

**Learning to Program our Robot in Python**

https://bit.ly/1895\_Robot\_Python

**Purpose:**

This document helps students to learn how to program in Python by programming a basic driving robot.

**Goal:**

The target audience for this document is new Robotic Team students who have little programming experience. At the end of this document, you should have a robot that drives under control of a joystick and can autonomously drive in a square. A colorful LED strip is incorporated into the project.  
  
The goal of this document is to build code and understand the purpose of most of the code. The full code is provided at the end of the document for support but recommend you walk through the instructions and assemble the code one function at a time.

**Motivation:**

**Learning to program takes patience, time and practice.**

**Programming robots is fun and challenging because you get to be creative, solve problems and physically see the results of your efforts.**

**Approach:**

We will break the lesson down into multiple steps, each expanding the functionality of our robot.

Step 1: Demonstrate Robot Modes of Operation

Step 2: Add Joysticks to the Code

Step 3: Make the Robot Drive

Step 4: Adding Wheel Rotation Sensors

Step 5: Using a Button to Start a Command

Step 6: Adding a Gyro Sensor to the DriveTrain

Step 7: Autonomous drive in square

Step 8: Student Exercise: Add a rangefinder

I know we want to jump right in and make the robot dance but a bunch of background information is needed to really understand what is going on. You will learn more if you take the time to read the background information.

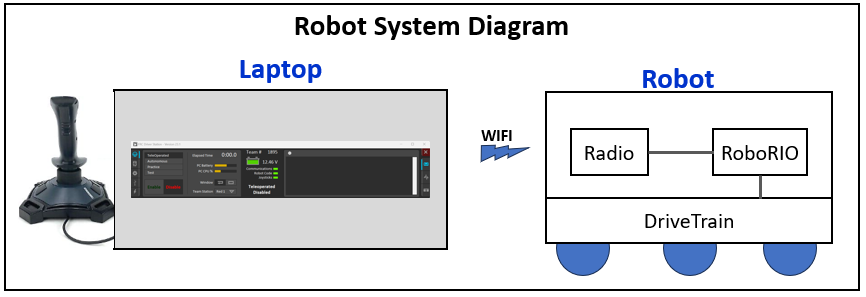
**Preferred Python Prerequisite Skills:**

* Understand that Python is case sensitive
* Understand that Python uses indenting (tabs) to define scope (sections of code)
* Understand that a Python function is a block of code that can accept inputs and return results
* Understand that Python uses libraries to perform tasks

**Require Resources:**

To follow this document, you need the following items:

* A robotics team laptop (which includes the software development tools)
* A joystick
* A practice robot



**References**:

Python (Robotpy)

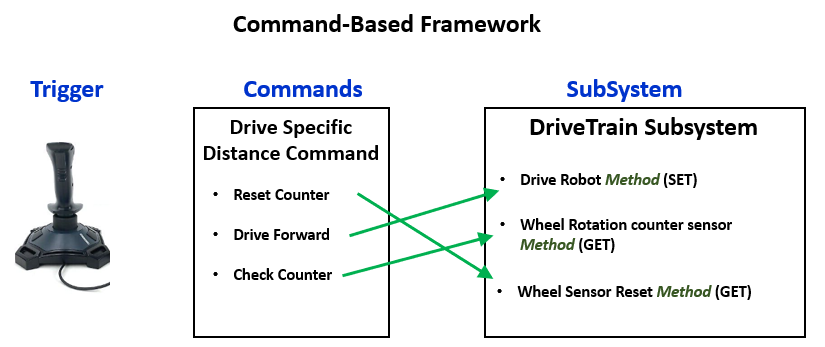
<https://robotpy.readthedocs.io/en/stable/guide/anatomy.html>

Command Based Framework:

<https://docs.wpilib.org/en/latest/docs/software/commandbased/index.html>

**Structure of the Robot Software**

Our team uses a “Command-Based” programming structure. This means we create “**commands”** that tell the robot what we want it to do. For example, we create a command which tells the robot to move forwards, backwards and turn based on the movements of the joystick. Similarly, we can create commands to tell the robot to drive forward a specific distance or turn to a specific angle when we press a button or a collection of commands when the robot is driving autonomously.



To further organize our software, we break the robot down into separate parts such as a drivetrain that moves the robot or an arm that can pick items up. We call these parts “**subsystems**”.

**Object Oriented Programming**

We write our code by creating (instantiating) a model of the robot in the mind of the computer. The model is a software representation of the real physical robot.

We instantiate (create) our whole robot, then instantiate (create) each of the subsystems such as a drivetrain or arm. We can consider the drivetrain an object. This means we can hide the details of what is happening inside the object and only expose a set of actions we would like the subsystem to perform or questions that we can ask it. The details of how it performs the actions or answers the questions are hidden (or not important).

For example, to create a drivetrain, we instantiate the drivetrain subsystem declaring that it has 2 motors to drive its wheels. We then create smaller functions within the drivetrain which **tell** the drivetrain to take some action such as move forward for a specific distance or turn to a specific direction. We can also **ask questions** about the drivetrain. How far has the drivetrain moved, what is the current direction, or how far the robot is from a wall in front of it. In Object Oriented Programming we call these functions GET and SET “methods”.

We expose a limited set of commands and questions which other subsystems or commands can access.

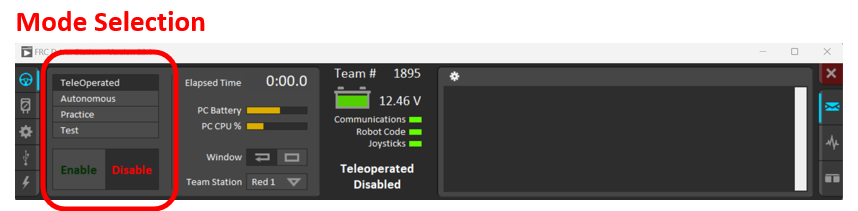
**Modes of Operation**

Our robot operates in three main modes:

* Autonomous: The robot operates on it’s own with a predefined set of commands
* TeleOp: The human commands (drive) the robot
* Disabled: The robot does not move

Practice and Test mode are not needed for basic programming.

