Pushing Local Commits to Github

Introduction

For this project, you'll need to fork an existing repository, fix a bug in a script, push your commit to GitHub, and create a pull request with your commit.

What you'll do

- Fork another repository
- Commit changes to your own fork and create pull requests to the upstream repository
- Gain familiarity with code reviews, and ensure that your fix runs fine on your system before creating the pull request
- Describe your pull request

Forking and detect function behavior

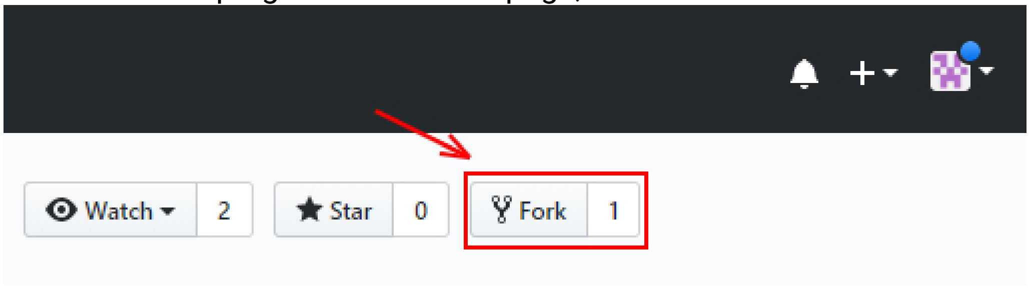
For this exercise, you need to fork an existing repository: `google/it-cert-automation-practice`.

- Open Github. If you don't already have a Github account, create one by entering a username, email, and password. If you already have a Github account proceed to the next step.
- Log in to your account from the Github login page.

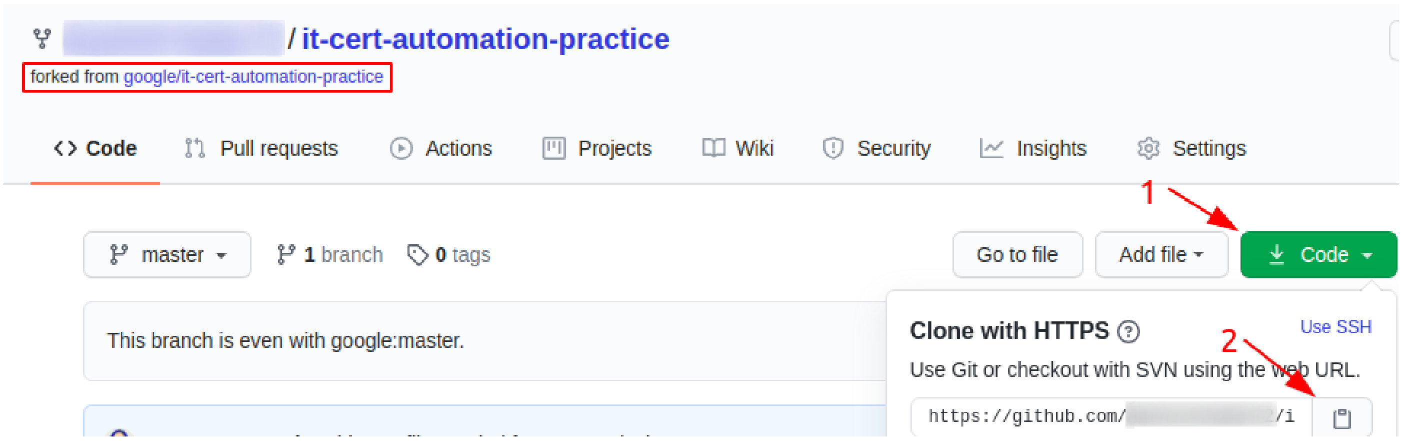
A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Forking a repository is a simple two-step process. We've created a repository for you to practice with!

- On GitHub, navigate to the google/it-cert-automation-practice repository.
- In the top-right corner of the page, click Fork.



That's it! Now, you have a fork of the original `google/it-cert-automation-practice` repository.



This branch is even with google:master.

Clone with HTTPS

Use Git or checkout with SVN using the web URL.

