Chapter 3

Using `Git` and GitHub.com

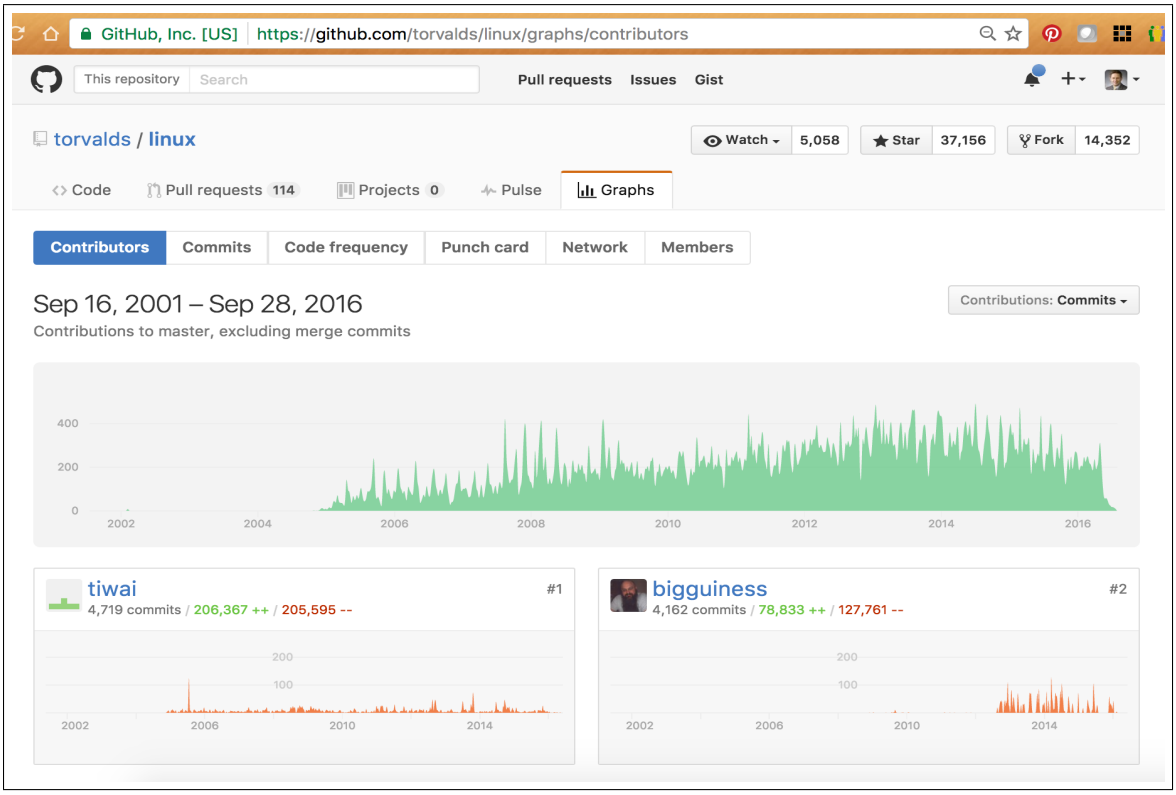
In technical terms, `Git` is a distributed version control system (DVCS). This means that users have complete copies of a source repository on their local hard drive. `Git` is valuable as a local version control system (LVCS) in that it can allow you to track changes in files and directories on your local computer without any connectedness to the internet or to other collaborators. However, `Git` 's most powerful characteristics come from its ability to carefully allow multiple users to collaborate on the same files and record the changes in an ordered, structured, hierarchical way.

`Git` is the software on your local machine that executes the commands and takes the snapshots that track the changes in marked files on your local machine and integrates those changes with remote repositories. The remote repositories are hosted by companies like GitHub.com or Bitbucket.org. This chapter will focus on GitHub.com, but Bitbucket.org is a good alternative with slightly different strengths and weaknesses.