The mode of operation is selected on the Laptop DriverStation application.



**How do we Write the Python Code for our Robot**

We write Python code in a software development environment tool called VSCode.

We use VSCode to:

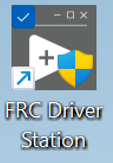
* Organizes our files
* Type in our code (or pasting in example code)
* Offer suggestions to complete your entry (IntelliSense)
* Color codes the keywords in Python and highlights ERRORS
* Deploy the code into the robot

We will learn to write code on a “Practice” robot that is linked to the laptop using a WIFI connection. To control the robot, we will use a “DriverStation” application and a joystick. The DriverStation app reads the joystick movements and buttons and sends them to the controlling computer on the robot (RoboRIO).

**Getting Started!**

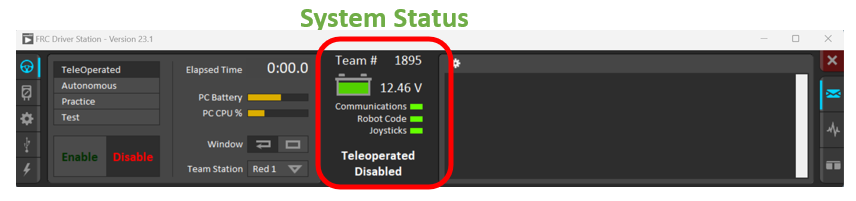
**[ Step 1 ]**

**<< ACTION >>**

**Setting Up the Robot, Laptop, and DriversStation** *[Some assistance may be required]*

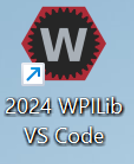
1. Power on the robot
2. Plug the joystick into the laptop
3. On the laptop, double-click on the DriversStation application icon
4. Connect the laptop to the Robot’s WIFI

Three green lights on the DriverStation indicates **Success**

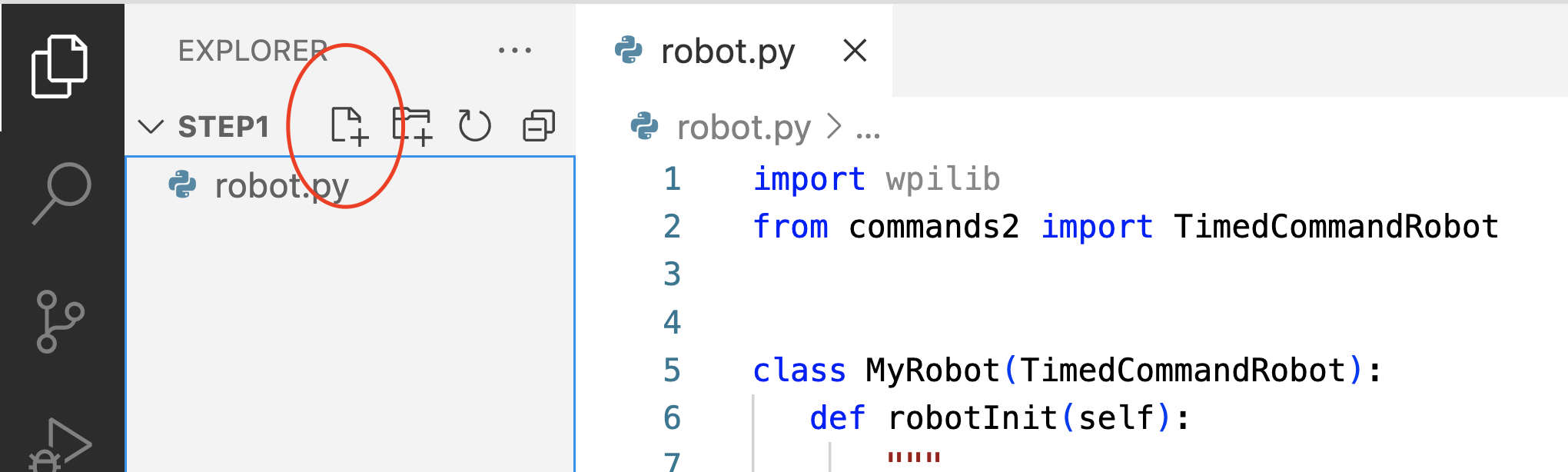


**<< ACTION >>**

**Starting to Code:**

Coding of our robots is performed on our Robotics team laptops. The laptops have tools and libraries needed to write the Python code.

1. Start **VSCode** by double clicking on the VSCode icon.   
   VSCode may start by opening the previous user’s project. Continue with the following steps.
2. Create a new project called “RobotOne”.
3. Within VSCode, select File -> Open folder
4. Browse to c:\\_robotics\2024
5. Select the “New Folder” button
6. Type in “PythonRobot1”
7. Select “Create”
8. Select “Open”
9. Create a new file within the project called “robot.py”
10. Hover the mouse on the left side (under Explorer) which causes 3 icons to appear.
11. Select the left “paper” icon with a plus
12. Type in “robot.py” and press enter.



