HW03: Fully Functional Gitty Psychedelic Robotic Turtles

- Homework HW03 should be completed **alone**.
- NOTE: You may see the term `main` and `master` when describing branches. While I am trying to remove all uses of master in all assignments, it is still frequently used in industry and git and pops its ugly head in on occasion. Anytime we're talking about branches, they are the same branch, and should appear in your repository as `main`!

Learning Objectives

- Continue practicing creating and using functions.
- More practice on using the turtle library.
- Learn about how computers represent colors.
- Learn about source control and Git.

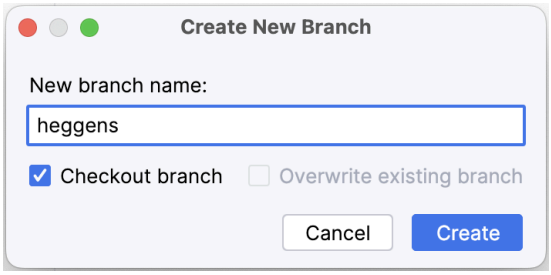
How to Start

- To begin, make a copy of this document by going to File >> Make a Copy...
- Change the file name of this document to ***username* - HW03: Fully Functional Gitty Psychedelic Robotic Turtles** (for example, **heggens - HW03: Fully Functional Gitty Psychedelic Robotic Turtles**). To do this, click the label in the top left corner of your browser.

NOTE: The next instruction is DIFFERENT than usual!

GitHub Repo Link (DO NOT CHANGE):	https://github.com/Berea-College-CSC-226/hw03-main

- Open PyCharm.
- Click **Clone Repository** to clone the HW03 repository using the link above.
- Create a new git branch, named with your username (e.g., **hw03_heggens** for me).



WARNING: Commit your changes as usual, but <u>do not PUSH</u> any changes yet until you've done a lot of reading below!

Go to **hw03_questions_do_not_edit.md. Follow the instructions at the beginning,** then come back to this document to complete the rest of the assignment.

Task 1: Turtle Houses, Animals, People

Refer back to **Teamwork T03** for some examples of code that uses functions from the turtle library:

- **t03_multicolor_square.py**
- **t03_spiral_input.py**
- **t03_functions_house.py**

Make sure you can read code with functions and follow the flow of execution. You'll find many of the functions used in the examples are helpful for your task ahead.

Image Colors

Knowing a bit about how colors are represented in computers will be important for your task ahead as well.

Images displayed on a screen use light for the display. Any three colors (or frequencies) of light that produce white light when combined with full intensity are called primary colors. The most commonly used set of primary colors of light is the set Red (R), Green (G), and Blue (B). Using a term borrowed from neuroscience, each color is typically called a color channel.

The RGB Color Wheel is an online tool for exploring color channels. Try inputting RGB color channel values between 0 and 255, as well as moving the cursor around the color wheel and seeing what values for RGB it produces.

To specify RGB colors this way in your code, you will have to add the following command: `screen.colormode(255)`. (By default, the turtle library will expect values between 0 and 1. This changes it to 0 and 255.)

> Go to *your* **hw03_username.md,** and answer **SECTION 1,** then come back to this document to complete the rest of the assignment.

Methods in the Turtle Library

In addition to colors, you'll find that there are things you want to do with turtles that you haven't yet used in other programs. Often, in computer science, we must refer back to the documentation to find what we are looking for. The **documentation for the turtle library** includes ALL methods the turtle library currently supports.

> Go to *your* **hw03_username.md,** and answer **SECTION 2**, then come back to this document to complete the rest of the assignment.

Task 2: Enough Already, Release the Turtles!

NOTE: WAIT TO START WRITING CODE UNTIL YOU HAVE READ THROUGH TASKS 2 AND 3.

In this assignment, you will draw something complex, like a house, animal, or person. Some ground rules:

1. Make sure to include the header comment block with your name and username in your Python file. Also in the header of your Python document, please paste the web address for your Google Doc.
2. Make sure the thing you draw:
    - Has at least one complex thing which looks like something; a building, animal, person, etc.
    - Is set against a background which is not white. You can use an image or a color as your background.