`https://github.com/[redacted]/i`

First, clone the repository using the following command:

```
git clone https://github.com/[git-username]/it-cert-automation-practice.git
```

Note: If you enabled two-factor authentication in your Github account you won't be able to push via HTTPS using your account's password. Instead you need to generate a personal access token. This can be done in the application settings of your Github account. Using this token as your password should allow you to push to your remote repository via HTTPS. Use your username as usual. Note down this personal access token as we would need it further in the lab. For more help to generate a personal access token, click here.

Output:

```
Cloning into 'it-cert-automation-practice'...
remote: Enumerating objects: 55, done.
remote: Total 55 (delta 0), reused 0 (delta 0), pack-reused 55
Receiving objects: 100% (55/55), 15.11 KiB | 619.00 KiB/s, done.
Resolving deltas: 100% (20/20), done.
```

Go to the `it-cert-automation-practice` directory using the following command:

```
cd ~/it-cert-automation-practice
```

First, verify that you have already setup a remote for the upstream repository, and an `origin`.Type the following command and press **Enter**. You'll see the current configured remote repository for your fork.

```
git remote -v
```

Output:

```
origin https://github.com/XXXXXXXX/it-cert-automation-practice.git (fetch)
origin https://github.com/XXXXXXXX/it-cert-automation-practice.git (push)
```

In terms of source control, you're "**downstream**" when you copy (clone, checkout, etc) from a repository. Information is flowed "downstream" to you.

When you make changes, you usually want to send them back "**upstream**" so they make it into that repository so that everyone pulling from the same source is working with all the same changes. This is mostly a social issue of how everyone can coordinate their work rather than a technical requirement of source control. You want to get your changes into the main project so you're not tracking divergent lines of development.

Setting the upstream for a fork you have created using the following command:

```
git remote add upstream https://github.com/[git-username]/it-cert-automation-practice.git
```

To verify the new upstream repository you've specified for your fork, type `git remote -v` again. You should see the URL for your fork as origin, and the URL for the original repository as upstream.

```
git remote -v
```

Output:

```
origin https://github.com/XXXXXXXX/it-cert-automation-practice.git (fetch)
origin https://github.com/XXXXXXXX/it-cert-automation-practice.git (push)
upstream      https://github.com/XXXXXXXX/it-cert-automation-practice.git (fetch)
upstream      https://github.com/XXXXXXXX/it-cert-automation-practice.git (push)
```

Configure Git

Git uses a username to associate commits with an identity. It does this by using the **git config** command. Set the Git username with the following command:

```
git config --global user.name "Name"
```

Replace **Name** with your name. Any future commits you push to GitHub from the command line will now be represented by this name. You can even use **git config** to change the name associated with your Git commits. This will only affect future commits and won't change the name used for past commits.

Let's set your email address to associate them with your Git commits.

```
git config --global user.email "user@example.com"
```

Replace **user@example.com** with your email-id. Any future commits you now push to GitHub will be associated with this email address. You can also use **git config** to change the user email associated with your Git commits.

Fix the script

In this section we are going to fix an issue that has been filed. Navigate to the issue, and have a look at it.
Branches allow you to add new features or test out ideas without putting your main project at risk. In order to add new changes into the repo directory `it-cert-automation-practice/Course3/Lab4/`, create a new **branch** named `improve-username-behavior` in your forked repository using the following command:

```
git branch improve-username-behavior
```

Go to the `improve-username-behavior` branch from the master branch.

```
git checkout improve-username-behavior
```

Now, navigate to the working directory `Lab4/`.

```
cd ~/it-cert-automation-practice/Course3/Lab4
```

List the files in directory `Lab4`.

```
ls
```

Output:

```
validations.py
```

Now, open the **validations.py** script.

```
cat validations.py
```

Output:

```
#!/usr/bin/env python3

import re

def validate_user(username, minlen):
    """Checks if the received username matches the required conditions."""
    if type(username) != str:
        raise TypeError("username must be a string")
    if minlen < 1:
        raise ValueError("minlen must be at least 1")

    # Usernames can't be shorter than minlen
    if len(username) < minlen:
        return False
    # Usernames can only use letters, numbers, dots and underscores
    if not re.match('^[a-z0-9._]*$', username):
        return False
    # Usernames can't begin with a number
    if username[0].isnumeric():
        return False
    return True
```

This script should validate usernames if they start with an letter only.

