11
Forking Workflow Written by Jawwad Ahmad

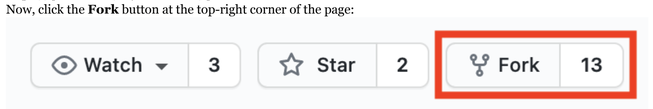
In this chapter, you'll learn all about the **Forking Workflow**. You use the Forking Workflow to contribute to a project to which you only have read-only access. It's mainly used when contributing to open-source projects, but you can also use it with private repositories.
When you don't have push access to a project, you'll need to push your changes to a public copy of the project. This personal, public copy of the project is called a **fork**. The original or source repository is conventionally referred to as the **upstream** repository.
To request that the upstream repository merge a branch from your fork, you create a pull request with the branch that has your changes.
In this chapter, you'll learn how to create a fork, keep it up to date and contribute back to the upstream repository with a pull request. You'll also learn how to merge in open pull requests and branches from other forks.

Getting started

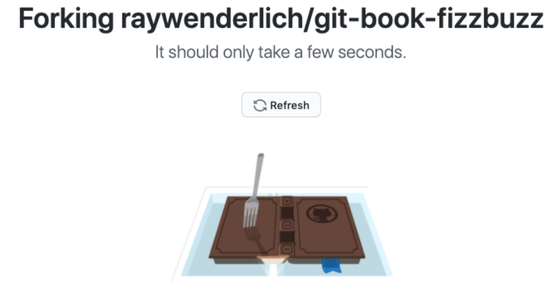
As a software developer, you've likely heard of FizzBuzz. In case you haven't, it's a programming task where, for numbers from 1 to 100, you print either the number itself or a word. For multiples of three, you print **Fizz**, for multiples of five, you print **Buzz**, and for multiples of both three and five, you print **FizzBuzz**.
For example, here are the first fifteen items:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Buzz
13
14
FizzBuzz
```

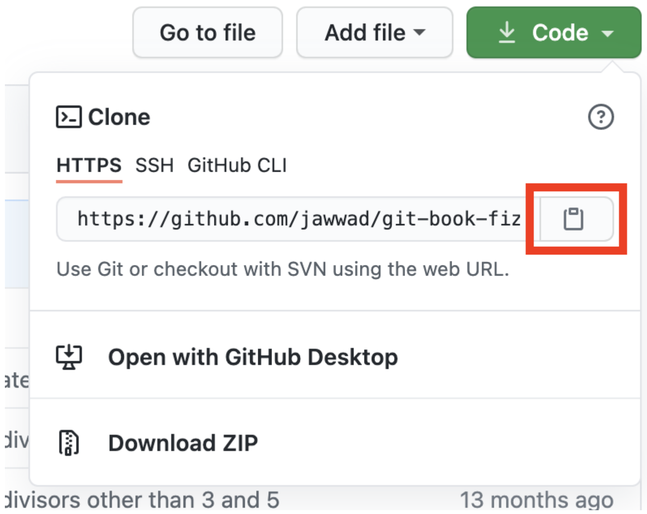
For this tutorial, you'll create a fork of a repository that implements FizzBuzz. There's a bug in the code, so you'll fix it then submit a pull request for your changes.
In a browser, open the following URL for the repository:
https://github.com/raywenderlich/git-book-fizzbuzz
Now, click the **Fork** button at the top-right corner of the page:



You'll see a progress screen indicating that GitHub is creating your fork:



Unfortunately, GitHub no longer shows the fork-in-a-book-on-a-copier image :[
Once GitHub finishes, it will redirect you to the newly-created fork under your personal GitHub account. You'll see the URL of the page change to `https://github.com/{your-github-username}/git-book-fizzbuzz`.
Next, click on the **Code** button drop-down, then click the clipboard icon to copy the repository's URL:



Now, open Terminal and **cd** to the **starter** folder of this project:
```
cd path/to/projects/starter
```

Next, type **`git clone`**, add a **space** and **paste** the copied repository URL.
You should have the following, with your GitHub username in place of `{username}`:
```
git clone https://github.com/{username}/git-book-fizzbuzz.git
```

Press **Enter** to execute the command. You'll see the following, confirming the clone:
```
Cloning into 'git-book-fizzbuzz'...
...
...
Resolving deltas: 100% (15/15), done.
```

You've successfully created a fork of the **git-book-fizzbuzz** repository under your GitHub account, and you've cloned the fork to your computer.
Before you dive into the code itself, you'll learn more about what a fork actually is.

A fork is simply a clone

In the previous section, you created a fork and then cloned it. So if a fork is just a clone, then you cloned your clone!
More specifically, a fork is a public, server-side clone of the project under your own account, which means you can push changes to it.
Forking is a workflow and not part of Git itself. There's no *git fork* command that will create a fork of a repository. When you create a fork in GitHub, it creates a server-side clone of the project under your account and enables certain features available only to forks, like the ability to create pull requests.
As far as Git is concerned, there's no difference between the upstream repository, your fork of the repository, and the local clone of your fork.
To help you internalize this, you'll create another clone directly from the upstream repository. This will show you how an upstream clone differs from a clone of the fork.
Make sure you're still in the **starter** folder and run the following command, all on a single line, to clone the upstream repository as **upstream-git-book-fizzbuzz**:
```
git clone https://github.com/raywenderlich/git-book-fizzbuzz.git upstream-git-book-fizzbuzz
```

Now, run the following to compare the clone of your fork with the clone of the upstream:
```
diff -r git-book-fizzbuzz upstream-git-book-fizzbuzz -x logs -u
```

Note: `-x logs` is short for `--exclude=logs`, which ignores timestamps in the logs directory. `-u` is short for `--unified`, which shows the diff in unified format — that is, with a - and + instead of < and >.
In the results, you'll see the following, which shows that the only difference is the remote origin URL:
```
...
 [remote "origin"]
-    url = https://github.com/{username}/git-book-fizzbuzz.git
+    url = https://github.com/raywenderlich/git-book-fizzbuzz.git
...
Binary files git-book-fizzbuzz/.git/index and upstream-git-book-fizzbuzz/.git/index differ
```

The final line, telling you the `.git/index` files of the two branches are different, is inconsequential since this is a binary Git uses to keep track of staged changes.
You can even update the clone of the upstream repository to point to your fork by updating its origin URL.
Run the following, replacing `{username}` with your GitHub username:
```
cd upstream-git-book-fizzbuzz
git remote set-url origin https://github.com/{username}/git-book-fizzbuzz.git
```

Run `cd ..` to go back to the starter folder, then execute the diff command again:
```
cd ..
diff -r git-book-fizzbuzz upstream-git-book-fizzbuzz -x logs -u
```

Now, you'll no longer see any differences other than the binary `.git/index` file.
With this, you see that there's absolutely no difference between a clone of your fork and a clone of the upstream repository other than the **origin** URL.
Now, delete the **upstream-git-book-fizzbuzz** clone since you no longer need it:

```
rm -rf upstream-git-book-fizzbuzz
```

Next, you'll explore the code and then play a quick game of *find-the-bug*!

Exploring the code

Change to the **git-book-fizzbuzz** directory and open **fizzbuzz.py** in an editor.

```
cd git-book-fizzbuzz
open fizzbuzz.py  # or open manually in an editor of your choice
```

Start reading from the end of the file. The following lines mean that the `main()` method is executed when running this as a script:

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

And `main()` simply executes `fizzbuzz()`:

```
def main():
    fizzbuzz()
```

And above that, `fizzbuzz()` executes `fizzbuzz_for_num(n)` for each number n from 1 to 101 exclusive, which really means from 1 to 100.

```
def fizzbuzz():
    for n in range(1, 101):
        value = fizzbuzz_for_num(n)
        print(value)
```

Finally, `fizzbuzz_for_num(...)` contains the logic that determines which string to return for a given number. It additionally allows using words other than **Fizz** and **Buzz**, and even allows you to use divisors other than **3** and **5**:

```
def fizzbuzz_for_num(
    n,
    fizz_divisor=3,
    fizz_word="Fizz",
    buzz_divisor=5,
    buzz_word="Buzz",
):
    should_fizz = n % 3 == 0
    should_buzz = n % 5 == 0
    if should_fizz and should_buzz:
        return fizz_word + buzz_word
    elif should_fizz:
        return fizz_word
    elif should_buzz:
        return buzz_word
    else:
        return str(n)
```

There is, however, a bug in the code above. See if you can spot it. The bug only manifests itself when using divisors other than 3 and 5.
If you haven't spotted it already, you certainly will when you look at the contents of the 85ca623 commit which added the bug.
Run **`git show 85ca`** to see the contents of that commit. In **fizzbuzz.py**, you'll see the following change:

```
 def fizzbuzz_for_num(
     n,
+    fizz_divisor=3,
     fizz_word="Fizz",
+    buzz_divisor=5,
     buzz_word="Buzz",
 ):
```

The commit added the `fizz_divisor` and `buzz_divisor` parameters to the method signature but never updated the code in the method itself to use the new parameters! Next, you'll fix this bug and open a pull request for it.
The second thing to notice in the `git show` output is that the commit also added tests to **test_fizzbuzz.py** in the **`test_with_alternate_divisors`** method.
Run the tests with the following command:

```
python test_fizzbuzz.py
```

You'll see the following three failures in the output:
Note: If you see an error that says: AttributeError: 'TestFizzBuzz' object has no attribute 'subTest', this means your default Python is Python 2. Try running `python3 test_fizzbuzz.py`; if that doesn't work, you can skip this step.

```
...
    self.assertEqual(fizzbuzz_for_num(7, fizz_divisor=7, buzz_divisor=11), "Fizz")
AssertionError: '7' != 'Fizz'
- 7
+ Fizz
...
...
    self.assertEqual(fizzbuzz_for_num(11, fizz_divisor=7, buzz_divisor=11), "Buzz")
AssertionError: '11' != 'Buzz'
- 11
+ Buzz
...
...
    self.assertEqual(fizzbuzz_for_num(77, fizz_divisor=7, buzz_divisor=11), "FizzBuzz")
AssertionError: '77' != 'FizzBuzz'
- 77
+ FizzBuzz
...
...
Ran 6 tests in 0.001s

FAILED (failures=3)
```

The output above is saying that given `fizz_divisor=7`, and `buzz_divisor=11`:

. For number 7, it should have returned **Fizz**, but it returned 7.

. For number 11, it should have returned **Buzz**, but it returned 11.

. For number 77, it should have returned **FizzBuzz**, but it returned 77.

It's nice that the commit also included tests, but it looks like someone forgot to run them to verify that their code *actually* worked. :[
Though it's certainly helpful that there are tests so you can verify that your upcoming fix will work! :]

Fixing the custom divisors bug

Create a new branch for your fix named **fix-divisors-bug**:

```
git checkout -b fix-divisors-bug
```

Switch back to the editor where you have **fizzbuzz.py** open. On line 11, replace **3** with **fizz_divisor** and on line 12 replace **5** with **buzz_divisor**:

```
11)     should_fizz = n % 3 == 0  # replace 3 with fizz_divisor
12)     should_buzz = n % 5 == 0  # replace 5 with buzz_divisor
```

After saving the file, run `git diff` and confirm that you see only the following changes:

```
...
-     should_fizz = n % 3 == 0
-     should_buzz = n % 5 == 0
+     should_fizz = n % fizz_divisor == 0
+     should_buzz = n % buzz_divisor == 0
      if should_fizz and should_buzz:
...
```

And now for the moment of truth! Execute the tests with the following command:
`python test_fizzbuzz.py`

This time, all the tests should pass! :]
```
test_divisible_by_both (__main__.TestFizzBuzz) ... ok
test_divisible_by_five (__main__.TestFizzBuzz) ... ok
test_divisible_by_none (__main__.TestFizzBuzz) ... ok
test_divisible_by_three (__main__.TestFizzBuzz) ... ok
test_with_alternate_divisors (__main__.TestFizzBuzz) ... ok
test_with_alternate_words (__main__.TestFizzBuzz) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
```

Now, you can commit your changes.
It's a good idea for the commit message for your pull request to detail why you made the changes, how you fixed the bug, and how you tested your fix.
However, typing out long paragraphs in a tutorial is no fun. Instead, you'll use the message in **commit_message.txt**, which is one level up in the **starter** folder.
However, you don't need to open the file manually. Run the following command to commit your changes with the message taken from the **commit_message.txt** file.
`git commit -a --file=../commit_message.txt`

Now, run **`git show`** to show the content of your latest commit. You should see the following as the message for your commit:
```
Fix bug in which alternate divisors were not used

This commit updates the fizzbuzz_for_num method to start using
the fizz_divisor and buzz_divisor parameters that were added
to the method signature in 85ca623.