- Has some embellishments or interesting details, such as windows, text, trees, flowers in front of a house, intricate windows, smoke out of the chimney, or something; these are not all required, they are just suggestions. Be creative!
- Use an unnamed color using either RGB or Hexadecimal.
- Use creativity (such as the use of color, an intricate shape, a cool design…)

3. Technical requirements we will be looking for:
   - Use functions for encapsulating "mental chunks". *A key learning goal for this homework is to practice creating functions, so don't skip this one!*
   - Include a `main()` function definition and call at the end of your code.
   - The highest level of your program (i.e., no indenting) should **only** contain the following:
     i. The standard CSC 226 header block included in all files, and a link to this document
     ii. Any import statements
     iii. Function definitions
     iv. A `main()` function
     v. A call to the `main()` function
   - Make effective use of functions and use docstrings to help clearly explain what each function is designed to do.
     i. All of your own function definitions should come before the `def main():` function definition AND the call to the `main()` function. In other words, the last lines of your code should be:

```
def main():
    # your code inside of main
main()
```

     ii. All of your own function definitions should be at the highest level of your program (i.e., no indenting). Though it is possible to do, functions should really not be DEFINED inside of other functions (they should all be CALLED inside another function, like `main`, though).
   - Lastly, be sure to include comments for the sections in your code which draw the different objects.

Hint: The **hw03_stubs_do_not_edit.py** sample file contains many of the technical requirements described above, and shows the general structure of your code. Feel encouraged to copy the file and paste a new version for yourself. **DO NOT edit hw03_stubs_do_not_edit.py directly!**

```
################################################################################
# Header Block
################################################################################

import turtle

def function_1():
    """
    Docstring for function_1
    """
    pass
    # ...


def function_2():
    """
    Docstring for function_1
    """
    pass
    # ...


def main():
    """
    Docstring for main
    """
    pass
    # ...

main()
```

READING: Refresher on Git

As part of this course, we have been introducing you to a tool often used in software engineering: **Git**. Git is a **version control system** for software. Git also has another extremely powerful use for software engineers; preventing two (or more) programmers from writing code in the same file and accidentally deleting each other's work.

Imagine you're building a program such as the software you are looking at right now: Google Drive. Certainly you realize that more than one developer created all this. To ensure multiple developers don't clobber each other's hard work, they probably use git to manage versions.

So, each developer **clones** all of the code, makes some changes to that code, then requests the code be incorporated in the final product via a **pull request**. Git monitors all of the files, and when it notices two developers wrote code in the same spot, it flags those two sets of code as having a **merge conflict**. Then, the developers can discuss whose code should be kept, or modify the code so that both can be kept. By developing *good communication* within an organization about who is working on what features, coupled with Git as a failsafe, developers are able to produce very large programs more quickly, with less duplication of labor, and with fewer errors.

As you write code, you'll want to mark important points where you accomplish subtasks. **Committing** a file is like tagging changes, or revisions, in a document. A **commit** is typically an important point in your development process where you want to save progress. So, for example, say you and your pair partner disagree on the best approach to solving a particular algorithm. You can commit your code and start fresh on the new idea. If your new idea doesn't pan out and you need to go back, then you **revert** your changes back to the previous commit. No work was lost!

Commits are only useful if they have a **commit message** associated with them. Commit messages are tags indicating what changes were made since the last commit. *In previous assignments*, after you're done coding, you send the code back up to Github using a **push**. A push saves your changes on Github's servers, so you (and any collaborators) can then use the code for their continued development of the software you're creating.

> Go to *your* **hw03_questions.md** and answer **SECTION 3**, section, then come back to this document to complete the rest of the assignment

