



F. Wermelinger
Office: Pierce 211

Pair-Programming 3

Git local branches, merge conflicts and merge tool

Issued: September 16, 2022 Due: September 30, 2022 11:59pm

In this pair-programming session you will further practice Git.

You should work on the two exercises in groups of 3 to 4 students via a tmate session. Your team members can submit the same file. Please indicate your names in a header in the files. See the tutorials on the class website for an example pair-programming workflow.¹ Do not forget to commit and push your work when you are done. Ensure that you are on your *default branch* for this and not, possibly, on your homework branch.

Exercise 1: Local Branches in Git

Deliverables:

1. Updated `README.md` file in your private course repository root. (The same file should also be placed in `lab/pp3/README.md`. If the file already exists and you want to keep it please rename it.)

Suppose you want to implement a new feature in your repository. You should create a new branch to work on that feature, but you do not want this branch to be long-lived. Once you have merged the feature into your default branch, there is no need to keep this branch any longer.

For this exercise, you will work in the root directory of your private course repository. *Make sure you are on your default branch* and perform the following steps:

1. Create a new branch called `feature_branch` based off of your default branch and switch to this branch.
2. Type `git branch` or `git status` to ensure you are on the desired branch.
3. Edit the `README.md` file in your repository root. Create a new one if it does not exist already.

¹<https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-pp>

4. Add a main title with the name of the class followed by a lower level header called "Content". Add a bullet list in the "Content" section that lists the directories in the root of your repository. For example homework, lab and lecture. Provide a one sentence description for each of the listed items.
5. Type `git status` to inspect the state of your repository after you have made these changes.
6. Stage and commit your modifications.

After these steps you have committed your new feature in your local `feature_branch`. Assuming that the code has been tested and works according to the specification, you would like to merge the changes into your default branch. To perform the merge, you need to change to your default branch first (e.g., `main` or `master`):

1. Switch to your default branch.
2. Inspect the `README.md` file (in case it exists). You can use the `cat` or `vim` commands for example. As you expected, the file contents should be *different* compared to your `feature_branch`.
3. Now *merge* the `feature_branch` into your default branch.
4. Spend some time to analyze the Git output after the merge has concluded. Try to make the following observations:
 - What does the Git output tell you?
 - Perform a `git status` again. What do you observe regarding your local default branch and its tracked remote (presumably named `origin`)?
 - Inspect the contents of the `README.md` file again. Execute the command

```
$ git diff HEAD~..
```

to support your reasoning.

The merge brought the state of the default branch to the state of the `feature_branch`. The old branch head (i.e. before the merge happened) is now reachable at `HEAD~1` (you could also omit writing the digit "1" or use larger digits to reach further back, see SPECIFYING REVISIONS in `git help rev-parse`). This notation means "the first parent of the branch `HEAD`". All the changes are now integrated in the default branch, there is no need to keep the `feature_branch`:

1. Delete the `feature_branch`.
2. Push your changes to the remote repository.

Great! Now you should have a basic understanding of how to work with local branches. Understanding branches is the most essential concept when working with Git. In addition to this exercise, it is highly recommended to read Chapter 3 in the Git book² for more details and workflow ideas.

Please see [solution/exercise_1.sh](#) for a scripted version of the steps performed in this exercise.

²<https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>

Exercise 2: Git Merge Conflicts

Deliverables:

1. [lab/pp3/exercise_2.png](#)

You already have experience with Git merge conflicts from the first homework assignment. Let us do one more for practice. Please ask your TF if some of the steps below are not clear to you.

To practice Git it is always a good idea to setup local dummy repositories such that we are working in a safe environment. For this exercise, the [lab/pp3/data/setup.sh](#) script is provided for you in order to setup the initial state for the exercise. Running the script will create a parent directory called `exercise_2` with three Git repositories `remote`, `A` and `B`, where the former is a bare Git repository used for the remote origin. To create these repositories, simply execute the `setup.sh` script. Note that since this is a sandbox environment, you should not add the created data to your private class repository (simply remove the `exercise_2` directory when done or add it to your `.gitignore` file).

The story goes like this: two developers A and B work on some Python code in `goat.py`. They both committed changes to parts of the code. Developer B has pushed the changes already. Change into developer A's repository, using `cd A` for example. The coder steps into the shoes of developer A and another team mate takes developer B's role. Try to push your local changes to the remote using

```
$ git push
```

you should see a rejection by Git because it appears that the remote has changed since your last local changes. To update your local repository you have two choices: you can *fetch* the remote and then *merge* the remote branch into your local branch. The alternative is the *pull* shortcut, this simply combines the fetch and merge into one command since these two operations are very frequently used in this sequence. Hence, before you start changing code, it is always a good idea to update your local repository first. Try the following

```
$ git fetch origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 286 bytes | 286.00 KiB/s, done.
From ../remote
    cc1714f..dbf9057  main      -> origin/main
```