`Git` software was born out of a dispute between the Linux kernel developers and the original version control provider for that group. The developers ended up creating their own free distributed version control sytem, which is `Git` .[1] GitHub.com currently hosts thousands of open source repositories with virtually unlimited numbers of contributors to each repository. The GitHub repository for the Linux kernel has over 200 contributors. Figure 3.1 shows a snapshot of the contributors page of the Linux repository on GitHub. This

[1] See a short history of Git in Chacon and Straub (2014, p.5), which is also freely available online at https://git-scm.com/book/en/v2/.

Figure 3.1: Screenshot of Linux Contributors



This snapshot was taken on September 28, 2016.

page shows exactly who has contributed, when they contributed, and what they contributed. Some employers are more interested in a potential employee's GitHub page than they are in that person's resume.

3.1 Why Not Use Dropbox or Google Docs?

Easy and commonly used alternatives to `Git` as your version control and collaboration platform are Google Docs and Dropbox. These two `Git` alternatives are file storage systems that sync changes to files across multiple storage locations of a single user or across many users. For simple file sharing, storage, and syncing, Google Docs and Dropbox are often preferred to `Git` . But for projects in which hierarchical permissions of who can edit, careful tracking of contribution attribution, and version history are important, `Git` is preferred.

Dropbox is nice because changes to a shared document on one person's machine are automatically updated on another person's machine. Dropbox offers some storage of previous

versions of files. But it does not have detailed description and does not go back very far. Furthermore, Dropbox has trouble merging changes to a document that happen simultaneously. Suppose that you and your collaborator open a shared document simultaneously on your respective machines, and you both make changes to that document. Dropbox does not know whose changes dominate, so it updates the main document with the changes of whoever saves first and then makes a "conflicted copy" from the saved changes of whoever saves last. It is then up to the user to figure out how to manually merge those two files.

Google Docs have no merging problem because the document is automatically updated in real time on each user's computer, regardless of whether the document has been opened or not. This is made possible because a Google Doc resides primarily on remote Google servers. Despite this remote predominance, Google Docs do allow users to store copies of the files on their local drives to be able to use the documents while off-line. To a slightly greater degree than Dropbox, Google Docs allow some version history of who made changes, as well as a nice chat and comment interface for collaboration. But in Google Docs, everybody often has the same level of permission on making changes.

`Git` requires more deliberate decisions and effort about what gets merged, what does not get merged. And git has more specific rules about who decides what gets incorporated into the code and what does not. But with this extra complexity comes extra order, which is essential for large projects with lots of contributors. Additionally, `Git` provides a more specific version history with more refined ability to revert your code to a particular point in that history.

`Git` , Dropbox, and Google Docs each have different strengths and weaknesses. But `Git` is the standard for large projects with many contributors and a need for careful version control, changelog history, and contribution attribution.

3.2 Installing `Git` and Settings

A good set of instructions for installing `Git` is available on the `Git` website.[2] This `Git` site states, "Even if it's already installed, its probably a good idea to update to the latest version."

[2]See https://git-scm.com/book/en/v2/Getting-Started-Installing-Git.

This textbook recommends that you follow this instruction and update `Git` on your local machine. It is worth noting that `Git` comes installed on every Mac OSX.

Once `Git` is installed on your machine, you should update the settings in the `git config` tool. The `git config` tool controls how `Git` looks and operates and customizes `Git` with your information. The obvious starting place is to enter your user name with which you contributions will be associated as well as your e-mail address at which collaborators can contact you.

```
>>> git config --global user.name "FirstName LastName"
>>> git config --global user.email youremail@example.com
```

The `--global` option tells `Git` that these values are the default values that only need to be entered once. You can see all of the `--global` settings in `git config` by typing the `--list` command.

```
>>> git config --list
```

3.3 `Git` and GitHub Structure, Workflow

A number of different `Git` workflows are used in open source projects, but most recommended flows include some form of *fork→branch→pull request*. This textbook suggests the workflow displayed in Figure 3.2. At first glance, this workflow looks very complicated and might make the user wish for the ease of Dropbox or a Google Doc. But the workflow depicted in Figure 3.2 exhibits some important principles and rules that protect code integrity and allow for many organized contributors.