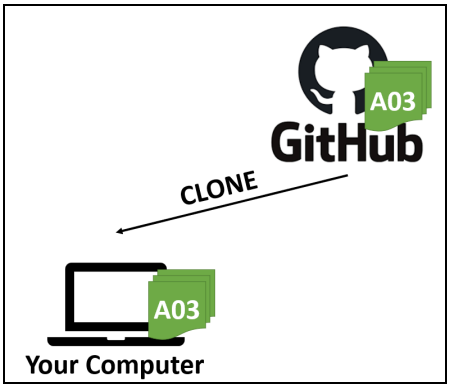
---

The Basics of Git

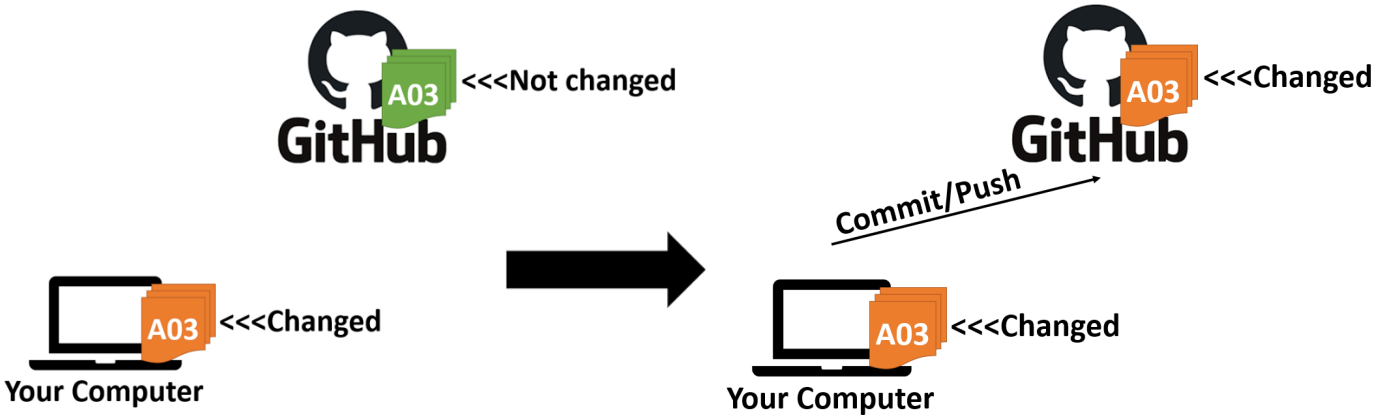
Now let's learn a little more git. Instead of cloning using the usual GitHub Classroom, we're going to clone a repository directly so we can *all* work on a single set of code. When you work on an open source project, or possibly a codebase in industry, you will likely use a platform much like Github. Remember that with many people working on one set of code, there has to be a mechanism to keep one person's work from clobbering another's. That's what **branches** are for! Before we write any code or make any changes, let's see a picture of the situation we're dealing with.

1. This is **Github**! Github is like a cloud storage container for codebases, and it keeps track of different versions of the code over time, as well as much more! Notice how Github is holding some files. That's Github's copy of our program; Github always keeps its own copy.

2. Whenever we **clone** a repository, what we're doing is asking Github to let us make our own personal copy on our computer. **Notice that Github's copy and our copy are SEPARATE! So far, changing one does not change the other!** Let's take a look:



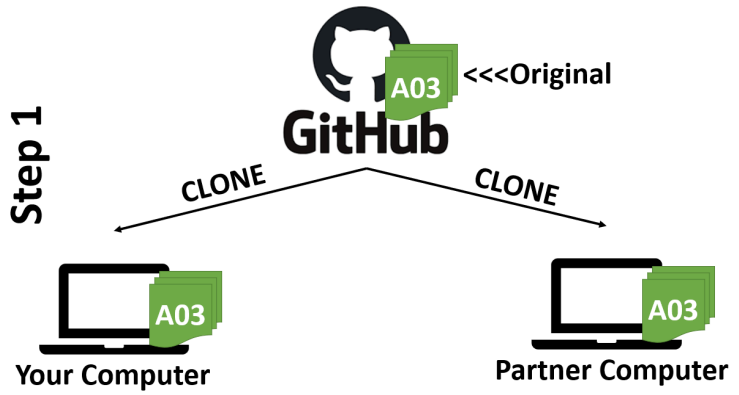
3. See how both sets of code are green? We'll use color to denote the version of the code (please let me know if the color choice causes any complications). Say for instance that you make some changes to that code. Then the color of the code will change in our example. If you make some changes to your code on **your computer**, the changes will not reflect in Github until you **commit and push**, like this:

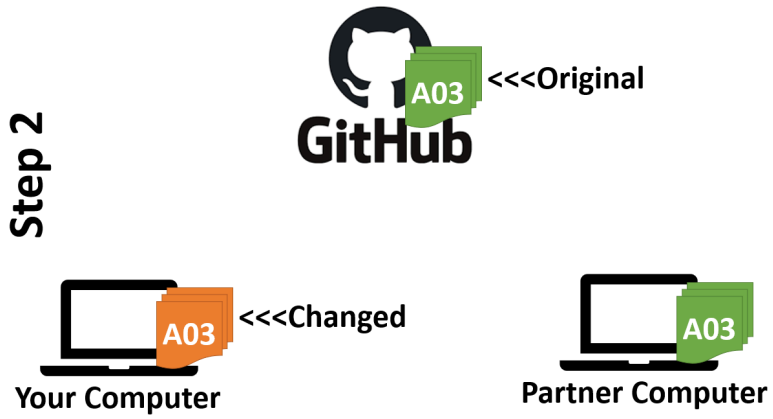


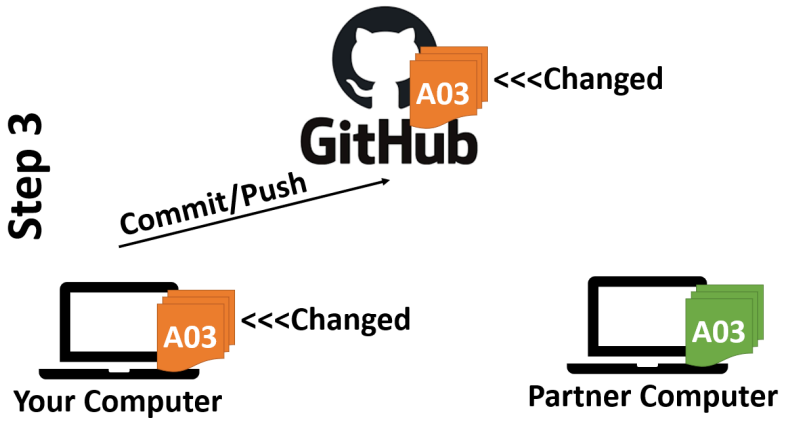
We've seen this process before! This is what happens when you submit your work for each assignment.

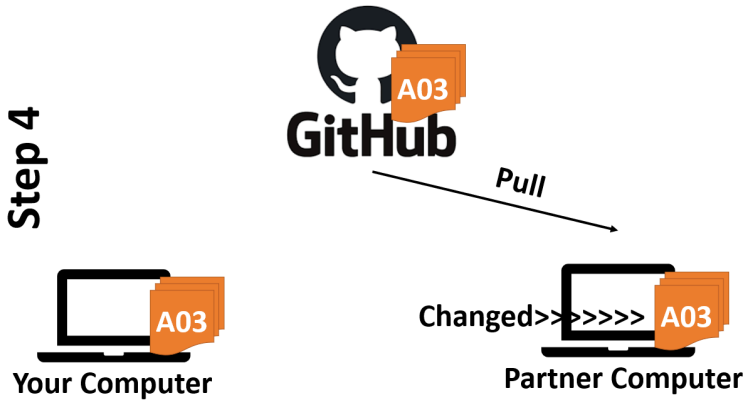
Using Git with a Partner

4. You may have seen another case when you are working with a teammate. You might have had a partner who did some work on the code, and then wondered how to get those changes onto your computer. That process is just like the one above, but the second partner needs to **pull** to see those changes, as illustrated below in the next four images. Pay attention to the terminology used!:





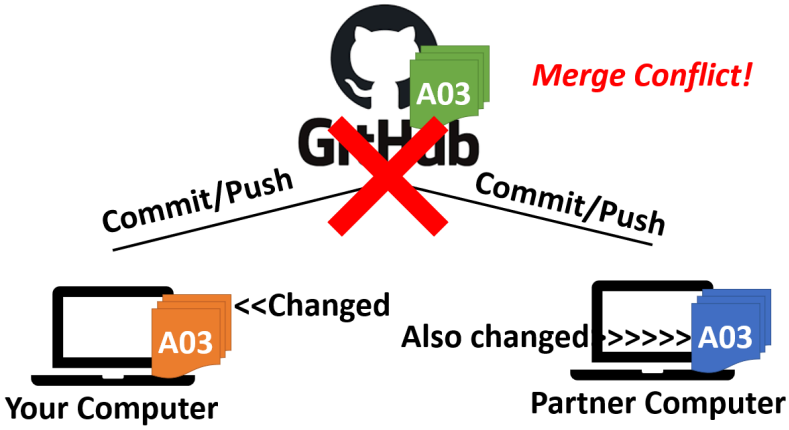




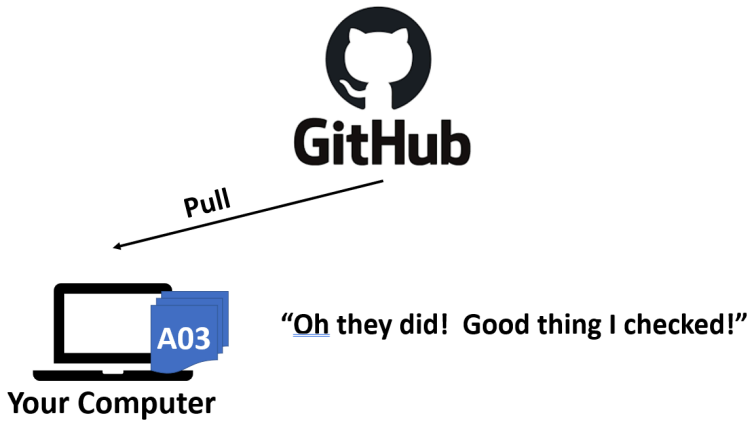
And that's it! That's how you get code from one partner's computer to the other!

