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.

```
pip install robotpy==2022.4.8
pip install robotpy'[commands2,sim]'
pip install robotpy-romi
```

Then you will want to install the software necessary to make tests.

```
python3 robot.py add-tests
```

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

```
python3 robot.py test
```

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:

```
import pytest

from drivetrain import Drivetrain

def test_arcadeDrive(drivetrain: Drivetrain) -> None:
    # Setup
    arcadeDrive = drivetrain.drive.arcadeDrive

    # Action
    drivetrain.drive.arcadeDrive(0.2, 0.3)

    # Assert

    arcadeDrive.assert_called_once()
```

Testing Reseting Encoders

Mocking Sometimes (often) in FRC we need to imagine that we have an object in a test that is not really there. When we are testing our code there isn't a gyro or a motor. How can we really test those things.

It is pretty tough to write some of those tests.

At the top of the file we add...

```
from unittest.mock import MagicMock
```

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()
```

Monkeypatching

MonkeyPatching allows us to override functions.

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

    def mock_getLeftDistanceMeter(self):
        return 2.0

    monkeypatch.setattr(Drivetrain, "getLeftDistanceMeter", mock_getRightDistanceMeter)
    monkeypatch.setattr(Drivetrain, "getRightDistanceMeter", mock_getLeftDistanceMeter)
    # Action

    dist = drivetrain.averageDistanceMeter()

    # 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: MonkeyPatch, left_Distanc
                              output) -> None:
    # Setup
    def mock_getRightDistanceMeter(self):
        return left_Distance

    def mock_getLeftDistanceMeter(self):
        return right_Distance

    monkeypatch.setattr(Drivetrain, "getLeftDistanceMeter", mock_getRightDistanceMeter)
    monkeypatch.setattr(Drivetrain, "getRightDistanceMeter", mock_getLeftDistanceMeter)
    # Action

    dist = drivetrain.averageDistanceMeter()

    # Assert
    assert dist == output
```

Checking for errors

If you want to make sure that an error that should be raised is raised. You can use this formulation.

```
with pytest.raises(ValueError):
    #Code that makes the error rise.
```