1. Copy the code in the text box below and paste it into the “robot.py” file.

**Hint:** Click in the number column then select TAB. This will highlight ALL of the python code, then select Control-C.

Highlight the python code only. Exclude the numbering

**[ Step 1 ] [ robot.py ]**

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24** | **import wpilib**  **from commands2 import TimedCommandRobot**  **class MyRobot(TimedCommandRobot):**  **def robotInit(self):**  **"""**  **This function is called upon program startup and**  **should be used for any initialization code.**  **"""**  **print ("Robot Initialization (robotInit) Completed ")**  **def autonomousInit(self):**  **"""This function is run once each time the robot enters autonomous mode."""**  **print ("Autonomous Initialization (autonomousInit) Completed ")**  **def autonomousPeriodic(self):**  **"""This function is called periodically during autonomous."""**  **def teleopInit(self):**  **"""This function is called once each time the robot enters teleoperated mode."""**  **print ("TeleOpInit Initialization (teleopInit) Completed ")**  **def teleopPeriodic(self):**  **"""This function is called periodically during teleoperated mode."""** |
| --- | --- |

**Description of this block of code [robot.py]**

This is minimal code. This block of code only consists of the “MyRobot” class reduced to having functions to initialize the robot (robotInit), initialize each mode of the robot (autonomousInit, teleopInit) and blocks which run periodically while in each move (autonomousPeriodic, teleopPeriodic). A print statement at the end of each initialization block indicates when the function is complete.

At the top of the block of code, the import statement “import wpilib” incorporates the FIRST Robotics Worcester Polytechnic Institute (WPI) Library into the python program. This library establishes communications between the laptop DriverStation application and the robot, implements the modes of the robot (disabled, autonomous, teleOp), interfaces the RoboRIO (robot brain) to the robot hardware and many additional functions.

The triple quotes after each function definition are comments and describe the purpose of each python function. If you hover the mouse of a function, comments are displayed by the development environment.

The VSCode development environment helps you code by continuously checking your work. If you change the first letter of a print statement from lowercase (p) to uppercase (P), VSCode will highlight the error with a yellow squiggle underline. [Try it.]

Throughout the Python code you will see numerous references to “**self**”. This is how Object Oriented Python refers to the current “Instance” of the object you are working on.

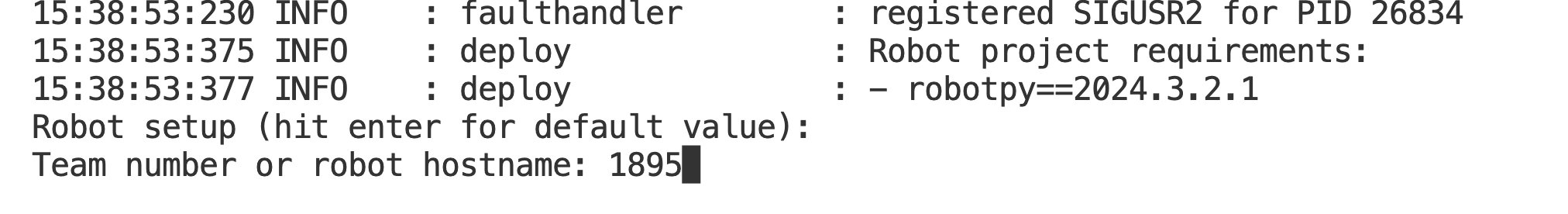
Now that we have our first piece of code within the VSCode on the laptop we need to push it into the Robot.

**<< ACTION >>**

**Deploying the Code to the RoboRIO**

Python is new to FIRST Robotics and is not fully integrated into the VSCode environment like JAVA and C++. A few (hopefully temporary) steps are necessary to deploy the python code to the robot.

1. Within VSCode, Open a terminal window.   
    On the top menu, select **Terminal** then **New Terminal.** *(May need to click on 3 dots)*
2. Deploy the code to the robot using the command:   
    **py3 -m robotpy deploy --skip-tests**
3. When prompted, enter the team number. 1895



The DriverStation “Robot Code” indicator will turn red for a brief time as the new code is deployed and the RoboRio code is restarted.

**<< ACTION >>**

**First Running of the Code**

Configure the DriversStation application to display “print” statements. Locate the “gear” icon near the top of the application, right-side middle. Right-click and select “+**Print**”.

Once the three green indicators on the DriversStation are present, we can run the code

1. On the DriversStation, select “**Autonomous**” mode.
2. Select “**Enable**”
3. In the DriversStation message window, you should see the following statements text mixed in within other text:  
      
   “Robot Initialization (robotInit) Completed”

“Autonomous Initialization (autonomousInit) Completed ”

This indicates the initialization functions have run.

1. Select “**Disable**”
2. Select “**TeleOp**” mode
3. Select “**Enable**”
4. In the DriversStation message window, you should see the print statement text:

TeleOpInit Initialization (teleopInit) Completed"

In the next section, we will add additional functionality to make this code more interesting.

**Adding JoySticks to the Code**

**[ Step 2 ]**

In autonomous mode, our robots drive without human guidance. We program in the desired behaviors. In teleOp mode, humans guide the robot with a joystick or a gamepad.

**<< ACTION >>**  
We will now add a joystick to our code so later we can drive our robot.

1. Add the following code to create line 10 in the robot.py file. (Leave out the number)  
   You can copy and paste the updated lines.

( HINT: Recommend cutting and pasting code since manual typing is error prone)

10: self.controller = wpilib.Joystick(0)

This creates a joystick controller on our robot instance (self). The zero in parenthesis is the first joystick slot which is detected by the DriveStation application. SInce this code is located within the “robotInit” function, it only runs ONE time when the robot code is started.

1. Add the following code to create lines 26 to 28. This reads the joystick X and Y axes and saves the current values in the variables (Xaxis and Yaxis). Since this code is located within the teleOp Periodic function, this code runs 50 times a second.

26: Xaxis = self.controller.getRawAxis(0)

27: Yaxis = self.controller.getRawAxis(1)

28: print("Joystick: Forward Motion Axis: ", Yaxis, " Turning Motion Axis: ", Xaxis )

The full code is below. The added lines of code are highlighted in red.