Merge Conflicts and How to Avoid Them

5. You may see a potential problem here. **What if you and your partner both edit the code and then try to push?!**



Sometimes, you don't get a merge conflict from this process. For instance, if you and your partner changed completely different files, then there *likely* won't be any problem. **But, if you both change the same lines of the same file, that's when conflicts happen!** The way to get around this is to **make sure you have pulled any changes your partner made before you make your own changes**. Here is an example:



---

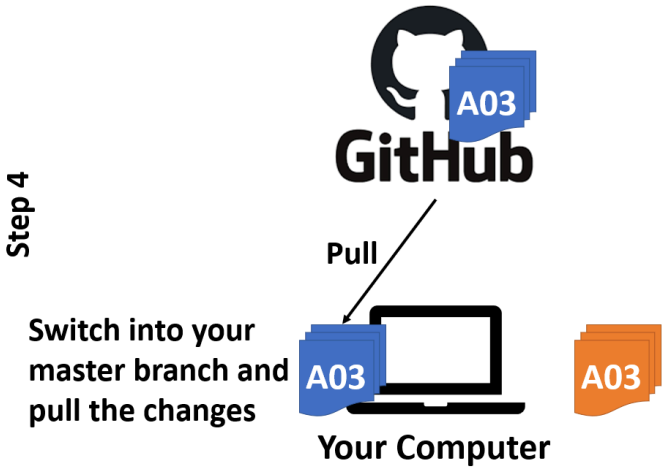
6. That's the general idea you want to keep in mind moving forward. **The whole process above is perfectly fine when everyone is working on the same branch of the code (main or another branch that you created). However, on bigger projects YOU WILL NOT BE EDITING THE MAIN BRANCH DIRECTLY.** This is where branches become important:

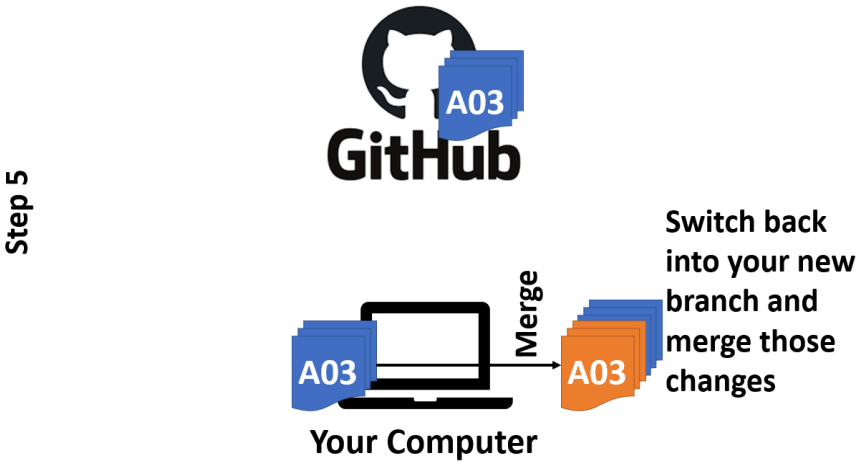
Git Branching

7. When working on a big project (or one of your own that you want to keep organized), you'll want to have a **main** copy of your code (the "clean copy" you'll want to submit when you're finished), and one or more "scratch" copies that you can use to try new things out on, and then delete when you're done if they don't work. Or if they DO work, you can (very intentionally) incorporate them into your **main** copy! The process follows the same logic we saw above, but just with one extra step in the middle. Step 1 is the same as always: clone the repo; step 2 now includes a term we call **checkout a branch**. You have been doing this all semester!

