Day 05

Merging branches

In the natural development of a project, it is often the case that we will have multiple branches of development. These branches are often created to work on different features of a project. When the features are complete, we will want to merge the branches back together. This is done with a **merge**.

Merging

Merging is the process of taking the changes that have been made in one branch and applying them to another branch. This is done with the command `git merge`. The command is used like this:

For instance, if we have a branch called `dev` and we want to merge it into the `main` branch. The first thing would be to commit our changes into the `dev` branch. Then we would do this:

```
git checkout main
git merge dev
```

If there are no conflicts, then the changes will be applied to the main branch. If there are conflicts, then we will need to resolve them. This is done by opening the files that have conflicts and choosing which changes to keep.

Subclassing and Libraries

Subclasses

One of the big ideas of Object-Oriented Programming is **subclasses**. Subclasses make it possible to reuse ideas and build upon ideas in a way that makes programming way more efficient. Probably the best example of this is Minecraft. In Minecraft there are various ideas that are all similar to each other but have a lot of commonality too. There are bunches of different blocks, and there a bunch of different animals. To get all of these different things to be programmed you might think that there is a lot of copying of the programs that represent these different objects, but there isn't. The programmers use a kind of taxonomic system that allows programs to share the same code when they need to, and have specialized code when they need to This is subclassing.

Here is an example. Let's say I build a robot class, in a file called `robot.py` that looks something like this:

```
class Robot:

    def move(self, x, y):
        pass

    def beep(self):
        pass
```

It isn't much of a robot. It is really just a drivetrain and a something that goes beep, but it is mine and I made it. Don't judge. Now a build a better version of the same robot, but this time it has an arm that can rotate and grab. Instead of copying over all the code, that I used before I would do something like this:

```
from robot import Robot

class ArmBot(Robot):

    def __init__(self):
        super().__init__()

    def rotate_arm(self, degrees):
        pass

    def grab_claw(self):
        pass
    def release_claw(self):
        pass
```

The parentheses (Robot) at the top here tells this class that an ArmBot is a kind of Robot from file that I had previously made. It should therefore have access to all the things that a Robot can do, and I don't have explain how my ArmBot moves and beeps. I will just use the code from Robot. One line that looks a little wonky here is what is going on with `__init__`. The `super` term refers to the super class, in this case `Robot`. We need to do the things that make a `Robot` in order to make an `Armbot`. So we are saying make a Robot, and then we can finish making an Armbot.

This is a wonderful way to build on the work of others.

It is especially good when we use code that other people have written, because it allows us to build on others ideas.

Libraries

Though we can make our own objects for robotics it is more efficient and allows teams to build more consistently and learn from each other, if the classes that we used are standardized to some level. This leads people to build what are called **libraries**.

Libraries are collections of classes that have been made to be useful for various purposes. Libraries often have documentation that is called an **API**, or **Application Programming Interface**. We use libraries from several places. The first is **WPILib**. WPILib contains the most general grouping of libraries that allows an FRC robot to function. **Robotpy** is simply a python version of the most common FRC robot libraries so that we can utilize the WPILib framework.

Homework

For homework let's practice subclasses. In this repository make two files. One named `baserobot.py`. That file describes a class `BaseRobot` that needs to have the following methods.

- `__init__`: The constructor should take in two numbers and set them to be the self.x and self.y of the Robot.
- `move`: Move takes in two numbers in increases that x and y coordinates of the robot by those numbers
- `beep`: Beep prints the word **Beep** to the console.
- `__str__`: returns a string that says the position of the robot.

Then have another file, `armbot.py`, that describes a subclass of BaseRobot called `ArmBot`. An `ArmBot` can do all the things that BaseRobot can do, but with these differences:

- `__init__`: In the init method it should call the init of the BaseRobot class using super, but also set a value for its arm position, `self.arm_position` to be zero. It also sets a value for the claw indicating that the claw is closed. do this by setting `self.claw_open` to `False`.
- `move_arm`: This method takes in a number and increases the arm_position by that amount.
- `grab_claw`: This method sets the value of `self.claw_open` to be `False`.
- `release_claw`: This method sets the value of `self.claw_open` to be `True`.
- `__str___`: Reports the position of the robot and status of the arm and the claw.