**[ Step 2 ] [ robot.py ]**

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36** | **import wpilib**  **from commands2 import TimedCommandRobot**  **class MyRobot(TimedCommandRobot):**  **def robotInit(self):**  **"""**  **This function is called upon program startup and**  **should be used for any initialization code.**  **"""**  **self.controller = wpilib.Joystick(0)**  **print ("Robot Initialization (robotInit) Completed ")**  **def autonomousInit(self):**  **"""This function is run once each time the robot enters autonomous mode."""**  **print ("Autonomous Initialization (autonomousInit) Completed ")**  **def autonomousPeriodic(self):**  **"""This function is called periodically during autonomous."""**  **def teleopInit(self):**  **"""This function is called once each time the robot enters teleoperated mode."""**  **print ("TeleOpInit Initialization (teleopInit) Completed ")**  **def teleopPeriodic(self):**  **"""This function is called periodically during teleoperated mode."""**  **Xaxis = self.controller.getRawAxis(0)**  **Yaxis = self.controller.getRawAxis(1)**  **print("Joystick: Forward Motion Axis: ", Yaxis, " Turning Motion Axis: ", Xaxis )**  **def testInit(self):**  **"""This function is called once each time the robot enters test mode."""**  **print ("TestInit Initialization (testInit) Completed ")**  **def testPeriodic(self):**  **"""This function is called periodically during test mode."""** |
| --- | --- |

**Run the Updated Code**

Deploy the updated code to the robot and try teleOp mode as previously described.   
(Hint: enter **py3 -m robotpy deploy --skip-tests** in a terminal window.)  
(**Better Hint**: Hit the up arrow and the last command is recalled)

While in ENABLED TeleOp mode, you should see a print statement displaying the current joystick values.

Move the joystick in the X and Y directions and see the numbers change.

<<< [ Insert a sample of the output here ]>>>

**Making our Robot Drive**

**[ Step 3 ]**

Robots can be complex. One approach to make a complex item easier to understand is to break the items into smaller more manageable pieces. We typically organize our robot into subsystems such as the “DriveTrain”, “Arm”, “Intake”, “LEDs” and so forth. We can then focus our thinking power on one smaller item and share the work with multiple people.

To make our robot move, we will create a “drivetrain” **subsystem** and a drivetrain **command** which reads the joystick axes values and uses them to drive the robot.

Here is an overview of the step to create a drivetrain subsystem:

1. Add a **DriveTrain** subsystem file (drivetrainsubsys.py)
2. Add a **command** to control the DriveTrain subsystem (teleopdrivecmd.py)
3. Update the original robot.py file to instantiate (create) the drivetrain subsystem and the TeleOp command

**<< ACTION >>**

**DriveTrain Subsystem**

1. Add a file to our project called “drivetrainsubsys.py” as described above
2. Copy the code from below and place it into the file

**[ Step 3 ] [drivetrainsubsys.py]**

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66**  **67**  **68**  **69**  **70**  **71**  **72**  **73**  **74**  **75**  **76**  **77**  **78** | **import wpilib**  **from commands2 import Subsystem, Command**  **import wpilib.drive**  **from phoenix6.configs import TalonFXConfiguration**  **from phoenix6.hardware.talon\_fx import TalonFX**  **from phoenix6.signals.spn\_enums import (InvertedValue,NeutralModeValue)**  **from phoenix6.controls import DutyCycleOut**  **from phoenix6 import StatusCode**  **class DriveTrain(Subsystem):**  **def \_\_init\_\_(self) -> None:**  **super().\_\_init\_\_() # Call the parent's (Super) initialization function**  **# Apply all the configurations to the left and right side Talon motor controllers**  **self.\_\_configure\_left\_side\_drive()**  **self.\_\_configure\_right\_side\_drive()**  **print ("DriveTrain Subsystem Initialization complete")**  **def \_\_configure\_left\_side\_drive(self) -> None:**  **self.\_left\_leader\_motor = TalonFX(1) # 1 is the CAN bus address**  **# Applying a new configuration will erase all other config settings since**  **# we start with a blank config so each setting needs to be explicitly set**  **# here in the config method**  **config = TalonFXConfiguration()**  **# Set the left side motors to be counter clockwise positive**  **config.motor\_output.inverted = InvertedValue.COUNTER\_CLOCKWISE\_POSITIVE**  **# Set the motors to electrically stop instead of coast**  **config.motor\_output.neutral\_mode = NeutralModeValue.BRAKE**  **# Apply the configuration to the motors**  **for i in range(6): # Try 5 times**  **ret = self.\_left\_leader\_motor.configurator.apply(config)**    **def \_\_configure\_right\_side\_drive(self) -> None:**  **self.\_right\_leader\_motor = TalonFX(2) # 2 is the CAN bus address**  **# Applying a new configuration will erase all other config settings since**  **# we start with a blank config so each setting needs to be explicitly set**  **# here in the config method**    **config = TalonFXConfiguration()**  **# Set the left side motors to be counter clockwise positive**  **config.motor\_output.inverted = InvertedValue.CLOCKWISE\_POSITIVE**  **# Set the motors to electrically stop instead of coast**  **config.motor\_output.neutral\_mode = NeutralModeValue.BRAKE**  **# Apply the configuration to the motor with a 5 second timeout**  **for i in range(6): # Try 5 times**  **ret = self.\_right\_leader\_motor.configurator.apply(config)**  **########################### Drivetrain Drive methods #######################**  **def drive\_teleop(self, forward: float, turn: float):**  **reduction = 0.5**  **turn = turn \* reduction**  **forward = forward \* reduction**  **speeds = wpilib.drive.DifferentialDrive.curvatureDriveIK(forward, turn, True)**  **print ("Wheel Speeds: Left:", speeds.left, " Right: ", speeds.right)**  **self.\_left\_percent\_out: DutyCycleOut = DutyCycleOut(0)**  **self.\_right\_percent\_out: DutyCycleOut = DutyCycleOut(0)**  **self.\_left\_percent\_out.output = speeds.left**  **self.\_right\_percent\_out.output = speeds.right**  **self.\_left\_leader\_motor.set\_control(self.\_left\_percent\_out)**  **self.\_right\_leader\_motor.set\_control(self.\_right\_percent\_out)** |
| --- | --- |

**The code "drivetrainsubsys.py" does the following:**

Lines 1 to 10 import a number of Python Libraries. Adding these libraries allows you to use Python code that someone else has written which implements the command based code, subsystems and control our smart motors (Falcon 500s) using the Phoenix 6 library.