8. This new branch you **checked out** is where you'll do your practice work that you *might* want to incorporate into the **main** branch, if it works. You can try out new things here to your heart's content! And if something doesn't work out in your branch, don't worry. You can always delete it and create a new one if you need to.
9. Once you've made some changes and you're satisfied with them, you'll want to commit and push them to Github. **BUT REMEMBER, if anyone else is working on the same branch,** you want to make sure your code is as up-to-date as it can be. In other words, always PULL before you PUSH!.
10. ALSO REMEMBER, you currently have two separate copies of the code on your computer: your main branch and your new branch**.** We need to make sure they're **both** up to date. To do this, you'll use the following steps:
    a. **Checkout** your main branch
    b. **Pull** from Github (to get the most recent changes from others on main)
    c. **Checkout** your branch
    d. **Merge the** main branch into your new branch (to pull those new changes into your code and verify everything works still)

11. These steps, starting at step 3 (steps 1 and 2 are the same as above) are illustrated below:





12. At this point, if you look at your repository in Github you will notice that your branch is there. Remember, before you pushed your changes from your branch to Github, Github didn't know it existed; it was only on your computer. However, your changes are still in your branch, not the main branch. For that, you'll now need to submit a **pull request**. A **pull request** is the same as submitting your homework and formally asking the instructor to grade it. A **pull request** is a formal request to have your code incorporated into the main branch and become officially part of the software you are helping to build. Many software engineers pride themselves on the number of **pull requests** they have had accepted into major projects.

Task 3: Making Your Own Branch

*NOTE: In all of the instructions below, you should get a confirmation message in the bottom right of PyCharm. If you get any failure messages, please seek help from the teaching assistants or the instructors **before** moving on.*

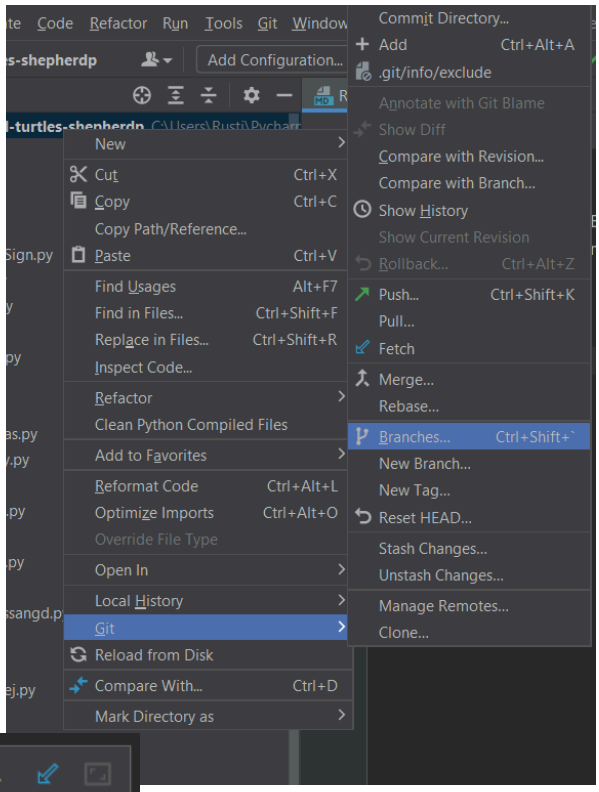
1. Now let's see how to put the ideas above into practice. You might remember that we didn't start this assignment using the usual Github Classroom process. That is because we are all going to work in a single repository for this assignment. Copy the following link, and use it to "Clone Repository" in PyCharm (the same process you've been doing, just with this link instead): **shared Github repository**.

2. Now, imagine you're working on a feature (e.g., a fluffy cloud) for your amazing project above, and you get the idea that your cloud could REALLY use a house to float over. You really want to add a house to your project, and it must be done ASAP; the due date is right around the corner, after all! You don't want to lose the feature you've been working on, as it's almost done, but you need to make this new house a priority.

   Branching allows us to save versions of our code, with each branch representing a different feature, issue, bug, or some other change that needs to be made. In PyCharm, right click on the repository folder, then click Git >> Branches, and create a new branch. Name the branch you're about to create something unique and descriptive (e.g., **hw03_heggens_house**).

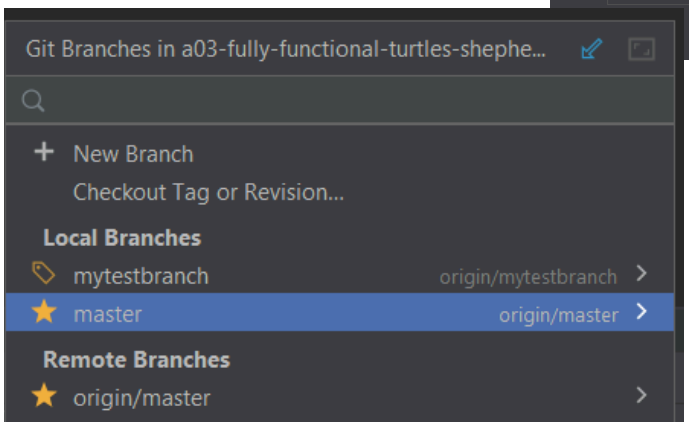
3. Next, create a new file by clicking on File >> New >> Python File, and name it **hw03_username.py** (with your username). You will also need to copy and paste any image files you use into your project. You should see the new file on the left with all the other files.

