Day 23 CS570

Unit Testing Parameterization and Practice

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.

Examples from the Code

Here is the code from a subsystem that we is in the current code for the `Elevator` subsystem. Take a look at the code, and I would like to you write a test for one of the methods in this class. For extra practice you might look to write a fixture to help test the `Elevator` class.

```
import rev
import config
import constants

from units.SI import meters
import ntcore
from toolkit.subsystem import Subsystem
from toolkit.motors.rev_motors import SparkMax
import robot_states as states

class Elevator(Subsystem):

    def __init__(self) -> None:
        super().__init__()
        # Absolute encoder
        self.motor_extend: SparkMax = SparkMax(
            config.elevator_can_id, config=config.ELEVATOR_CONFIG, inverted=False, config_others=[config.EL
        )
        self.motor_extend_encoder = None

        self.motor_extend_follower: SparkMax = SparkMax(
            config.elevator_can_id_2, config=config.ELEVATOR_CONFIG, inverted=True, config_others=[config.E
        )

        self.zeroed: bool = False
        self.elevator_moving: bool = False
        self.locked: bool = False
        self.target_length: meters = 0.0

    def init(self) -> None:
        self.motor_extend.init()
        self.motor_extend.optimize_normal_sparkmax()

        self.motor_extend_follower.init()
        self.motor_extend_follower.optimize_sparkmax_absolute_encoder()

        # Set the motor_extend encoder to the motor's absolute encoder
        self.motor_extend_encoder = self.motor_extend_follower.get_absolute_encoder()

        # Limits motor acceleration
        self.motor_extend.motor.setClosedLoopRampRate(config.elevator_ramp_rate)

    # Static methods are methods that don't need to access the instance
    # variables of a class. They are used to perform operations that don't
    # depend on the instance variables of a class. So in this case, we use them
    # for conversion functions.
    @staticmethod
    def length_to_rotations(length: meters) -> float:
        return (length * constants.elevator_gear_ratio) / constants.elevator_driver_gear_circumference

    @staticmethod
    def rotations_to_length(rotations: float) -> meters:
        return (rotations * constants.elevator_driver_gear_circumference) / constants.elevator_gear_ratio

    def limit_length(self, length: meters) -> meters:
        if self.locked and length > constants.elevator_max_length_stage:
            return constants.elevator_max_length_stage
        if length > constants.elevator_max_length:
            return constants.elevator_max_length
        elif length < 0.0:
            return 0.0
        return length

    def set_length(self, length: meters, arbff: float = 0) -> None:
        """
        Sets the length of the elevator in meters
        :param length: Length of the elevator (meters)
        :param arbff: feed forward for the elevator
        """
        length = self.limit_length(length)
        self.target_length = length

        print(length)
        print(self.length_to_rotations(length), 'elevator rotation')

        self.motor_extend.set_target_position(
            self.length_to_rotations(length), arbff
        )


    def set_elevator_climb_down(self) -> None:
        """
        Climb down with feed forward
        """
        self.motor_extend.set_raw_output(-.5)

    def get_elevator_current(self) -> float:
        return self.motor_extend.motor.getOutputCurrent()

    def get_length(self) -> float:
        """
        Gets the length of the elevator in meters
        :return: Length of the elevator in meters
        """
        return self.rotations_to_length(self.motor_extend.get_sensor_position())

    def get_length_total_height(self) -> meters:
        return self.get_length() + constants.elevator_bottom_total_height

    def set_motor_extend_position(self, length: meters) -> None:
        """
        Set the position of motor_extend
        :param position: Position of the motor
        """
        length = self.limit_length(length)

        self.motor_extend.set_sensor_position(
            self.length_to_rotations(length)
        )
```

```python
def get_elevator_abs(self) -> meters:
    length = (self.motor_extend_encoder.getPosition() - config.elevator_zeroed_pos) * constants.elevato
    length = 0 if length < 0 else length

    return length

def zero(self) -> None:
    """
    Zero the elevator
    """

    length = self.limit_length(self.get_elevator_abs())

    print(length, 'elevator length (m)')
    # Reset the encoder to zero
    self.set_motor_extend_position(length)

    self.zeroed = True

def set_voltage(self, voltage: float) -> None:
    self.motor_extend.pid_controller.setReference(voltage, rev.CANSparkMax.ControlType.kVoltage)

def get_voltage(self) -> float:
    return self.motor_extend.motor.getAppliedOutput()

def stop(self) -> None:
    """
    Stop the elevator where it is
    """
    self.set_length(self.get_length())

def lock(self) -> None:
    self.locked = True

def unlock(self) -> None:
    self.locked = False

def periodic(self) -> None:
    if config.NT_ELEVATOR:
        table = ntcore.NetworkTableInstance.getDefault().getTable('elevator')

        table.putNumber('elevator height', self.get_length())
        table.putNumber('elevator abs height', self.get_elevator_abs())
        table.putBoolean('elevator moving', self.elevator_moving)
        table.putBoolean('elevator locked', self.locked)
        table.putBoolean('elevator zeroed', self.zeroed)
        table.putNumber('elevator height total', self.get_length_total_height())
        table.putNumber('elevator target height', self.target_length)
        table.putNumber('elevator motor lead applied output', self.motor_extend.motor.getAppliedOutput(
        table.putNumber('elevator motor follow applied output', self.motor_extend_follower.motor.getApp
        table.putNumber('elevator current', self.motor_extend.motor.getOutputCurrent())

    # set drivetrain control speed
    states.drivetrain_controlled_vel = constants.drivetrain_max_vel * max((1 - (self.get_length() / con
    states.drivetrain_controlled_angular_vel = constants.drivetrain_max_angular_vel * max((1 - (self.ge
```