Discuss the output with your team mates. In particular, try to understand the last line. Have a look at `git log` in the repository you are in and compare the commit references. The fetch command has *not* updated your local branch but only `origin/main`. Try to understand this by issuing the following

```
$ cat .git/refs/remotes/origin/main
dbf90578b74a1b4e7c68bf4f15ec6d4fbcfc477d
git rev-parse HEAD
668483e6823f1cddb35ae3145358c618fdc7349c
git rev-parse HEAD~1
cc1714fb77f2520ff83c8a55e0e952ea58aff496
```

To update your local branch you need to merge

```
$ git merge origin/main
```

If you had issued a `git pull` instead of the `git fetch` above you would be taken to this state directly. Git will let you know that there is a merge conflict it cannot resolve and requests your action. The troublesome file is

```
$ cat goat.py
#!/usr/bin/env python3
# File      : goat.py
# Description: pp3 exercise 2
# Copyright 2022 Harvard University. All Rights Reserved.

def main():
<<<<<<< HEAD
    goat = 4 * ["Developer A is the greatest for all eternity!"]
=====
    goat = 100 * ["Developer B is the greatest of all time!"]
>>>>>> origin/main
    print('\n'.join(goat))

if __name__ == "__main__":
    main()
```

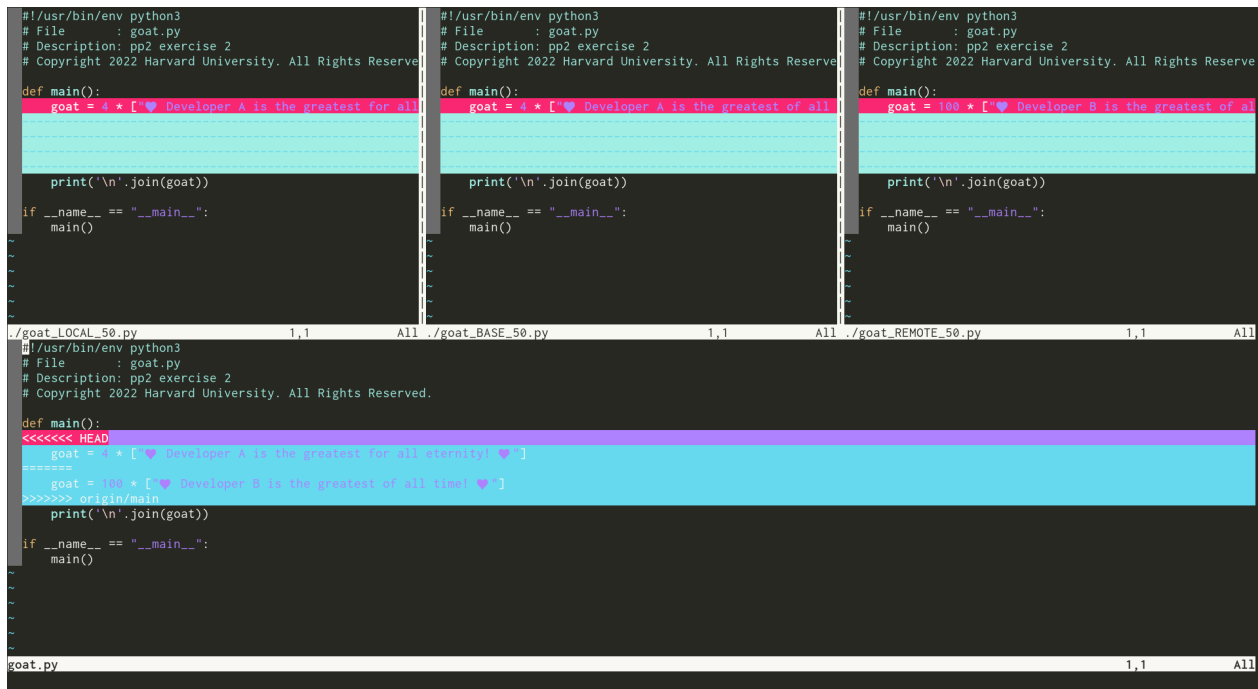
Recall from homework 1 what the markers mean. You are already familiar how you could solve this conflict. Git offers a tool for more complex merge conflicts that can be used to aid resolution. Different tools can be configured (see `git help mergetool`), here we are using vim. Configure the merge tool with

```
$ git config merge.tool vimdiff # add --global option to put in ~/.gitconfig
```

and then call the tool with

```
$ git mergetool
```

You should see vim opening up with several *splits* containing different file states of the conflicting file. If you are using the Docker container you should see what is shown in



```
#!/usr/bin/env python3
# File      : goat.py
# Description: pp2 exercise 2
# Copyright 2022 Harvard University. All Rights Reserved.

def main():
    goat = 4 * ["♥ Developer A is the greatest of all"]

    print('\n'.join(goat))

if __name__ == "__main__":
    main()

~/goat_LOCAL_50.py 1,1 All

#!/usr/bin/env python3
# File      : goat.py
# Description: pp2 exercise 2
# Copyright 2022 Harvard University. All Rights Reserved.

def main():
    goat = 4 * ["♥ Developer A is the greatest of all"]

    print('\n'.join(goat))

if __name__ == "__main__":
    main()

~/goat_BASE_50.py 1,1 All

#!/usr/bin/env python3
# File      : goat.py
# Description: pp2 exercise 2
# Copyright 2022 Harvard University. All Rights Reserved.

def main():
    goat = 100 * ["♥ Developer B is the greatest of all"]

    print('\n'.join(goat))

if __name__ == "__main__":
    main()

~/goat_REMOTE_50.py 1,1 All

~/goat.py 1,1 All
```