The first characteristic to note from the workflow displayed in Figure 3.2 is the protected sanctity of the main code repository, labeled Ⓐ. There is only one arrow ⑩ leading into the main repository. Submitting a pull request is the only way for foreign code to be incorporated into the main repository. Related to this point is the characteristic that all work on the main repository Ⓐ is performed in separate and separated repositories, both remote and local. This is highlighted by the horizontal dotted line in Figure 3.2 that separates the main repository from everything else in the figure.

Figure 3.2: Flow diagram of Git and GitHub workflow

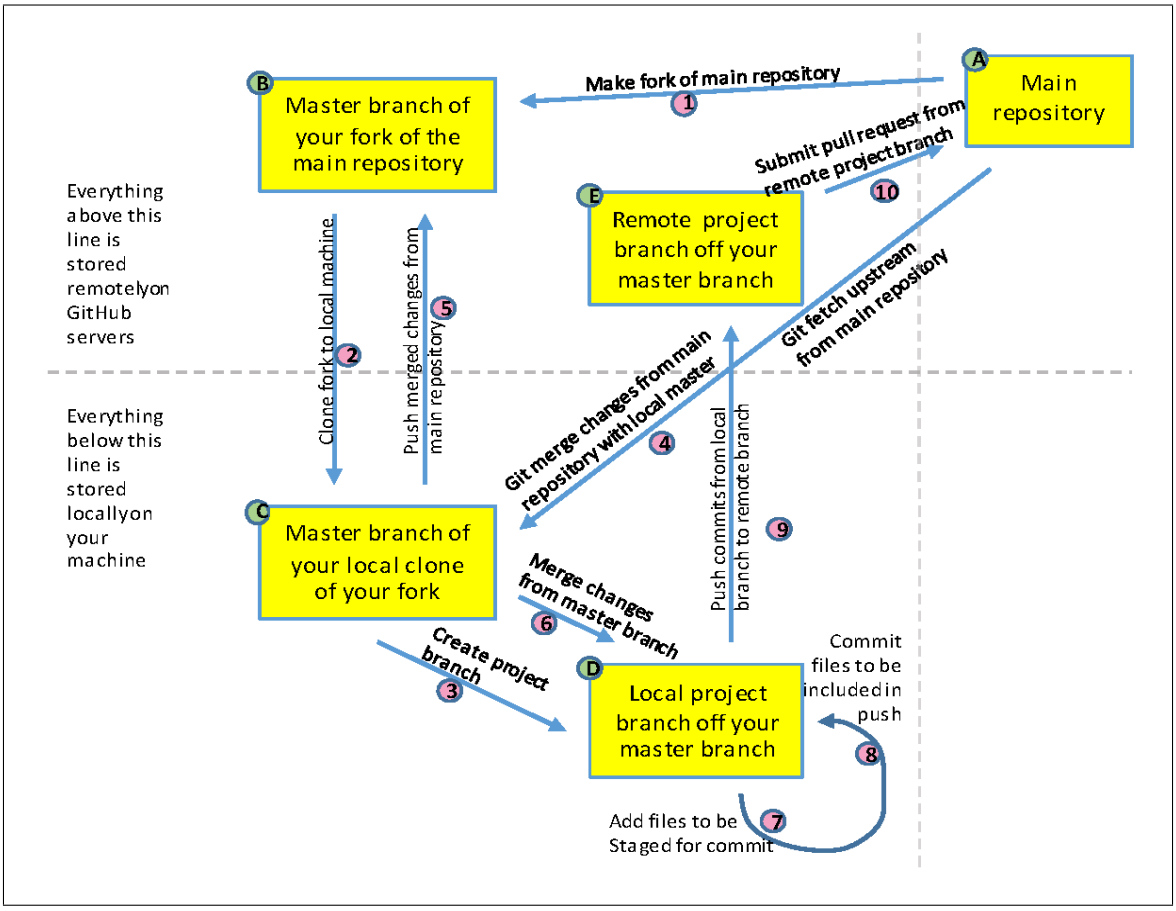


Table 3.1: List of common Git commands

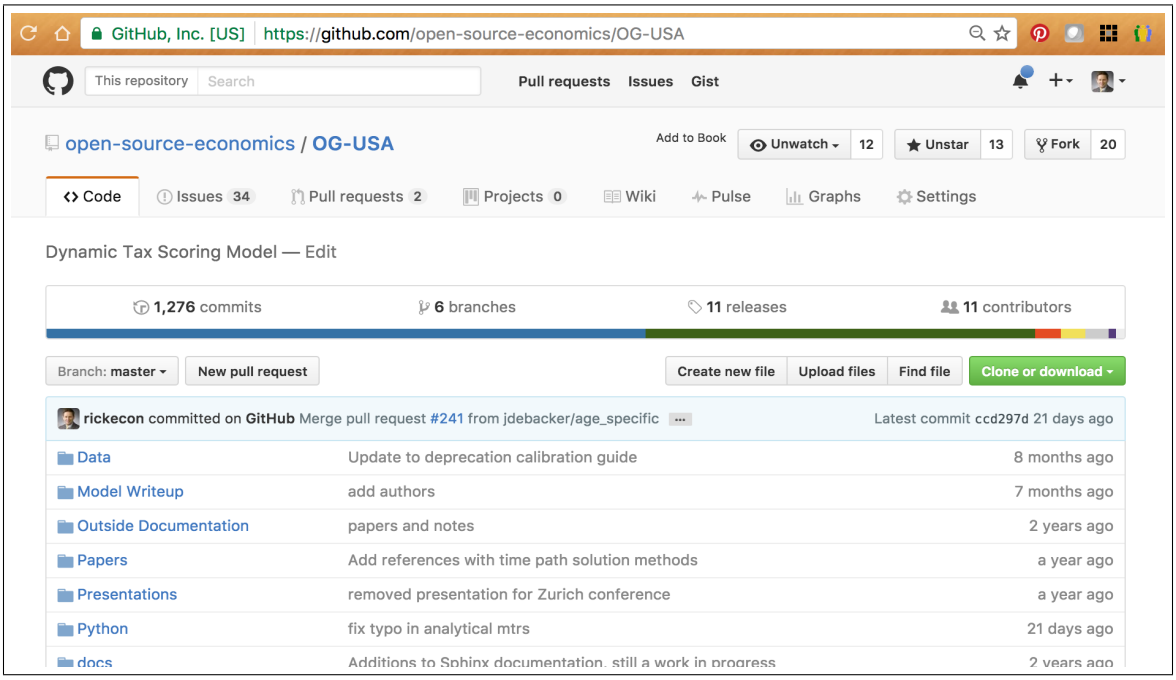
Fig. 3.2 reference	Git command
①	Click "Fork" button at `https://github.com/main_acct/main_repo_name`
②	`git clone https://github.com/fork_acct/main_repo_name.git`
③	`git checkout -b [BranchName]`
④	`git fetch upstream` and `git merge upstream/master`
⑤	`git push origin master`
⑥	`git merge master/[BranchName]`
⑦	`git add [FileName]` or `git add -A`
⑧	`git commit -m "[descriptive commit message]"`
⑨	`git push origin [BranchName]`
⑩	Click "New pull request" button at `https://github.com/fork_acct/main_repo_name`

One last introductory distinction to make with `Git` is the difference between remote repositories and local repositories. The horizontal line in Figure 3.2 separates these two concepts. Every object above the line in the figure is *remote* and is located on a GitHub server somewhere in the cloud. Of the remote repositories (Ⓐ, Ⓑ, and Ⓔ above the horizontal line), the two branches of your fork of the main repository (Ⓑ and Ⓔ) to the left of the horizontal line will be called *origin* and the main repository Ⓐ will be called *upstream*. Section 3.3.3 discusses the significance of the *upstream* reference.

3.3.1 Create a fork and clone it

Assume that the main repository is a GitHub repository. The first step one takes in the `Git` workflow when joining a project is to "fork" the main repository. This is shown in step ① in Figure 3.2. This is done by going to the URL of the main repository on GitHub.com and clicking on the "Fork" button toward the upper-right corner of the screen as shown in Figure 3.3. GitHub will then give you the option to choose a GitHub account in which to place your fork of the main repository.

Figure 3.3: Main repository GitHub main page



A fork is a copy of the main repository that is placed in your remote GitHub account.

