Day 23 CS570

Unit Testing Parameterization and Practice

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_Distance, right_Distance,
                              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.
```

Another Example

Take a look at this code for a DriveStraight autonomous:

```
from wpimath.controller import PIDController


class DriveStraight:
    direction_kp = 0
    direction_ki = 0
    direction_kd = 0
    distance_kp = -20
    distance_ki = 0
    distance_kd = 0

    def __init__(self, drivetrain, goal_in_meters):
        self.drivetrain = drivetrain
```

```
        self.distance_controller = PIDController(self.distance_kp, \
                                                 self.distance_ki, self.distance_kd)
        self.distance_controller.setSetpoint(goal_in_meters)
        self.direction_controller = PIDController(self.direction_kp,\
                                                  self.direction_ki, self.direction_kd)
        self.direction_controller.setSetpoint(0)

    def run(self):
        difference = (
                self.drivetrain.getLeftDistanceMeter()
                - self.drivetrain.getRightDistanceMeter()
        )
        distance = self.drivetrain.averageDistanceMeter()
        rotate = self.direction_controller.calculate(difference)
        forward = self.distance_controller.calculate(distance)
        if self.distance_controller.atSetpoint():
            self.drivetrain.arcadeDrive(0, 0)
        else:
            print(
                f"Fwd: {forward}, Rot: {rotate}  distance:{self.drivetrain.averageDistanceMeter()} "
                + f"difference:{difference}"
            )
            self.drivetrain.arcadeDrive(rotate, forward)
```

Before we write some tests for this class it might be good to refactor the code to break the `run` method into several different methods. This is because the `run` method doesn't have much cohesion. It both calculates the settings for the motors and sets the motors to those settings. I will refactor to make this two separate methods, then it will be easier to write my tests.

Refactored Code

```
from typing import Tuple
from wpimath.controller import PIDController

class DriveStraight:
    direction_kp = 1
    direction_ki = 0
    direction_kd = 0
    distance_kp = 20
    distance_ki = 0
    distance_kd = 0

    def __init__(self, drivetrain, goal_in_meters):
        self.drivetrain = drivetrain
        self.distance_controller = PIDController(self.distance_kp, \
                                                 self.distance_ki, self.distance_kd)
        self.distance_controller.setSetpoint(goal_in_meters)
        self.direction_controller = PIDController(self.direction_kp, \
                                                  self.direction_ki, self.direction_kd)
        self.direction_controller.setSetpoint(0)
        self.direction_controller.setTolerance(.03)

    def run(self):
        rotate, forward = self.calculate()
        if self.distance_controller.atSetpoint():
            self.drivetrain.arcadeDrive(0, 0)
        else:
            self.drivetrain.arcadeDrive(rotate, forward)
```

```
    def calculate(self) -> Tuple[float, float]:
        difference = (
                self.drivetrain.getLeftDistanceMeter()
                - self.drivetrain.getRightDistanceMeter()
        )
        distance = self.drivetrain.averageDistanceMeter()
        rotate = self.direction_controller.calculate(difference)
        forward = self.distance_controller.calculate(distance)
        print(
            f"Fwd: {forward}, Rot: {rotate}  " +
            "distance:{self.drivetrain.averageDistanceMeter()} "
            + f"difference:{difference}"
        )
        return rotate, forward
```

Here are the tests that I wrote for this class:

```
from unittest.mock import MagicMock

import pytest
from pytest import MonkeyPatch

from drivestraight import DriveStraight
from drivetrain import Drivetrain


def greater(a, b):
    return a > b


@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


@pytest.mark.parametrize(('left_Distance', 'right_Distance', \
                          'direction_output', 'distance_output'), (
        (1.5, 1.4, False, True),
        (2.2, 2.3, True, False),
        (2, 2, False, False)
), )
def test_calculate(drivetrain: Drivetrain, monkeypatch: MonkeyPatch,\
                   left_Distance, right_Distance, direction_output, \
                   distance_output) -> None:
    # Setup
    autoroutine = DriveStraight(drivetrain, 2)
```

```
    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
    rotate, forward = autoroutine.calculate()

    # Assert
    assert greater(rotate, 0) == direction_output
    assert greater(forward, 0) == distance_output


@pytest.mark.parametrize(('left_Distance', 'right_Distance', 'at_Setpoint'), (
        (1.95, 1.95, False),
        (2.2, 2.3, False),
        (2, 2, True),
        (2.01, 1.99, True)
), )
def test_run_setpoint(drivetrain: Drivetrain, monkeypatch: MonkeyPatch,\
                      left_Distance, right_Distance, at_Setpoint) -> None:
    # Setup
    autoroutine = DriveStraight(drivetrain, 2)

    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
    autoroutine.run()

    # Assert
    if at_Setpoint:
        autoroutine.drivetrain.drive.arcadeDrive.assert_called_once_with(0, 0)
    else:
        autoroutine.drivetrain.drive.arcadeDrive.assert_called_once()
```

Homework

Given the class below for a *climb the ramp* autonomous. Write a unit test for this class. Be sure to test that the robot moves through all the various of "Not on the ramp yet", "On the ramp", and "at the top" appropriately. Write tests for three of the four methods `drive_straight`, `did_tip_up`, `reached_top`, `reset`. Use the tools of MagicMock, Fixtures, MonkeyPatching, and parameters as necessary.

```python
from wpimath.controller import PIDController

from autoroutine import AutoRoutine
from drivetrain import Drivetrain


class ClimbRamp(AutoRoutine):
    forward_rate = .8
    ended_ramp = False
    started_ramp = False

    def __init__(self, drivetrain: Drivetrain):
        self.drivetrain = drivetrain
        self.drivetrain.resetGyro()
        self.direction_controller = PIDController(4 / 10000, 2 / 10000, 0)
        self.direction_controller.setSetpoint(0)
        self.direction_controller.setTolerance(10)
        self.drivetrain.resetEncoders()
        self.reset()

    def run(self):
        if not (self.started_ramp or self.ended_ramp):
            self.drive_straight()
            self.started_ramp = self.did_tip_up()
        elif self.started_ramp and not self.ended_ramp:
            self.drive_straight()
            self.ended_ramp = self.reached_top()
        else:
            self.drivetrain.arcadeDrive(0, 0)

    def drive_straight(self):
        error = self.drivetrain.getLeftEncoderCount() - \
                self.drivetrain.getRightEncoderCount()
        rotate = self.direction_controller.calculate(error)
        at_set_point = self.direction_controller.atSetpoint()

        print(f"{at_set_point=} {rotate=} {error=}")
        if not at_set_point:
            self.drivetrain.arcadeDrive(rotate, self.forward_rate)
        else:
            self.drivetrain.arcadeDrive(0, self.forward_rate)

    def did_tip_up(self) -> bool:
        tip = self.drivetrain.getGyroAngleY()
        print(f"{tip=}")
        if tip > 7:
            print("On Ramp")
            self.forward_rate = .5
            return True
        return False

    def reached_top(self) -> bool:
        tip = self.drivetrain.getGyroAngleY()
        print(f"{tip=}")
        if tip < 4:
            print("Finished Ramp")
            self.forward_rate = 0
```

```
            return True
        return False

    def reset(self) -> None:
        self.ended_ramp = False
        self.started_ramp = False
        self.forward_rate = .8
```