The DriveTrain subsystem is defined by a "Class". This is how we create an Object.

The "\_\_init\_\_" function code runs one time when we run the code. We use this function to initialize the smart motors. First the left side then the right side.

Below in the "drive\_teleop" function, we actually move the robot using a library where we tell the motors how to move based on the position of the joystick.

There is a print statement that prints the motor speeds to the console output.

**<< ACTION >>**

**TeleOp Command for the DriveTrain**

1. Add a file to our project called “teleopdrivecmd.py”
2. Copy the code from below and place it into the file

**[ Step 3 ] [ teleopdrivecmd.py ]**

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24** | **import wpilib**  **import wpilib.drive**  **from commands2 import Command**  **from drivetrainsubsys import DriveTrain**  **class TeleopDrive(Command):**  **def \_\_init\_\_(self, drivetrain: DriveTrain, controller: wpilib.Joystick):**  **self.drivetrain = drivetrain**  **self.addRequirements(self.drivetrain)**  **self.controller = controller**  **print ("TeleOpDrive Command Instantiated")**  **def initialize(self):**  **print ("TeleOpDrive Command Initialized")**  **def execute(self):**  **self.drivetrain.drive\_teleop(-self.controller.getRawAxis(1),**  **-self.controller.getRawAxis(0))**  **def isFinished(self) -> bool:**  **return False**    **def end(self, interrupted: bool):**  **self.drivetrain.drive\_teleop(0,0)** |
| --- | --- |

The "**teleopdrive.py**" file contains the TeleopDrive command.

We call this command when we want the robot to move.

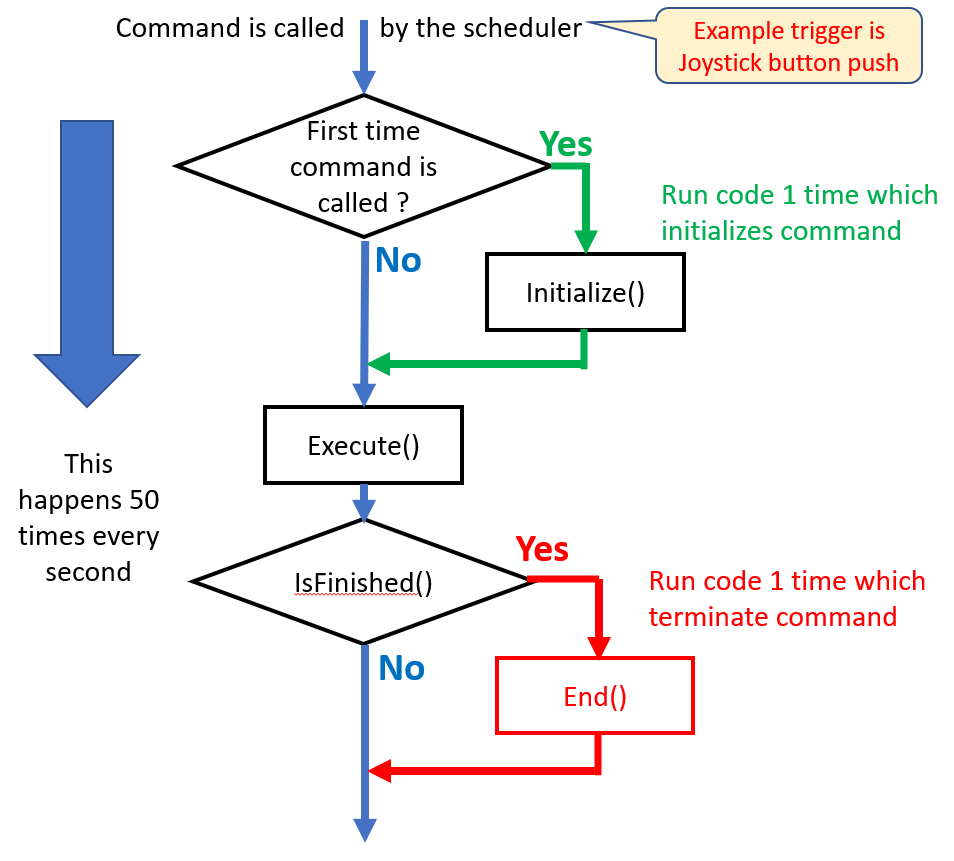
Robot Commands all have a similar structure. The "\_\_init\_\_" function is called when the command is instantiated.

The "Initialize" function is called each time the command is called (like when a button is pushed). In this specific case, nothing really occurs but a simple print statement.

The "execute" function is the heart of command. The code in this function (block of code) is run 50 times per second. Here we read sensors, make decisions and control motors or actuators.

After each "execute" function is run, the "IsFinished" function asks if we are done with the command. Some sensor is read or a previous check is reviewed. If we end up with a False boolean value (No we are not done), the command will be run again. If we say yes or True, then the command is complete and we run the "end" function to end the command like stopping motors or turning off lights.

The “Flow Chart” below illustrates how “Commands” work.



**Updates to robot.py**

**<< ACTION >>**

Updated the original **robot.py** file as follows:

* Update lines 2 to 4 to import (include) the new subsystem and command python files.
* Update line 13 - 18 to instantiate (create) the drivetrain subsystem and set the “TeleOpCommand” command as the “Default Command”. This means this command runs when no other command is running.
* Add lines 22 - 25 to set the default command for the drivetrain subsystem.  
    