4. Go ahead and make some changes to your code, and then it's time to **commit and push**. You are welcome to go ahead and do so once, as long as you are on your branch. If you do that now, you can answer the questions in SECTION 4 of **hw03_questions.md** now (otherwise, answer them when asked later in the assignment).



Pull main into your local

We always want to make sure you are in sync with the main before pushing your code back into it. In other words, **always pull before you push**.



1. In PyCharm, switch to the main branch by going to Git >> Branches… and selecting the local branch called **main**, then click "Checkout". While we're here, let's clear up something that might be confusing. In particular, there are two branches listed here called **main**, which seems a little strange at first! But remember, you have three copies of the code floating around right now that are important for you:

    - The copy on Github (that's **origin/main** under **Remote Branches**)
    - The main branch on your computer (that's **main** under **Local Branches**)
    - The new branch you made (that's **mytestbranch** under **Local Branches** in this example above)

    And if you have pushed your code as suggested above, you'll also see your branch under the Remote branches (e.g., for my example, **origin/mytestbranch**)!

2. Now click Git >> Pull to update your local main branch with the remote's main branch.

The important thing to note here is a) where are you pulling to, and b) where are you pulling from. In this case, I asked you to pull from **remote/main** to **local/main**. This is updating your local project with any changes made by other students, so we can incorporate them into our code before we push.

Merge main into your branch

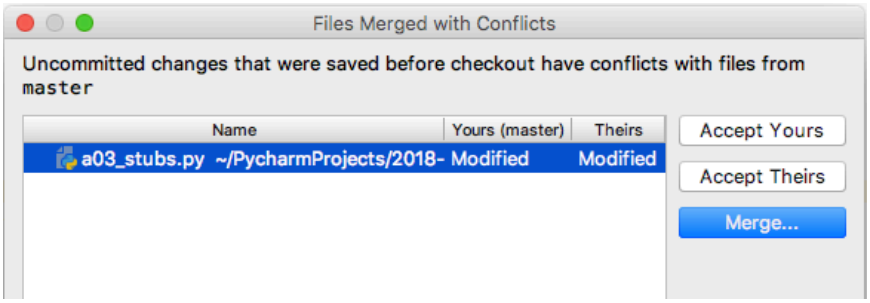
At this point, you've pulled all changes from the main branch in Github into your main branch on your local, but the changes are still in main, not your branch. Next, merge those changes into your branch:

3. Switch back to your branch, using the same steps as above.
4. Click Git >> Branches >> main >> Merge Selected Into Current (this may say Merge main Into yourBranchName).

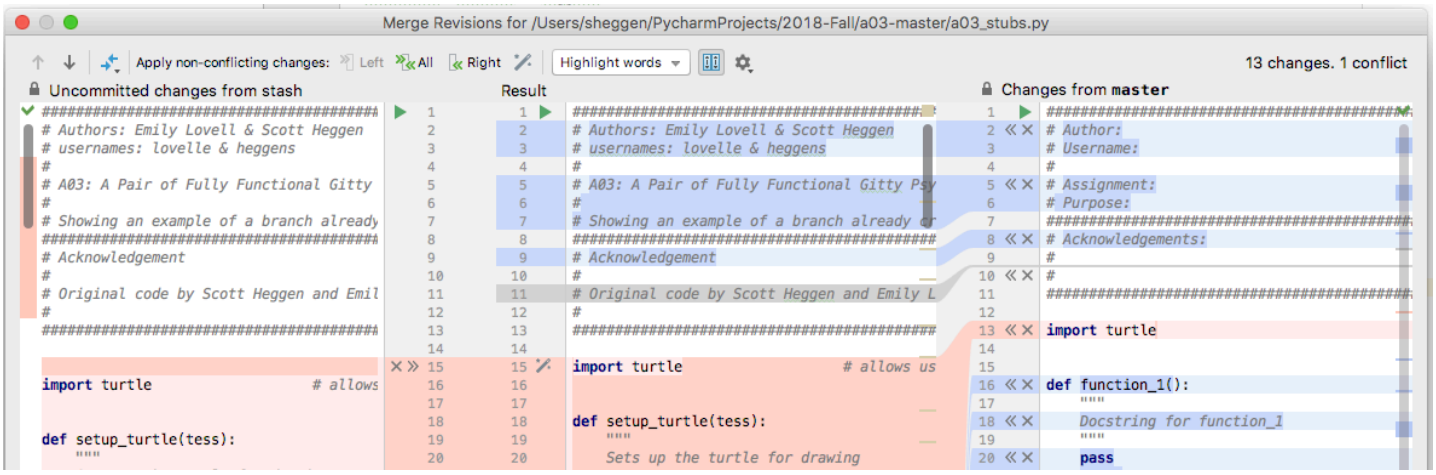
When you try to perform the merge, multiple things may happen, and not always the same for everyone.

