Day 22 CS570

Unit Testing

Testing code

Testing code is an important part of the development process. As you make code you revise and revise it and revise it. As you make changes in the code you can make revision that have negative effects on the code in general in unexpected ways. Writing and running tests are ways to make sure that as you update and improve your code you continue to have functionality that you understand and is going to work in a way that you want.

Writing Tests for FRC Robots

Check out:

https://robotpy.readthedocs.io/en/stable/guide/testing.html

To start writing tests for the code use the repository I have here, and make sure that you have set up the proper libraries.

Starter Code You can find starter code for this project here: https://github.com/mbardoeChoate/DriveStraight2024

Let's go through the starter code. There are three files, `robot.py`, `subsystems/drivetrain.py`, `pyproject.toml`. You may need to make sure that set up the project to be a robotpy project. You can do that by running the following command:

```
python3 -m pip install robotpy
python3 -m robotpy init
python3 -m robotpy sync
```

In order to run the tests you will need to type the following into the terminal.

```
python3 -m robotpy test
```

This command makes a tests folder, and adds some basics tests that can make sure that the code runs in auto and teleop without error. This makes it possible to test your code without having to run the robot.

How to write tests

In writing a test there are a few things to know.

You will want to organize your tests the way that you have organized your robot. So you will put the tests for one file largely in one file.

ArcadeDrive Test At the top of the file you will need a few imports. Particularly, we will be using the `pytest` library. We will at times use the `unittest.mock` library.

Let's create a python file called `test_drivetrain.py` in a `tests` directory.

```
import pytest

from drivetrain import Drivetrain
```

When writing a test it is important that the filename have the word `test` in it. This is how the testing library knows that this is a file that contains tests.

You will also want the methods in the file to have a name that starts with `test`. This is how the testing library knows that this is a method that is a test.

```
def test_arcadeDrive():
    # Setup the test
    drivetrain=Drivetrain()
    drivetrain.drive=MagicMock()

    # Action
```

```
    drivetrain.arcadeDrive(.2, .3)
    # Assert
    drivetrain.drive.arcadeDrive.assert_called_once_with(.3, .2)
```

Let's go through the meaning of each important line. First we define a test, and each test has three parts, the setup, the action, and the assert.

The first line creates a `Drivetrain` object. This is the object that we are going to test. The second line creates a `MagicMock` object. This is an object that is going to pretend to be the `drive` object in the `Drivetrain` object. The third line is the action. This is the part of the test where we call the method that we are testing. The fourth line is the assert. This is the part of the test where we check to see if the method that we are testing is doing what we expect it to do.

Testing Reseting Encoders

Mocking As you might have seen from above the `MagicMock` object is a way to create a fake object that we can use in our tests. The `Drivetrain` class has many subobjects that might require mocking, and sometimes it is easiest just to mock the whole object once, and then use that object in the tests.

`MagicMock` is a class that allows us to create a fake object that we can use in our tests. It tracks what methods are called on it and what arguments are passed to those methods.

Then we can use that object in our tests. And while it doesn't do anything it can tell us if the methods are called.

Making a Fixture A fixture is going to be a function that gives a version of an object that can be tested.

```
@pytest.fixture
def drivetrain() -> Drivetrain:
    # Create a drivetrain, but it has mock
    # classes for its dependencies
    drive = Drivetrain()
    drive.left_motor = MagicMock()
    drive.right_motor = MagicMock()
    drive.leftEncoder = MagicMock()
    drive.rightEncoder = MagicMock()
    drive.drive = MagicMock()
    drive.gyro = MagicMock()
    return drive
```

Now we will use that function as a input to a function. And we will have a drivetrain that won't really drive motors.

```
def test_reset_encoders(drivetrain: Drivetrain):
    '''A test to make sure that when we reset the encoders all the encoders are reset.
    '''
    # Setup
    # Need access to the reset functions of each encoder
    left_reset = drivetrain.leftEncoder.reset
    right_reset = drivetrain.rightEncoder.reset

    # Action
    drivetrain.resetEncoders()

    # Assert
    left_reset.assert_called_once()
    right_reset.assert_called_once()
```

You can see that there are several assert methods that can be used with a mocked object. A full list of those options can be found here: https://docs.python.org/3/library/unittest.mock.html#unittest.mock.Mock.assert_called

In a perfect world this test is written first, and then the subsystem is written, and this tests the subsystem. Because we know before we begin that if we say `resetEncoders` we want both of the encoders to be reset.

Monkeypatching

MonkeyPatching allows us to override functions. `monkeypatch` is imported with the pytest library.

```
def test_averageDistanceMeter(drivetrain: Drivetrain, monkeypatch) -> None:
    # Setup
    def mock_getRightDistanceInch(self):
        return 3.0

    def mock_getLeftDistanceInch(self):
        return 2.0

    monkeypatch.setattr(Drivetrain, "getLeftDistanceInch", mock_getRightDistanceInch)
    monkeypatch.setattr(Drivetrain, "getRightDistanceInch", mock_getLeftDistanceInch)
    # Action

    dist = drivetrain.getAverageDistanceInch()

    # Assert
    assert dist == 2.5
```

Parameterizing

We might want to run the averageDistanceTest many times...

We will add a decorator to the top...

```
@pytest.mark.parametrize(('left_Distance', 'right_Distance', 'output'), (
        (2, 3, 2.5),
        (10, 20, 15),
        (-3, 3, 0)) )
def test_averageDistanceMeter(drivetrain: Drivetrain, monkeypatch, left_Distance,
                              right_Distance, output) -> None:
    # Setup
    def mock_getRightDistanceInch(self):
        return right_Distance

    def mock_getLeftDistanceInch(self):
        return left_Distance

    monkeypatch.setattr(Drivetrain, "getLeftDistanceInch", mock_getRightDistanceInch)
    monkeypatch.setattr(Drivetrain, "getRightDistanceInch", mock_getLeftDistanceInch)
    # Action

    dist = drivetrain.getAverageDistanceInch()

    # Assert
    assert dist == output
```

Your turn...

Write a test that uses tests to see if we tell the drivetrain to `resetGyro` that the gyro's `reset` method is called. You will want to use the fixture, and I don't think you will need to monkeypatch.