Verified the fix by running existing tests in test_fizzbuzz.py
which were previously failing and now pass.
```

Next, you'll push the **fix-divisors-bug** branch to your fork so you can open a pull request with it.

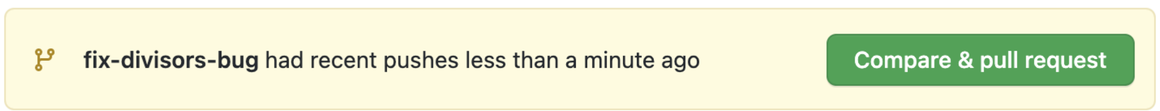
Opening a pull request

Run the following to push the current branch to your fork:
`git push -u origin head`

Specifying `head` tells Git to push the current branch, so the above is shorthand for:
`git push --set-upstream origin fix-divisors-bug  # same as above`

Now that the branch is available in your fork, there are a few different ways to reach the pull request creation page. The following are three ways that you can use:

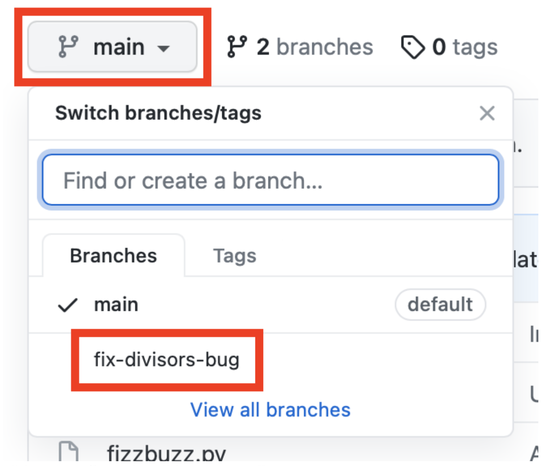
. If you see a banner similar to the following appear on the GitHub page for your fork, you can click on the **Compare & pull request** button. Sometimes the banner doesn't appear, so you can't always count on it.



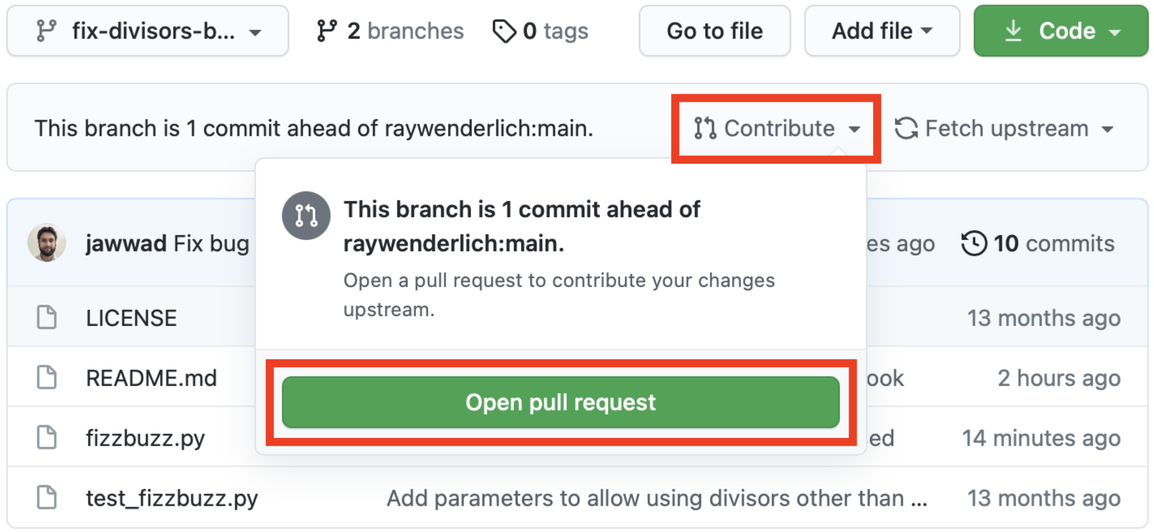
. If you look at the output of the previous `git push` command, you should see the following lines within the section prefixed with `remote:` that say:
```
...
remote: Create a pull request for 'fix-divisors-bug' on GitHub by visiting:
remote:      https://github.com/{username}/git-book-fizzbuzz/pull/new/fix-divisors-bug
...
```

You can open the URL listed above to go directly to the pull request creation page.

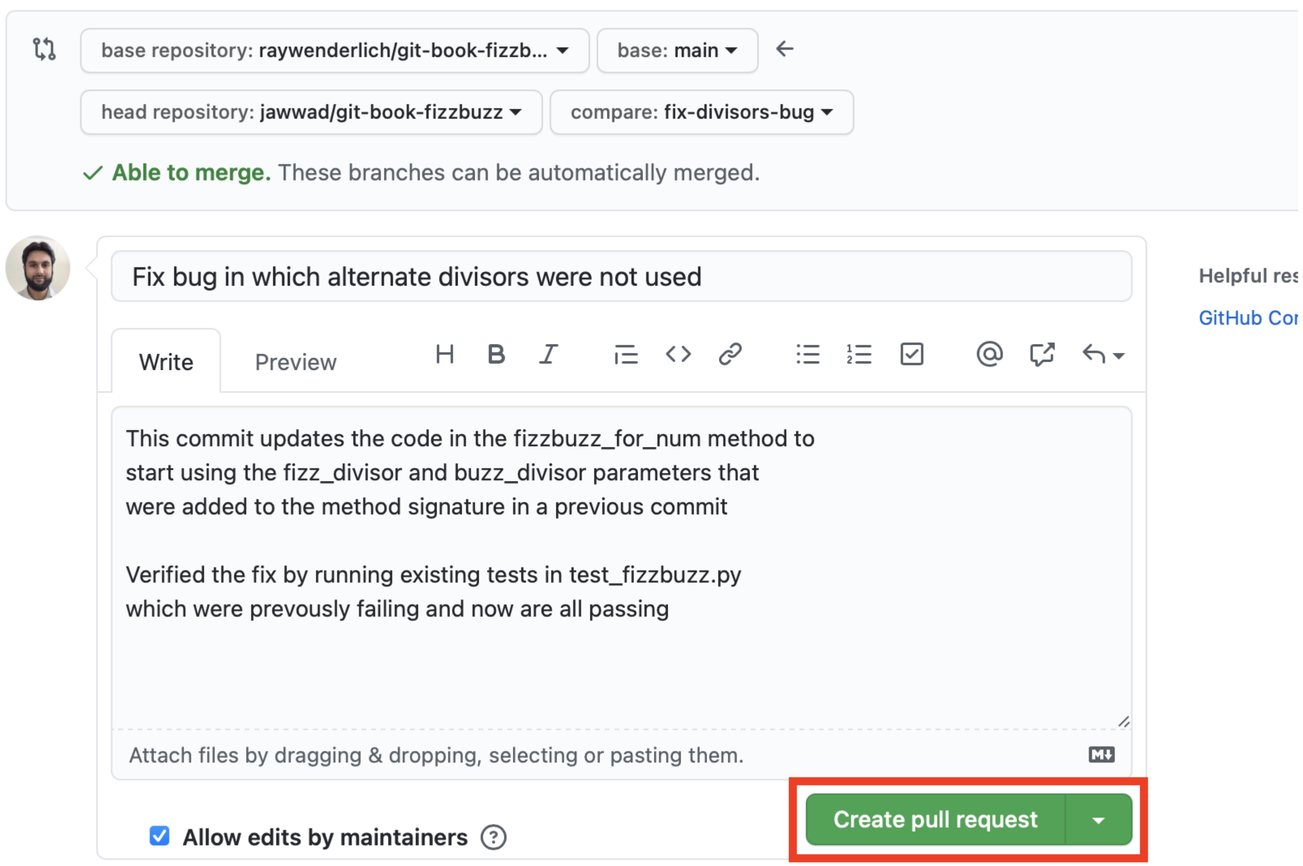
. Yet another way is to click the **branches** drop-down on the GitHub page for your fork and select the **fix-divisors-bug** branch



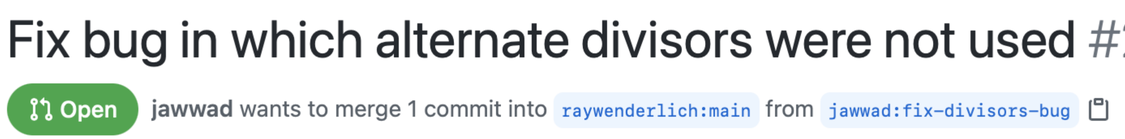
Click on **Contribute**, then click on **Open pull request**:



Each of these three methods will take you to the same **Open a pull request** page.
GitHub helpfully uses the commit message's first line as the title of the pull request and the remaining lines as the body.
Finally, click on the **Create pull request** button to finish creating the pull request:



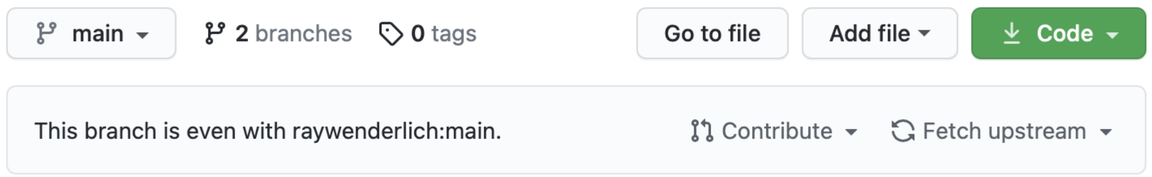
You've now created your pull request:



At this point, you'd normally just sit back, relax and wait for the maintainer of the upstream repository to merge your pull request. However, in this case, you still have the rest of the chapter to finish!
Although your pull request is amazing, I have a feeling that the maintainer of the upstream repository won't merge it. If he did, it would remove the bug that is needed for other readers of this chapter! But feel free to leave it open since it lets me know you've read this chapter!
Next, you'll learn how to keep your fork up to date with any additional changes pushed to the main branch of the upstream repository.

Rewinding your main branch

Unfortunately, there won't be any updates to the upstream repository from the time you cloned (or perhaps ever!), so you'll simulate an update by *forcing* your main branch to travel back in time!
Go back to your fork in GitHub, since creating the pull request would have taken you to the upstream **raywenderlich/git-book-fizzbuzz** repository. Also, make sure to switch back to the **main** branch in the drop-down.
First, note where it says: **This branch is even with raywenderlich:main.**



Now, run the following commands in Terminal to switch to your **main** branch and reset it back by two commits:

```
git checkout main
git reset head~2 --hard
```

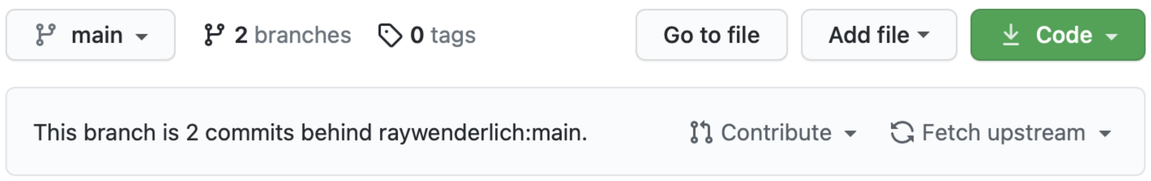
You'll receive confirmation that the branch has been reset:

```
HEAD is now at 8034fbf Add option to use words other than Fizz and Buzz
```

You also want to push this change to your fork. To do this, you'll do something that you were told never, ever to do... you'll force push the main branch! In this case, it's ok to do this since no one else would really be using your fork's main branch.

