to a conference. You can start a new branch, and anonymize the paper on that branch. That's the version you submit. You get some feedback at the conference which you incorporate, and fix some typos or add a reference or two to the master branch. Now you want to send it to a journal. You've made changes to the original paper, but it does take some effort to anonymize it. Rather than anonymize it again, you can just switch to the anonymized branch, marge the changes from the master branch, and have an updated but still anonymized version of the paper.

Let's see how this works in action. Suppose you want to prepare a version of this paper for a slightly different audience, e.g., Victorianists instead of philosophers. We are going to keep the Victorianist version on a branch, "victorianists".

To make a new branch, say

```
git branch victorianists
```

To switch to work on the branch, say

```
git checkout victorianists
```

Now any change you make and commit will be committed to the "victorianists" branch, instead of the "master" branch. If you say `git push`, Git will first complain that your new branch has no matching branch on GitHub.

```
fatal: The current branch victorianists has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin victorianists
```

After you do that, your remote will be updated with a matching branch, and you can push the branch to the GitHub repository. You can make the changes to the text required on the branch, e.g., change references to LaTeX to TEI-XML. You can look at what that would like here.

To add more material to the text, or to fix typos, you probably want to switch back to the "master" branch.

```
git checkout master
```

Now the file will be reset to the version on the “master” branch. Any changes you commit from here on will be recorded in the “master” version, but not in the “victorianists” version. You can merge those changes into the “victorianists” branch using

```
git checkout victorianists
git merge master
```

so the Victorianists also get the added content and corrections.

Branches are useful sometimes, it’s good to understand how they work because that’s how many of Git’s features work “behind the scenes”, but they can also be confusing. If you do work with branches, you have to be extra careful to make sure you commit your changes to the right branches, and that you merge changes from the “master” branch regularly to avoid having to do it manually when the branches get too far out of sync. It’s now also very important to `git pull` often, otherwise you will have to merge changes from the remote not just into “master” but also into your feature branches.