Here, you can check the `validate_user` function's behavior by calling the function. To edit the validations.py Python script, open it in a nano editor using the following command:

```
nano validations.py
```

Now, add the following lines of code at the end of the script:

```
print(validate_user("blue.kale", 3)) # True
print(validate_user(".blue.kale", 3)) # Currently True, should be False
print(validate_user("red_quinoa", 4)) # True
print(validate_user("_red_quinoa", 4)) # Currently True, should be False
```

Once you've finished writing this script, save the file by pressing Ctrl-o, the Enter key, and Ctrl-x.

Now, run the validations.py on the python3 interpreter.

```
python3 validations.py
```

Output:

```
True
True
True
True
```

Here, as we see the output, it function returns true even if the username doesnot start with an letter. Here we need to change the check of the first character as only letters are allowed in the first character of the username.

Continue by opening validations.py in the nano editor using the following command:

```
nano validations.py
```

There are lots of ways to fix the code; ultimately, you'll want to add additional conditional checks to validate the first character doesn't start with either of the forbidden characters. You can choose whichever way you'd like to implement this.

Once you've finished writing this script, save the file by pressing Ctrl-o, the Enter key, and Ctrl-x.

Now, run the validations.py.

```
python3 validations.py
```

Output:

```
True
False
True
False
```

Now, you've fixed the function behavior!

Commit the changes

Once the issue is fixed and verified, create a new commit by adding the file to the staging area. You can check the status using the following command:

```
git status
```

The **git status** command shows the different states of the files in your working directory and staging area, like files that are modified but unstaged and files that are staged but not yet committed.

You can now see that the validations.py has been modified.

```
On branch improve-username-behavior
Changes not staged for commit:
  (use "git add ..." to update what will be committed)
  (use "git restore ..." to discard changes in working directory)
	modified:   validations.py

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

Now, let's add the file to the staging area using the following command:

```
git add validations.py
```

Use the **git add** command to add content from the working directory into the staging area for the next commit.

```
git status
```

Output:

```
On branch improve-username-behavior
Changes to be committed:
  (use "git restore --staged ..." to unstage)
	modified:   validations.py
```

Let's now commit the changes. A git commit is like saving your work.

Commit the changes using the following command:

```
git commit
```

This now opens up an editor that asks you to type a commit message. Every commit has an associated commit message, which is a log message from the user describing the changes.

Enter a commit message of your choice and append a line: "Closes: #1" at the beginning to indicate that you're closing the issue. Adding this keyword has an additional effect when using Github to manage your repos, which will automatically close the issue for you (for more information, please see the documentation here).

```
Closes: #1
Updated validations.py python script.
Fixed the behavior of validate_user function in validations.py.
```

Once you've entered the commit message, save it by pressing Ctrl-o and the Enter key. To exit, click Ctrl-x.

Output:

```
[improve-username-behavior d947d11] Closes: #1 Updated validations.py python script.
Fixed the behavior of validate_user function in validations.py.
 1 file changed, 5 insertions(+), 2 deletions(-)
```

Push changes

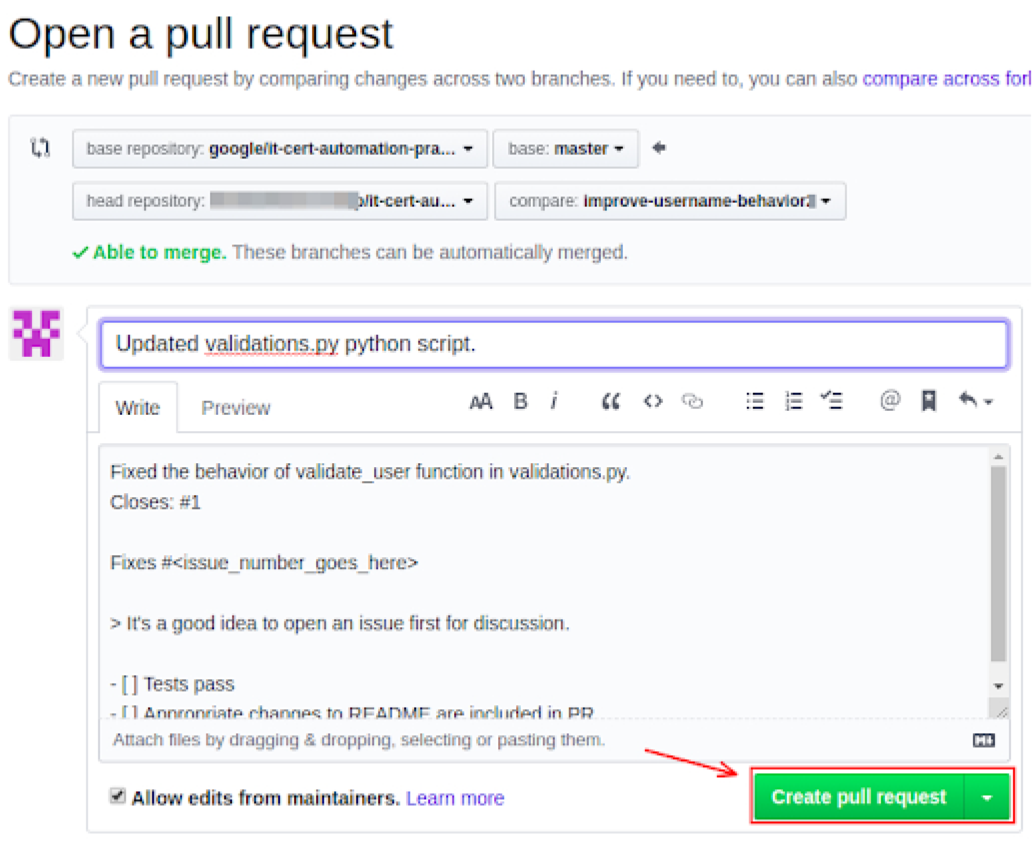
You forked a repository and made changes to the fork. Now you can ask that the upstream repository accept your changes by creating a pull request. Now, let's push the changes.

```
git push origin improve-username-behavior
```

Note: If you have enabled two-factor authentication in your Github account, use the personal access token generated earlier during the lab as the password. If you have not noted down the personal access token earlier, go to your **Github homepage** > **Settings** > **Developer settings** > **Personal access tokens**. Now click on the token you generated earlier and click 'Regenerate token'. Use your username as usual.
Output:

```
Username for 'https://github.com': XXXXXXXX
Password for 'https://XXXXXXXX@github.com':
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 2 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (5/5), 552 bytes | 552.00 KiB/s, done.
Total 5 (delta 2), reused 3 (delta 1), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: Create a pull request for 'improve-username-behavior' on GitHub by visiting:
remote:                             https://github.com/XXXXXXXX/it-cert-automation-
practice/pull/new/improve-username-behavior
remote:
To https://github.com/XXXXXXXX/it-cert-automation-practice.git
 * [new branch]      improve-username-behavior -> improve-username-behavior
```

Then, from GitHub, create a pull request from your forked repository `[git-username]/it-cert-automation-practice` that includes a description of your change. Your branch `improve-username-behavior` is now able to merge into the master branch. It should look like the image below:


Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forl

base repository: google/it-cert-automation-pra... base: master

head repository: [redacted]p/it-cert-au... compare: improve-username-behavior

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

Updated validations.py python script.

```
Fixed the behavior of validate_user function in validations.py.
Closes: #1

Fixes #<issue_number_goes_here>

> It's a good idea to open an issue first for discussion.

- [ ] Tests pass
- [ ] Appropriate changes to README are included in PR
```

Attach files by dragging & dropping, selecting or pasting them.

✓ Allow edits from maintainers. Learn more

After initializing a pull request, you'll see a review page that shows a high-level overview of the changes between your branch (the compare branch) and the repository's base branch. You can add a summary of the proposed changes, review the changes made by commits, add labels, milestones, and assignees, and **@mention** individual contributors or teams.

Once you've created a pull request, you can push commits from your topic branch to add them to your existing pull request. These commits will appear in chronological order within your pull request and the changes will be visible in the "Files changed" tab.

Other contributors can review your proposed changes, add review comments, contribute to the pull request discussion, and even add commits to the pull request.

You can see information about the branch's current deployment status and past deployment activity on the "Conversation" tab.

Note: PR won't be merged on the master branch so that other users can also make a similar change to fix the issue.