The name of your fork is the same as the name of the main repository. The remote forked repository is labeled Ⓑ in Figure 3.2. The only difference at this point is that the main repository is in a different account than your fork. It is important in the `Git` workflow that all changes that are made to the code from the main repository are made is a completely quarantined copy—the user's fork.

Once you have successfully forked the main repository, you want to *clone* your fork. This action is represented by ② in Figure 3.2. Cloning is the `Git` terminology for making a copy of a remote repository on your local machine, which local repository is tracked and related to the remote repository by the `Git` software. The local clone of your remote fork is represented by Ⓒ in Figure 3.2. You clone the remote fork opening your terminal on your local machine, navigating to the directory in which you want to place the cloned repository and typing the following command,

```
>>> git clone [remote fork Git URL]
```

where `[remote fork Git URL]` is the address that you copy when you click on the green "Clone or Download" button, which is below the "Fork" button on the main page of your remote fork Ⓑ (not the main repository Ⓐ).

3.3.2 Branching, making changes, updating your remotes

Branching is one of the most powerful functions of `Git` . Once you are ready to start modifying the code, this textbook recommends that you always make those changes in a new branch of your local fork. Each repository can have multiple branches. Branches represent copies of the repository that need not be identical. Think of each branch as representing a different project on the code repository.

The main branch of a repository is called *master*. It is automatically created with each newly forked or cloned repository. The master branch's purpose in this `Git` work flow is to be the baseline or fundamental reference, which is kept in sync with the main repository Ⓐ (see Section 3.3.3). Branches Ⓑ and Ⓒ in Figure 3.2 are the master branches of the remote fork and local fork, respectively.

You can always check what branch of your repository you are in by typing the following

command.

```
>>> git branch
```

This will list all the branches of your repository, and it will highlight with an "*" the branch you are currently in. It is very important to always be sure you are working in the correct branch.

All changes to the master branch and new work should be in a new branch. You create a new branch off the master branch by the following command, which is action ③ in Figure 3.2.

```
>>> git checkout -b [NewBranchName]
```

This command both creates the new branch and changes your directory to the new branch, no longer in the *master* branch. If you have multiple branches, you can change between them by typing:

```
>>> git checkout [BranchName]
```

It is important to note that the files in your local `Git` directory change when you change branches to the files associated with that branch. It is, therefore, important to make sure you are making changes in the branch with which you intend those changes to be associated.

As you work on your code and change the files in your repository, there are three steps you need to follow. You must (i) *add*, (ii) *commit*, and (iii) *push* changes to your branch in your remote fork Ⓔ from your local branch Ⓓ.

You can check the status of the branch of your local repository by typing:

```
>>> git status
```

This will show you if any files or folders have been modified, added, or deleted. You choose which of those files to track or stage for a future commit by adding them to the "staging area" as shown in action ⑦ in Figure 3.2. You can add a particular file by using the following command,

```
>>> git add [FileName]
```

or you can add all the modified, added, or deleted files with the following command.

```
>>> git add -A
```

If you type `git status` after adding files to the "staging area" to be tracked by `Git` , you will see that the files you added are now shown with a different status.

Once you have completed an intuitive well defined set of changes is a good time to commit those *added* files. A *commit* is a bundled group of changed files that can be summarized in one or two short sentences. You commit all files in the staging area that have been previously added by typing the following command. This is action ⑧ in Figure 3.2.

```
>>> git commit -m "[descriptive commit summary message]"
```

As mentioned above, a commit message should be no more than two sentences, but is probably better as one sentence. This implies that you should commit your work often and not wait until you have completed whatever change your branch was created for. Never go too long without committing. And you should always commit at the end of a coding session or when switching branches.

A *push*, shown as action ⑨ in Figure 3.2, is an action that takes all the commits that have not already been *pushed* and copies them to the remote origin repository. This textbook recommends that commits from a project branch Ⓓ be pushed to a similar remote origin branch Ⓔ. A push is done by typing the following command.[3]

```
>>> git push origin [BranchName]
```

Each *push* will likely contain multiple *commits*. Notice that your master branch and project branch—both in your remote fork and in your local repository—will be different from each other.