```
git push -f origin main
```

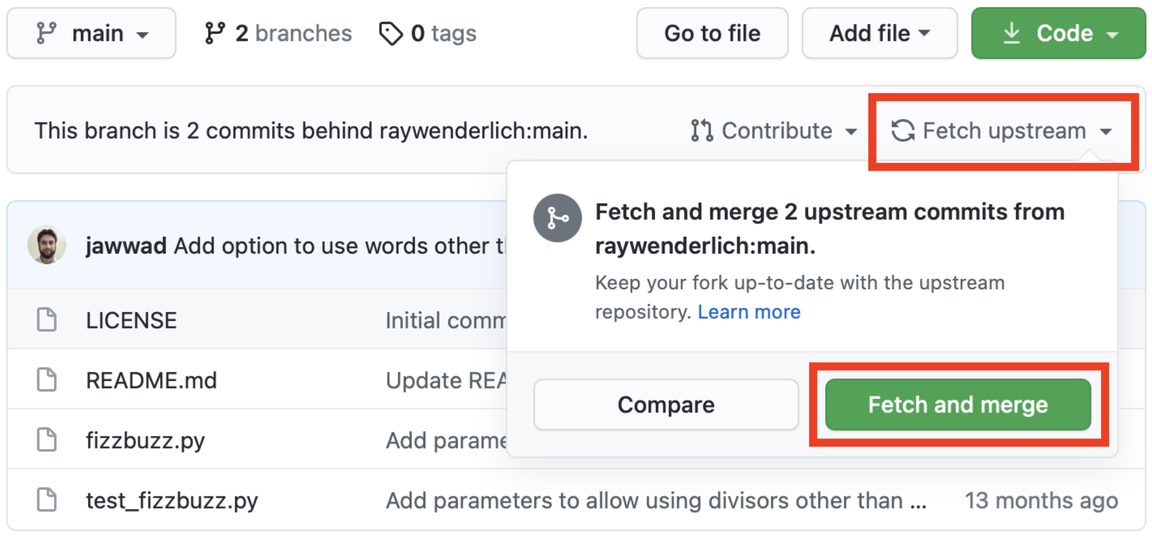
Note: Just running `git push -f` would have accomplished the same thing. However, it's good practice to always specify the branch when you're force pushing so that you don't accidentally force push the wrong branch.
Refresh the page in GitHub, and you'll see that it now says: **This branch is 2 commits behind raywenderlich:main.**



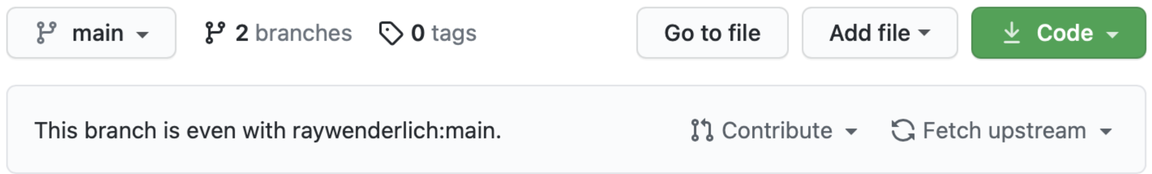
Time travel complete! Now you can pretend that the upstream repository has two new commits since you forked the repository.
Remember that there are two Git repositories that you'll now want to update with the new commits. One is your fork on GitHub, and the other is the local clone of your fork.
There are two ways you can do this. You can either update your fork first and then pull those updates into your local clone. Or you can update your local clone directly from the upstream repository and then push those updates to your fork.
The easier way is to update your fork directly from GitHub, so you'll do that. But you'll optionally learn how to do it the other way as well.

Updating your fork via GitHub

GitHub recently added the option of updating your fork directly from the upstream repository.
Click on the **Fetch upstream** drop-down and then click on **Fetch and merge**.



Once the merge is complete, it will again say: **This branch is even with raywenderlich:main.**



Finally, in your Terminal, run **`git pull`** to fetch and merge those additional updates from your fork:

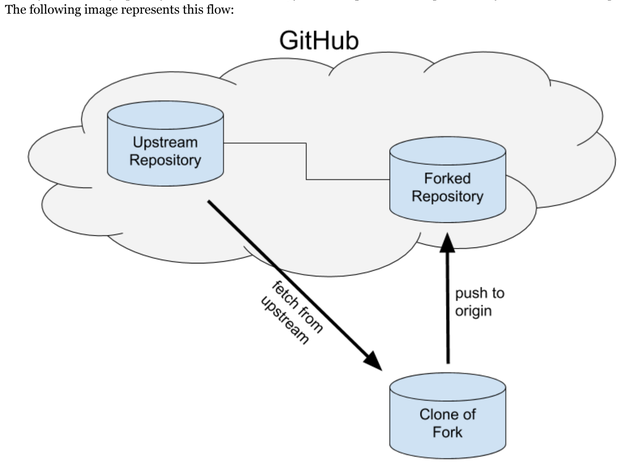
```
$ git pull