  A subsystems default command runs all the time except when another command for the same subsystem is called.
* Comment out the print commands for the joystick lines 39-41 by placing a “hashtag” (pound-sign) at the beginning of the line. Commenting out a like of code will prevent these lines of code from running and cluttering up our output.

**HINT:** One or more lines can be commented out by highlighting the lines and selecting the following TWO keys: **control - /** *(Control key and the slash key at the same time)*

**Key Concept**

In each mode of operation, there is a **Default Command** assigned to each subsystem. When we switch to a new mode (Disable to TeleOp or Disabled to Autonomous), the default command will perform “Initialization” actions to set things up. Once initialization is complete, the robot will execute a defined set of actions over and over again until the task is complete. This looping is performed 50 times each second.

**[ Step 3 ] [ robot.py ]**

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41** | **import wpilib**  **from drivetrainsubsys import DriveTrain**  **from teleopdrivecmd import TeleopDrive**  **from commands2 import Command, RunCommand, TimedCommandRobot**  **class MyRobot(TimedCommandRobot):**  **def robotInit(self):**  **"""**  **This function is called upon program startup and**  **should be used for any initialization code.**  **"""**  **self.controller = wpilib.Joystick(0)**  **# Instantiate (create) subsystems**  **self.drivetrainSubSys: DriveTrain = DriveTrain()**  **# Configure commands**  **self.configure\_default\_commands()**  **print ("Robot Initialization (robotInit) Completed ")**  **def configure\_default\_commands(self) -> None:**  **self.drivetrainSubSys.setDefaultCommand(**  **TeleopDrive(self.drivetrainSubSys, self.controller)**  **)**  **def autonomousInit(self):**  **"""This function is run once each time the robot enters autonomous mode."""**  **print ("Autonomous Initialization (autonomousInit) Completed ")**  **def autonomousPeriodic(self):**  **"""This function is called periodically during autonomous."""**  **def teleopInit(self):**  **"""This function is called once each time the robot enters teleoperated mode."""**  **print ("TeleOpInit Initialization (teleopInit) Completed ")**  **def teleopPeriodic(self):**  **"""This function is called periodically during teleoperated mode."""**  **# Xaxis = self.controller.getRawAxis(0)**  **# Yaxis = self.controller.getRawAxis(1)**  **# print("Joystick: Forward Motion Axis: ", Yaxis, " Turning Motion Axis: ", Xaxis )** |
| --- | --- |

**Deploy and test the DriveTrain Subsystem**  *(The wheels should move)*

**SAFETY FIRST:**

Verify the robot is on blocks and the wheels can spin without touching anything  
  
Please alert everyone that you are enabling a robot by saying “ENABLING” loudly.

Test the current code using the following steps:

1. Deploy the code
2. Select TeleOp mode and enable the robot.
3. Move the Joystick and verify the wheels move.

**Adding Wheel Rotation Sensors to our Code**

**[ Step 4 ]**

The motors which drive our robot have built-in sensors which count the number of times the motor/wheels rotate. We use this sensor to control how far our robot moves forward when commanded.

We will implement the following 3 changes to the code:

* We will enable the sensors on the Falcon 500 motors to count wheel revolutions within the drivetrain subsystem.
* We will also create our first **sensor based** command. This command when called will drive the robot forward 1 wheel revolution.
* The main program robot.py is updated to call the drive command when switched to **autonomous** mode.

The current value of the wheel/motor sensor (encoders) is printed on each loop of the code.

**Update the DriveTrain system using the following steps:**

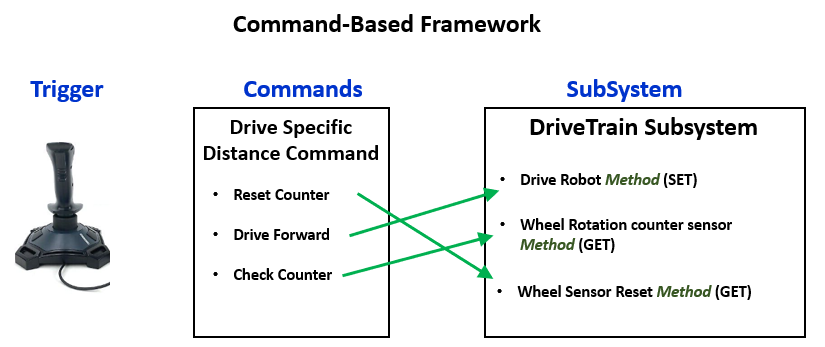
**<< ACTION >>**

* Update line 7-8 to include a feedback sensor function in the library. (Replace the two lines)
* Add lines 35 - 38. This code enables the rotation sensor and tells the sensor that the motor will spin 10 times to make the wheel rotate one time. This sets the left side of the robot.
* Update lines 59 - 62 with the same code as lines 35-38. This sets the right side of the robot.
* Add line 44 and line 68. This code will reset the sensor to zero. We would like the sensor set at zero when we start counting.
* Add lines 92 - 107.

We are adding functions that allow an external command to read the counters and reset them back to zero.

**<< IMPORTANT CONCEPT >>**

These functions are called "Methods" in Object Oriented Programming. This is how we talk to Objects. We can ask objects questions ("GET COUNTS") and we can give objects commands ("RESET COUNTERS") in a structured approach. The outside entity does not need to know what happens inside the object, Just how to access the "GETS" and "SETS".