Once you feel that your changes are done and you have pushed them to your remote branch so that Ⓓ and Ⓔ are identical, you are ready to incorporate them into the main repository Ⓐ. This is done through a *pull request*, as shown in action ⑩ in Figure 3.2. A pull request is made through the GitHub website. You go to the main page of the repository, make sure you are in the project branch on the website by clicking on the "Branch:[BranchName]"

[3] If you leave off the `BranchName` in the command, your changes will default to the remote master branch of your fork. We do not recommend this.

button in the upper-left area as shown in Figure 3.3. Then click on the "New pull request button", which is next to the "Branch:[BranchName]" button.

Before submitting this pull request, make sure it has an intuitive, descriptive, and concise title. Then make sure in the box below the title, that you put a detailed description of what is in the pull request. In the end, the pull request will be the cumulative changes from all the commits from all of the pushes since the creation of that branch. In the description, you may want to give context to the changes, and you may even want to point out areas on which you need an extra set of eyes.

Notice the different naming of this process. Rather than being called a *push* in which the energy is coming from the source of the changes, it is called a *pull request*. This name signifies that the energy comes from the destination of the change. You can think of a pull request as an invitation for the collaborators who run the main repository to *merge* your changes into the main repository. It is for this reason that this open source work flow allows for the full democratization of coding. Anyone can take the code and make any changes they want. But only the code that is accepted by those who manage the main repository is incorporated.

Once you make a pull request and before someone chooses to merge that pull request, the status of your branch is linked to the pull request. That is, you can continue to make changes to your local branch, add/commit/push those changes to your remote branch, and those commits will be automatically added to the pull request.

If the changes in your code are accepted, those who manage the main repository Ⓐ will *merge* in your changes. They may also open a dialog in the pull request in which the community can respond to and discuss the changes. In the end, the managers of the main repository have the option to reject the pull request.

Once your pull request is accepted and merged into the main repository. We recommend that you `git fetch upstream` the changes from the main repository Ⓐ to your local master branch Ⓒ, `git merge upstream/master` those changes, and `git push origin master` those changes to your remote master branch Ⓑ. You should then delete the local project branch Ⓓ by typing,

```
>>> git branch -d [BranchName]
```

and then delete that remote project branch Ⓔ from your remote repository by typing the following.

```
>>> git push origin --delete [BranchName]
```

3.3.3 Set the upstream remote, fetch, merge, and push

Once you have cloned your fork of the main repository, you will need a way to keep your fork updated—both your local cloned repository Ⓒ and your remote fork Ⓑ—with any changes that are made in the main repository Ⓐ. You will first want to designate a remote repository from which to draw code changes. `Git` designates your fork Ⓑ of the main repository as *origin*. Designate the main repository as the remote for your fork by opening your terminal in your local machine, navigate to the main directory of your local clone, and type the following code,

```
>>> git remote add upstream [main repo Git URL]
```

where `[main repo Git URL]` is the address that you copy when you go to the main repo main page and click on the green "Clone or Download" button, shown in Figure 3.2 under the "Fork" button.

Naming the main repository Ⓐ "upstream" in your local clone Ⓒ makes the commands easier to write that execute the updating step displayed as labeled ④ and ⑤ in Figure 3.2. Each time you come back to your local fork of the repository, you will want to check the status of your fork with respect to the remote upstream main repository Ⓐ and with respect to the remote origin fork of the repository Ⓑ. This section focuses on updating your local fork Ⓒ with new changes in the remote upstream main repository Ⓐ.

You can tell `Git` to go get any changes to the remote upstream main repository by opening your terminal, navigating to the directory of the master branch of your local fork Ⓒ, and typing the following.

```
>>> git fetch upstream
```

This command *fetches* all the changes from the upstream repository and stages them for potentially being *merged* into your local master branch Ⓒ. Note here that you are not

staging this to be added to your new project branch Ⓓ of your local repository. The purpose of your local master branch Ⓒ is to remain up-to-date with the remote master repository Ⓐ.