From https://github.com/{username}/git-book-fizzbuzz
   8034fbf..98b4ef3  main       -> origin/main
Updating 8034fbf..98b4ef3
Fast-forward
 README.md        |  4 ++--
 fizzbuzz.py      |  2 ++
 test_fizzbuzz.py | 10 ++++++++++
 3 files changed, 14 insertions(+), 2 deletions(-)
```

That's all there is to it! Your fork on GitHub and the local clone of your fork are now in sync with the upstream repository.

Updating your fork using an upstream remote

This method of updating is slightly more involved, but until very recently, it was the only way you could do so. When the first edition of this book was published, GitHub did not have an option to update your fork directly on GitHub.
In fact, I wrote the following paragraph in the first edition of the book:
GitHub is nice and lets you know that your fork's main branch is two commits behind raywenderlich.com's main branch. But it doesn't actually give you a server-side option of updating your branch directly from upstream. Clicking a button would be too easy, right? :]
I'm going to pretend that someone from GitHub saw that and then decided to prove me wrong!
Since you can easily update your fork from GitHub, you can skip the following section if you'd like. But it's a good opportunity to learn how to add an additional remote named **upstream** to your fork so that your local clone can pull changes from upstream directly.
The following image represents this flow:



In this flow, you'll first fetch and merge changes from `upstream` into the local clone of your fork. You'll then push those updates to your fork on GitHub.
First, you'll have to perform that time travel again to rewind your main branch so that there are updates that you can fetch from upstream.
Just like before, run the following commands in Terminal to switch to your **main** branch and reset it back by two commits:

```
git checkout main
git reset head~2 --hard
```

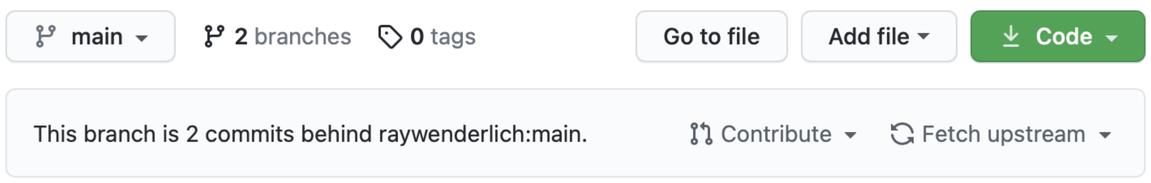
You'll see confirmation that the branch has been reset:

```
HEAD is now at 8034fbf Add option to use words other than Fizz and Buzz
```

And, once again, force push those changes to your fork on GitHub:

```
git push -f origin main
```

Refresh the page for your fork on GitHub, and it should again say: **This branch is 2 commits behind raywenderlich:main.**



Now you're ready to start the update process.
First, you'll need to connect the local clone of your fork directly to the upstream remote. You'll do that by adding an additional remote named **upstream** to your local clone. Run the following command to add it:

```
git remote add upstream https://github.com/raywenderlich/git-book-fizzbuzz.git
```

Now, run `git remote -v` to list the remotes. You'll see the following:

```
origin  https://github.com/{username}/git-book-fizzbuzz.git (fetch)
origin  https://github.com/{username}/git-book-fizzbuzz.git (push)
upstream    https://github.com/raywenderlich/git-book-fizzbuzz.git (fetch)
upstream    https://github.com/raywenderlich/git-book-fizzbuzz.git (push)
```

Next, run the following to fetch updates from the **upstream** remote:

```
git fetch upstream
```

Since you added a remote named **upstream**, running `git fetch upstream` created a *remote tracking branch* named **upstream/main** that will update any time you run it again.

```
From https://github.com/raywenderlich/git-book-fizzbuzz
 * [new branch]      main       -> upstream/main
```

Now, run **`git log --oneline --all`** and you'll see that **upstream/main** is two commits ahead of **main** and **origin/main**:

```
061a436 (origin/fix-divisors-bug, fix-divisors-bug) Fix bug in which alterna...
98b4ef3 (upstream/main) Update README to reflect updated name of book
85ca623 Add parameters to allow using divisors other than 3 and 5
8034fbf (HEAD -> main, origin/main, origin/HEAD) Add option to use words oth...
...
```

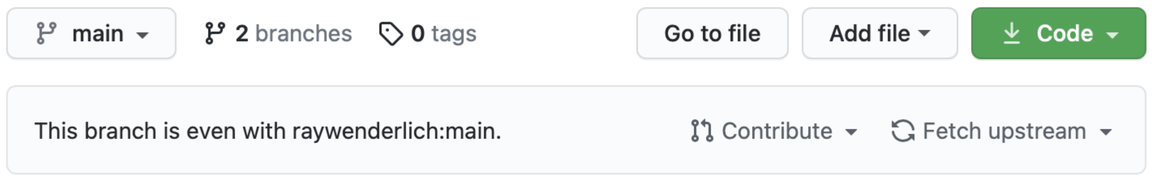
Now, merge **upstream/main** into **main** by running the following:

```
git merge upstream/main
```

Note: The fetch and merge were done separately for demonstration purposes, but you can also just run `git pull upstream main`, which is a shortcut for running the `git fetch upstream` and `git merge upstream/main` commands.
Finally, push the updated branch to your fork:

```
git push
```

Again, refresh the GitHub page for your fork, and you'll see that it once again says: **This branch is even with raywenderlich:main**.



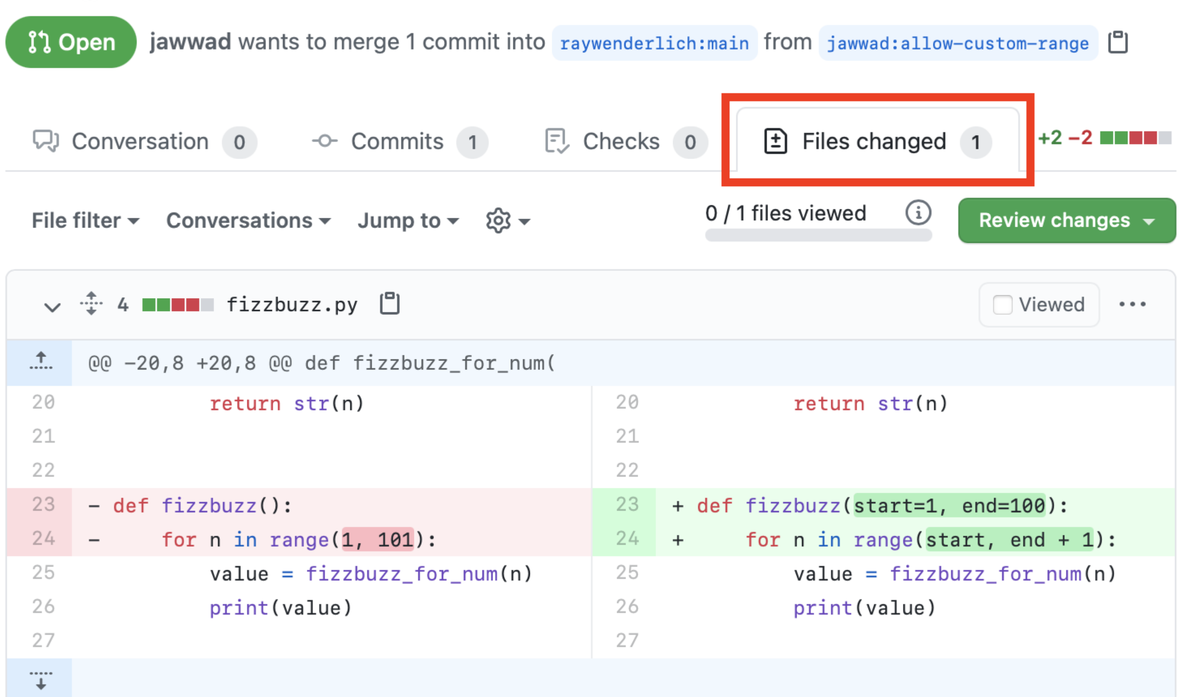
Congratulations! You've updated your fork's **main** branch with the two additional commits from the upstream's **main** branch. And a double congratulations if you did it the second optional way as well!

Fetching changes from other forks

You may occasionally want to merge feature branches from other forks into your fork. Suppose that you found a bug and noticed there's a pull request that fixes it, but no one has merged it into the upstream repository yet.
In this case, you can merge the branch from the pull request into your fork. It's not recommended that you merge anything other than **upstream/main** into your fork's **main** branch since your **main** branch should always mirror the upstream's **main** branch.
Run the following command to create a new **development** branch and merge your **fix-divisors-bug** branch into it:

```
git checkout -b development
git merge fix-divisors-bug
```

If you add an additional remote, fetching branches from that repository becomes easy. Running `git fetch {remotename}` will fetch all the remote branches and create *remote tracking* branches in the format **{remotename}/{branchname}**.
If you want to fetch a single branch from a different fork, adding the fork as an additional remote is unnecessary. You'd normally add remotes for forks that you want to fetch from more than once.
The feature branch you'll fetch already has a pull request open for it. It's for a minor feature that adds the ability to have **fizzbuzz.py** print a custom range instead of always using 1 to 100.
Navigate to the following page to see the pull request:
https://github.com/raywenderlich/git-book-fizzbuzz/pull/3
You'll see that it says: "**jawwad** wants to merge 1 commit into `raywenderlich:main` from `jawwad:allow-custom-range`". This tells you that the changes you want to pull are from jawwad's fork from the **allow-custom-range** branch.
Click the **Files changed** tab to see the included changes:



Note: Your view may differ slightly based on your settings and updates to GitHub. If you don't see the changes side by side and you'd like to, you can select the gear icon (to the right of **Jump to**) and select the **Split** view instead of **Unified**. Feel free to toggle between the two to see the difference.
This looks like a fairly simple update and something that might come in handy, so you'd like to merge it into your development branch.
There are three ways to do this. You can:

. Fetch changes directly from the other fork using its repository URL.

. Fetch changes from upstream using a special pull request reference.

. Add the other fork as an additional remote.
Next, you'll try out the first way by using the other fork's repository URL directly.

Fetching directly from a URL

To fetch from a URL, just use that URL in place of the remote name. So, for example, instead of `git fetch upstream`, you'd run:
```
git fetch https://github.com/raywenderlich/git-book-fizzbuzz.git
```

However, `fetch` behaves differently on URLs than on named remotes. As you saw previously, running `git fetch upstream` created the remote-tracking branch `upstream/main`. But if there isn't a named remote, there's no namespace to create remote-tracking branches in.
So you'll have to give the command the branch name to create. But when you specify a branch name as an argument, that argument is actually for the remote branch it should fetch:
```
git fetch {remote_url} {remote_branch_name}
```

So you have to give it the local branch to fetch it into as well:
```
git fetch {remote_url} {remote_branch_name:local_branch_name}
```

So what happens if you leave off the `:local_branch_name` part? The best way to find out is to try it out. Run the following fetch command:
```
git fetch https://github.com/jawwad/git-book-fizzbuzz.git allow-custom-range
```

You'll see the following output:
```
From https://github.com/jawwad/git-book-fizzbuzz
 * branch            allow-custom-range -> FETCH_HEAD
```

So what's this `FETCH_HEAD` thing? It's actually a reference that contains the last commit hash that was fetched. Run the following to see what it contains:
```
cat .git/FETCH_HEAD
```

You'll see the following:
```
c7580ff4a6231bbcfd21b46ddbb204ef472f590b        branch 'allow-custom-range' of https://github.com/jawwad/git-book-fizzbuzz
```

Now, you'll create a new branch based on `FETCH_HEAD`. Since naming a branch `allow-custom-range-from-fetch-head` feels just a bit long, abbreviate `allow-custom-range` as `acr` to name it `acr-from-fetch-head`.
```
git branch acr-from-fetch-head FETCH_HEAD
```

Run **`git log --oneline --graph --all`** to verify that the branch was created. You'll see the `acr-from-fetch-head` branch name next to the `c7580ff` commit hash:
```
* 061a436 (HEAD -> development, origin/fix-divisors-bug, fix-divisors-bug) F...
* 98b4ef3 (upstream/main, origin/main, origin/HEAD, main) Update README to r...
| * c7580ff (acr-from-fetch-head) Add start and end parameters to the fizzbu...
|/
* 85ca623 Add parameters to allow using divisors other than 3 and 5
* 8034fbf Add option to use words other than Fizz and Buzz
...
```

Next, you'll run the same command again, but this time you'll also give it a local branch name to fetch the changes into.
Run the following command to fetch the `allow-custom-range` branch from jawwad's fork into a local branch with the same name:
```
git fetch https://github.com/jawwad/git-book-fizzbuzz.git allow-custom-range:allow-custom-range
```

You'll see the following, indicating that a new local branch was created:
```
From https://github.com/jawwad/git-book-fizzbuzz
 * [new branch]      allow-custom-range -> allow-custom-range
```

Run **`git log --oneline --graph --all`** to confirm:
```
* 061a436 (HEAD -> development, origin/fix-divisors-bug, fix-divisors-bug) F...
* 98b4ef3 (upstream/main, origin/main, origin/HEAD, main) Update README to r...
| * c7580ff (allow-custom-range, acr-from-fetch-head) Add start and end para...
|/
* 85ca623 Add parameters to allow using divisors other than 3 and 5
* 8034fbf Add option to use words other than Fizz and Buzz
...
```

You'll see that the `c7580ff` commit shows references to both `allow-custom-range` and `acr-from-fetch-head`.
Both of these were just the first way of fetching using a repository URL. Next, you'll use the second way by using a special pull request reference.

Fetching a pull request

Any branches that are part of a pull request are available on the upstream repository in a special reference that uses the format: **`pull/{ID}/head`**. So for this pull request, it would be **`pull/3/head`**.
If you skipped the optional section on updating your fork using an upstream remote, just run the following command to first add an upstream remote:
```
git remote add upstream https://github.com/raywenderlich/git-book-fizzbuzz.git
```

You can run **`git remote -v`** to verify it was added and should see it in the output.
Now, run the following to create a local **acr-from-pull** branch from **pull/3/head**:
```
git fetch upstream pull/3/head:acr-from-pull
```

Then run the following command to verify a local `acr-from-pull` branch was created:
```
git log --oneline acr-from-pull
```

You'll see `acr-from-pull` on the same commit hash as `allow-custom-range`, indicating that `pull/3/head` also pointed to the same commit:
```
c7580ff (allow-custom-range, acr-from-pull, acr-from-fetch-head)
```

Before you actually merge this change, you'll use the third way of fetching, which is to add a fork as an additional remote and fetch from it. This will also allow you to experience how remote-tracking branches are automatically created when you have a named remote.

Adding an additional remote

Run the following to add jawwad's fork as an additional remote named `jawwad`:
```
git remote add jawwad https://github.com/jawwad/git-book-fizzbuzz.git
```

Run **`git remote -v`** to confirm its addition:
```
jawwad  https://github.com/jawwad/git-book-fizzbuzz.git (fetch)
jawwad  https://github.com/jawwad/git-book-fizzbuzz.git (push)
origin  https://github.com/jawwadahmad/git-book-fizzbuzz.git (fetch)
origin  https://github.com/jawwadahmad/git-book-fizzbuzz.git (push)
upstream    https://github.com/raywenderlich/git-book-fizzbuzz.git (fetch)
upstream    https://github.com/raywenderlich/git-book-fizzbuzz.git (push)
```

Now, run **`git fetch jawwad`**, and you'll see that the fetch command also created the remote-tracking branches — since there's now a `jawwad` namespace to create them in.
```
From https://github.com/jawwad/git-book-fizzbuzz
 * [new branch]      add-type-hints     -> jawwad/add-type-hints
 * [new branch]      allow-custom-range -> jawwad/allow-custom-range
 * [new branch]      fix-divisors-bug   -> jawwad/fix-divisors-bug
 * [new branch]      main               -> jawwad/main
```

This fetches all branches from that fork. You can verify this by comparing them to the branches on the following page:
https://github.com/jawwad/git-book-fizzbuzz/branches/all
If you run **`git log --oneline acr-from-pull`** again, you'll also see the additional reference to `jawwad/allow-custom-range` on that commit:
```
c7580ff (jawwad/allow-custom-range, allow-custom-range, acr-from-pull, acr-from-fetch-head)
```

This was just to demonstrate pulling from an additional named remote. Since you don't really need the additional remote, you can remove it with the following command:
```
git remote rm jawwad
```

The `git remote rm {remotename}` command deletes the remote-tracking branches as well.
You can run **`git remote -v`** to verify the remote was removed. And to verify that the remote-tracking branch was removed, you can run **`git log --oneline acr-from-pull`**. You'll no longer see a reference to the `jawwad/allow-custom-range` branch on the `c7580ff` commit.
You've seen three different ways to fetch updates from other forks. Now, you're finally ready to merge them!

Merging the pull request

Run the following to merge the **allow-custom-range** branch:

```
git merge allow-custom-range --no-edit
```

Now, delete the other two branches:

```
git branch -d acr-from-pull acr-from-fetch-head
```

It's a good idea to keep the **allow-custom-range** branch, even though you've merged it — just in case you need to re-create your development branch from the different branches you merged into it.
Finally, push your **development** branch up to your fork:

```
git push -u origin head
```

Congratulations! You learned how to fork a repo and keep it up to date with the original upstream repository. Plus, you learned various ways to fetch changes from forks and pull requests.

Key points

The Forking Workflow is used to contribute to repositories that you don't have push access to, like open-source repositories.
Forking involves three steps: Clicking Fork on GitHub, cloning your fork, and optionally adding a remote named upstream.
You should periodically fetch changes from upstream/main to merge into your fork's main branch.
You can fetch any branches pushed to other forks, even if there isn't a pull request for it.
To fetch all changes from a named remote, use `git fetch {remotename}`.
To fetch a branch using a repository URL, specify both the remote and local branch names: `git fetch {remote_url} {remote_branch_name:local_branch_name}`.

12. Conclusion 10. Gitflow Workflow

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.