FIGURE 1: Git merge tool configured for vim.

figure 1 The three splits on the top correspond to the file state that Git indicates with the markers in the conflicting file. The left window contains the local file state, middle the state of the base (common ancestor) and the right window contains the state of the file on the remote. The bottom window contains the file state of the current merge commit that needs to be resolved. Fix this conflict by communicating with your team mates what the solution should be. You can rotate windows in vim by pressing `Ctrl-w w`, `Ctrl-w =` equalizes all windows, `:cq!` exits the editor without saving changes and `:wqa` saves your merge resolution and exits vim. Note that if there were multiple conflicting files, upon exiting the merge tool, Git will automatically open the merge tool with the next conflicting file. This allows to resolve conflicts efficiently. In addition, exiting the merge tool adds the conflicting file to the index, you do not need to perform `git add` as in manual resolution. Note that `vimdiff` may leave a `.orig` file in the working tree. This is a copy of the original file before the conflict resolution and can be removed afterwards.

Finally, commit the conflict resolution with

```
$ git commit -m 'Fixed merge conflict'
$ git log
```

Take a screenshot of the output of `git log` and save the file in `exercise_2.png`. You can now push the resolved conflict with

```
$ git push
```

If you change into the B repository and invoke `git pull` all changes will be applied in the same sequence as developer A did before and, therefore, developer B will no longer run into a conflict.

A note about binary files: binary files are not well fitted for version control because Git treats them as a single blob. Whenever a binary file changes, Git must store the *whole* blob again in the history (not just the diff). A common example are PDF files that are under version control. Such binary files are on average larger than simple text files. If the binary files do not change once added it is not a big deal but your `.git` directory can become *very large in size* if you often commit changes for binary files. The general rule is to refrain from putting binary files under version control and stick with source code mainly.

Please see [solution/exercise_2.png](#) for the solution image.

Many Git commands accept commit *ranges* as an argument. The last line in the output of the fetch command above:

```
$ git fetch origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 286 bytes | 286.00 KiB/s, done.
From ../remote
    cc1714f..dbf9057  main      -> origin/main
```

describes a range of commits that have changed on the `origin/main` branch, which happens to be a remote in this example since the `fetch` command is used to update remotes locally. The syntax is `<rev1>..<rev2>` where `<rev1>` and `<rev2>` are *revisions* (e. g. a commit reference or HEAD). Such range specifies that all commits *reachable* from `<rev2>` should be included and all commits reachable from `<rev1>` should be excluded (these are usually commits further in the past). If either `<rev1>` or `<rev2>` is omitted, then HEAD is used instead. There are other ways to specify a range. See the SPECIFYING REVISIONS, SPECIFYING RANGES and REVISION RANGE SUMMARY in `git help rev-parse` for an excellent documentation (it is recommended to re-read these sections every 6 months).

Such revision ranges are useful when you want to inspect a *diff* between two revisions. After the fetch above (before entering the merge phase) you can inspect just the diff of the revision range and see what has changed without Git telling you that there is actually a merge conflict ahead. For this you can simply use

```
$ git diff cc1714f..dbf9057
diff --git a/goat.py b/goat.py
index 106cd13..176b5f6 100644
--- a/goat.py
+++ b/goat.py
@@ -4,7 +4,7 @@
    # Copyright 2022 Harvard University. All Rights Reserved.
```

```
def main():
-   goat = 4 * ["Developer A is the greatest of all time!"]
+   goat = 100 * ["Developer B is the greatest of all time!"]
    print('\n'.join(goat))

if __name__ == "__main__":
```

Note that some unicode characters used in the original file `goat.py` are omitted above for clarity. Moreover, since you work through this exercise at a different date and time, your commit references will be different than the ones shown here.

The way you should read the diff is “what changed *from* revision `cc1714f` to revision `dbf9057`?”. Lines prefixed with a `-` sign (highlighted in red) have been removed and lines prefixed with a `+` sign have been added throughout the range of commits. Since you execute this command as developer A, you can reason that developer B has changed some code you have previously worked on. You can already sense here that this will cause a merge conflict as you have *another commit in your local history* that is not visible at `origin/main` yet (only once you merge `origin/main`). Finally, you can also *reverse* the range to inspect the inverse change (e.g. what happens when a `git revert` is applied):

```
$ git diff dbf9057..cc1714f
diff --git a/goat.py b/goat.py
index 176b5f6..106cd13 100644
--- a/goat.py
+++ b/goat.py
@@ -4,7 +4,7 @@
 # Copyright 2022 Harvard University. All Rights Reserved.

def main():
-   goat = 100 * ["Developer B is the greatest of all time!"]
+   goat = 4 * ["Developer A is the greatest of all time!"]
    print('\n'.join(goat))

if __name__ == "__main__":
```