Potential Response 1: Some of you may get an error message in the bottom right corner of PyCharm saying it can't merge. This is unlikely, but the cause is simple: you have uncommitted work which can't be overwritten by git. Commit your branch, then repeat the steps above.

Potential Response 2: After selecting Merge above, you may have been asked to pick which file to keep: **Yours** (your local copy) or **Theirs** (what's in the Github repo).



Double click the file, and you'll see what differences exist between the two files. Make an educated decision about which file you should keep based on what you see in the **Merge Revisions** interface:



Merging can be a messy process, as you may be attempting to merge multiple people's work into a single working program. Again, good communication across the team will mitigate these problems, as git can only do so much! The likely cause of this situation is you forgot to do a step above renaming files (so you and someone else are both editing the template file that you were supposed to copy).

Potential Response 3: Everything goes through just fine. Whoo!

At this point, you've merged main into your branch, and you are ready to "submit" it via a Pull Request:

17. Check to make sure you're on your own branch (i.e., the top of PyCharm should NOT say main), then **Commit** and **Push** your branch to GitHub.

    If you get a different error, or do not know how to move forward, seek help from the instructors or the teaching assistants in the computer science lab before moving on.

18. Now, let's see how things look in the **GitHub repository** by going there in the browser. Note that it is called **Berea-College-CSC-226/hw03-main. If it is called anything else, that is the incorrect repository!**

> Go to **hw03_questions_username.md** and answer **SECTION 4**, then come back to this document to complete the rest of the assignment

---

Issue a Pull Request

The next step in the process is to formally ask for your code to be included into main. This is a request; you don't get to incorporate your code into main; the repository owner (i.e., instructor) does! You are making a formal request for your code to be included. We call this a **Pull Request,** which is the same process you've been using to submit your teamwork and homework assignments all semester.

19. In Github (in the browser), you'll notice your branch is now there. Next to your branch is a button which says "Compare and Pull Request". Click it. This interface allows you to request that your branch be included in the main branch. Without merging branches, your code and everyone else's code are never integrated into a single product, which is ultimately the goal of using Git (one product, many developers). **VERY IMPORTANT! Make sure your pull request is going to & from the right branch:**

20. Leave a meaningful message that lets the repository owner (your instructors) know that you would like your code integrated. In a real software engineering project, we would review your code, ensure everything works correctly, and accept the pull request.

    NOTE: The instructor is responsible for approving your pull request (i.e., merging your code into main), NOT YOU!

Task 4: PR Feedback

Very rarely are pull requests perfect the first time they are submitted. The repository owner will often give you feedback, and expect you to make those changes. This is a normal part of the process.

To simulate this process, check your PR daily. The instructor will leave you a note about what changes he needs you to make in order to accept your PR. One of the greatest features about pull requests: when you commit and push changes to your branch, they automatically update the pull request! So, fix the change requested by the TAs, commit, push, and you are done!

The last step is syncing **main** to your local copy of the repository. Remember, the pull request merged your code into **main** in Github, but not in your local machine's **main** branch. This is done after your pull request has been approved and **main** has now changed.

Task 5: Ask a question

Go to your **hw03_questions_username.md** and answer **SECTION 5**, then come back to this document to complete the rest of the assignment

What the heck just happened?

The following graphic summarizes the entire git workflow:

Submission Instructions

1. After you have submitted your pull request, you have successfully submitted this assignment.
2. Be sure to check GitHub regularly to see whether the instructors have left you any tasks to fix your pull request.
3. Fix any requested changes (Task 4), and commit and push to automatically update your Pull Request.
4. Your assignment is complete when the instructor **approves your Pull Request!**