**[ Step 4 ] [drivetrainsubsys.py] - Adding Encoder sensors and methods**

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66**  **67**  **68**  **69**  **70**  **71**  **72**  **73**  **74**  **75**  **76**  **77**  **78**  **79**  **80**  **81**  **82**  **83**  **84**  **85**  **86**  **87**  **88**  **89**  **90**  **91**  **92**  **93**  **94**  **95**  **96**  **97**  **98**  **99**  **100**  **101**  **102**  **103**  **104**  **105**  **106**  **107**  **108** | **import wpilib**  **from commands2 import Subsystem, Command**  **import wpilib.drive**  **from phoenix6.configs import TalonFXConfiguration**  **from phoenix6.hardware.talon\_fx import TalonFX**  **from phoenix6.signals.spn\_enums import (**  **InvertedValue, NeutralModeValue, FeedbackSensorSourceValue )**  **from phoenix6.controls import DutyCycleOut**  **from phoenix6 import StatusCode**  **class DriveTrain(Subsystem):**  **def \_\_init\_\_(self) -> None:**  **super().\_\_init\_\_() # Call the parent's (Super) initialization function**  **# Apply all the configurations to the left and right side Talon motor controllers**  **self.\_\_configure\_left\_side\_drive()**  **self.\_\_configure\_right\_side\_drive()**  **print ("DriveTrain Subsystem Initialization complete")**  **def \_\_configure\_left\_side\_drive(self) -> None:**  **self.\_left\_leader\_motor = TalonFX(1) # 1 is the CAN bus address**  **# Applying a new configuration will erase all other config settings since**  **# we start with a blank config so each setting needs to be explicitly set**  **# here in the config method**  **config = TalonFXConfiguration()**  **# Set the left side motors to be counter clockwise positive**  **config.motor\_output.inverted = InvertedValue.COUNTER\_CLOCKWISE\_POSITIVE**  **# Set the motors to electrically stop instead of coast**  **config.motor\_output.neutral\_mode = NeutralModeValue.BRAKE**  **# This configuration item supports counting wheel rotations**  **# This item sets the gear ratio between motor turns and wheel turns**  **config.feedback.feedback\_sensor\_source = FeedbackSensorSourceValue.ROTOR\_SENSOR**  **config.feedback.sensor\_to\_mechanism\_ratio = 10**  **# Apply the configuration to the motors**  **for i in range(6): # Try 5 times**  **ret = self.\_left\_leader\_motor.configurator.apply(config)**  **self.\_left\_leader\_motor.set\_position(0) # Reset the encoder to zero**  **def \_\_configure\_right\_side\_drive(self) -> None:**  **self.\_right\_leader\_motor = TalonFX(2) # 2 is the CAN bus address**  **# Applying a new configuration will erase all other config settings since**  **# we start with a blank config so each setting needs to be explicitly set**  **# here in the config method**  **config = TalonFXConfiguration()**  **# Set the left side motors to be counter clockwise positive**  **config.motor\_output.inverted = InvertedValue.CLOCKWISE\_POSITIVE**  **# Set the motors to electrically stop instead of coast**  **config.motor\_output.neutral\_mode = NeutralModeValue.BRAKE**  **# This configuration item supports counting wheel rotations**  **# This item sets the gear ratio between motor turns and wheel turns**  **config.feedback.feedback\_sensor\_source = FeedbackSensorSourceValue.ROTOR\_SENSOR**  **config.feedback.sensor\_to\_mechanism\_ratio = 10**  **# # Apply the configuration to the motors**  **for i in range(6): # Try 5 times**  **ret = self.\_right\_leader\_motor.configurator.apply(config)**  **self.\_left\_leader\_motor.set\_position(0) # Reset the encoder to zero**  **########################### Drivetrain Drive methods #######################**  **def drive\_teleop(self, forward: float, turn: float):**  **# Slow the robot down by the reduction value**  **reduction = 0.5**  **turn = turn \* reduction**  **forward = forward \* reduction**  **speeds = wpilib.drive.DifferentialDrive.curvatureDriveIK(forward, turn, True)**  **# print ("Wheel Speeds: Left:", speeds.left, " Right: ", speeds.right)**  **self.\_left\_percent\_out: DutyCycleOut = DutyCycleOut(0)**  **self.\_right\_percent\_out: DutyCycleOut = DutyCycleOut(0)**  **self.\_left\_percent\_out.output = speeds.left**  **self.\_right\_percent\_out.output = speeds.right**  **self.\_left\_leader\_motor.set\_control(self.\_left\_percent\_out)**  **self.\_right\_leader\_motor.set\_control(self.\_right\_percent\_out)**  **########################### Drive Encoder methods #######################**  **def get\_left\_side\_encoder\_count(self):**  **self.\_left\_leader\_motor.get\_rotor\_position()**  **return self.\_left\_leader\_motor.get\_position().value**  **def get\_right\_side\_encoder\_count(self):**  **self.\_right\_leader\_motor.get\_rotor\_position()**  **return self.\_right\_leader\_motor.get\_position().value**    **def reset\_left\_side\_encoder\_count(self):**  **self.\_left\_leader\_motor.set\_position(0) # Reset the encoder to zero**  **def reset\_right\_side\_encoder\_count(self):**  **self.\_right\_leader\_motor.set\_position(0) # Reset the encoder to zero** |
| --- | --- |

We will use the "AutoDriveXWheelCounts" command to learn several new concepts.

**Key Concepts:**

* The python filename is a lower-case copy of the Class Name
* The command must have the required subsystem imported to allow it to be referenced
* The phrase "**from** drivetrainsubsys **import** DriveTrain" means   
  **From the FILE** drivetrainsubsys.py **import the CLASS DriveTrain**
* Remember the filename and class names are case sensitive

When a subsystem or command is created (instantiated) the **\_\_init\_\_** function is called. Within the parentheses, values (parameters) can be passed into the subsystem or command. They are separated with commas.

**def \_\_init\_\_(self, drivetrain: DriveTrain, targetwheelcounts: float, forwardSpeed: int):**

The first parameter is "**self**". This is the name of the instance of this object. This is always passed into a fuction. For robot commands, the second parameter is usually the subsystem. Additional parameters can be passed such as the desired speed of the robot or the direction to turn.