Once these changes are staged with the `git fetch upstream` command, you can merge those changes from the remote master repo Ⓐ into your local master branch Ⓒ using the following command.

```
>>> git merge upstream/master
```

Because of the work flow that this textbook advocates in Figure 3.2, you should have no merge conflicts with this action. Your local master branch of the forked repository Ⓒ is meant simply as a local source that receives updates from the remote main repository Ⓐ. The only other time that your local master Ⓒ is updated from another source is when it was created by cloning ② the remote master Ⓑ, which action should happen only once.

Once the changes from the remote main repository Ⓐ have been fetched and merged into your local main branch Ⓒ, you just need to *push* ⑤ those changes up to your remote master branch of your fork Ⓑ. This is done by being in your master branch and typing the following.

```
>>> git push origin master
```

The `push` command copies the changes from your local master branch Ⓒ into your remote master branch Ⓑ. The term *origin* refers to the set of branches, including the master, in your remote fork of the main repository. In Figure 3.2, repositories Ⓑ and Ⓔ are branches of the *origin* remotes. At this point, your remote master branch of your fork Ⓑ, your local master branch of your fork Ⓒ, and the main repository Ⓐ are all syncronized.

The last function we detail here is the action of *merging* changes from your local master branch Ⓒ that came from the main repository Ⓐ into your local project branch Ⓓ. This is action ⑥ in Figure 3.2. Go to the branch that will be the destination of the merge or the branch where you want to incorporate the changes. In this case, that is the local project branch Ⓓ.

```
>>> git checkout [ProjectBranch]
```

Now merge the master branch into the project branch.

```
>>> git merge master/[ProjectBranch]
```

If the merge is nontrivial, then you will get a conflict message like the following.

```
Auto-merging master.txt
CONFLICT (content): Merge conflict in master.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Now type the following to open the globally set Git mergetool (see Section 3.2 for mergetool global setting).

```
>>> git mergetool
```

Finally, you can commit the changed files and be done.

```
>>> git commit -m "Description of merge commit"
```

3.4 Git Cheat Sheet Commands

In this section, I list a number of useful Git commands that are not as frequently used.

- List the last commit for each branch in a local repository.

  ```
  >>> git branch -v
  ```

- List branches that you have or have not yet merged into your current branch.

  ```
  >>> git branch --merged
  >>> git branch --no-merged
  ```

- Undo erroneous commits in your local branch. Let [commit#] be the commit number to which you want to rewind. This will usually be a reference like f2f7281451364c29c75e07ddb3be1d8d7 Type the following.

  ```
  >>> git reset [commit#]
  ```

- Undo erroneous commits merged into your *upstream* repository. Note that it is usually thought of as bad form to erase Git history.[4] Let "upstream" be the name of the repository and "BranchName" be the branch of that repository with the offending commits. First, pull the branch with the bad commits to your local repo:

```
>>> git pull upstream [BranchName]
```

  Let [commit#] be the commit number to which you want to rewind. Rewrite the commit history on your local repo using the following command:

```
>>> git reset --hard [commit#]
```

  Now push this back up to the remote repository.

```
>>> git push -f upstream [BranchName]
```

- Create a local branch that is a copy of someone's pull request branch.

```
>>> git checkout -b [NewBranchName]
>>> git pull [PR sender branch git URL] [NewBranchName]
```

3.5 Using GitHub for Collaborative Issue Tracking

GitHub repositories have an "Issues" section that is a powerful place for collaboratively discussing and resolving issues with the code. The issues interface of a GitHub repository is accessed via the "Issues" tab in the upper-right area of the main page of the repository as shown in Figure 3.3. GitHub issues create a central remote location for resolving questions with your code. An issue creates a permanent record of what the question was, the path to resolving it, and what was the resolution. GitHub issues also serve as a non-email method of communicating about a project. This is valuable because resolution of issues can often span weeks and even months.

A good example of an effective GitHub issue is issue # 237 of the OG-USA repository. One can tag GitHub collaborators in these issues, add images and equations, and reference

[4] See Git Koan, "Only the Gods".

other issues and pull requests. This link and this link has some of the markdown options for augmenting your discussion in GitHub issues.