The word after the colon is a “**Type Hint**”. This is used by the Software Development Environment (VSCode) to help you (the human) by making sure you are using the variable correctly. Python does require the programmer to declare the type of each variable created. Examples of variable types are integers (whole numbers), strings (words) and booleans (True/False). JAVA and C require variables to be “types” defined when the variable is first used. Python does not.

The **\_\_init\_\_** function assigns these passed-in parameters to instance specific values. These are preceded with "self." meaning the value for the instance we just created (instantiated).

The WPI library requires that we identify which subsystem is being used by the command. This prevents conflicting commands (Two commands given to the same subsystem at the same time). The identification is performed with the "**self.addRequirements**" function.

The initialize, execute, isFinished and end functions are the same as described above.

For this "AutoDriveXWheelCounts" command we are accepting the number of wheel rotation counts (self.**targetwheelcounts**) and robot drive speed (self.**forwardSpeed**) as parameters passed into this command when it is instantiated (created).

To make this command work, we are reading the left wheel rotation sensor. We reset the counter to zero each time the command is called, in each loop we set the motor speed to the value passed in. Within the "isFinished" function we read the left side counter and see if we have reached our target value. When we reach the target value, we finish the command and stop the motors.

**<< ACTION >>**

Add the **AutoDriveXWheelCounts** Command using the following steps:

* Create a file called autodrivexwheelcounts.py
* place the code into the file.

**[ Step 4 ]** [ autodrivexwheelcounts.py ]

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32** | **import wpilib**  **import wpilib.drive**  **from commands2 import Command**  **from drivetrainsubsys import DriveTrain**  **class AutoDriveXWheelCounts(Command):**  **def \_\_init\_\_(self, drivetrain: DriveTrain, targetwheelcounts: float, forwardSpeed: int):**  **self.drivetrain = drivetrain**  **self.targetwheelcounts = targetwheelcounts**  **self.forwardSpeed = forwardSpeed**  **self.addRequirements(self.drivetrain)**  **self.drivetrain.reset\_left\_side\_encoder\_count()**  **print ("Driving for ", targetwheelcounts, " Wheel Counts Command Initialized")**  **def initialize(self):**  **super().initialize()**  **self.drivetrain.reset\_left\_side\_encoder\_count()**  **def execute(self):**  **self.drivetrain.drive\_teleop(self.forwardSpeed, 0.0)**  **# print ("Wheel Counts: ", self.drivetrain.get\_left\_side\_encoder\_count())**  **def isFinished(self) -> bool:**  **if self.drivetrain.get\_left\_side\_encoder\_count() >= self.targetwheelcounts:**  **return True**  **else:**  **return False**    **def end(self, interrupted: bool):**  **self.drivetrain.drive\_teleop(0,0)** |
| --- | --- |

Update the **Robot.py** file with to call the "AutoDriveXWheelsCounts" class in autonomous mode using the following steps:

**<< ACTION >>**

* Update line 5 to include the "AutoDriveXWheelsCounts" Class
* Update lines 30-32 to set the "AutoDriveXWheelsCounts" Class as the autonomous mode command

**[ Step 4 ] [ robot.py ]**

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57** | **import wpilib**  **from drivetrainsubsys import DriveTrain**  **from teleopdrivecmd import TeleopDrive**  **from commands2 import Command, RunCommand, TimedCommandRobot**  **from autodrivexwheelcounts import AutoDriveXWheelCounts**  **from typing import Tuple, List**  **class MyRobot(TimedCommandRobot):**  **def robotInit(self):**  **"""**  **This function is called upon program startup and**  **should be used for any initialization code.**  **"""**  **# Instantiate (create) subsystems**  **self.drivetrainSubSys: DriveTrain = DriveTrain()**  **self.controller = wpilib.Joystick(0)**  **# Configure commands**  **self.configure\_default\_commands()**  **print ("Robot Initialization (robotInit) Completed ")**  **def configure\_default\_commands(self) -> None:**  **self.drivetrainSubSys.setDefaultCommand(**  **TeleopDrive(self.drivetrainSubSys, self.controller)**  **)**  **def getAutonomousCommand(self) -> Command:**  **# Configure command to drive when autonomous is selected (1 second at 50% speed)**  **return AutoDriveXWheelCounts(self.drivetrainSubSys, 1, 0.5)**  **def autonomousInit(self):**  **"""This function is run once each time the robot enters autonomous mode."""**  **self.\_auto\_command = self.getAutonomousCommand()**  **if self.\_auto\_command is not None:**  **self.\_auto\_command.schedule()**  **print ("Autonomous Initialization (autonomousInit) Completed ")**  **def autonomousPeriodic(self):**  **"""This function is called periodically during autonomous."""**  **def teleopInit(self):**  **"""This function is called once each time the robot enters teleoperated mode."""**  **print ("TeleOpInit Initialization (teleopInit) Completed ")**  **def teleopPeriodic(self):**  **"""This function is called periodically during teleoperated mode."""**  **# Xaxis = self.controller.getRawAxis(0)**  **# Yaxis = self.controller.getRawAxis(1)**  **# print("Joystick: Forward Motion Axis: ", Yaxis, " Turning Motion Axis: ", Xaxis )** |
| --- | --- |

**Using a Button to Start a Command**

**[ Step 5 ]**

We will now learn how to make the robot perform a command when a button is pressed.

When button 1 (Joystick Trigger Button) is pressed, our robot will drive forward for 1 second.   
Linking a command to a button is called “Button Bindings”.

To bind a button to a command, we will add:

* A new command to drive the wheels for a specific amount of time.
* Update the main program, robot.py, to bind the joystick button to a command

The new command AutoDriveXSeconds just makes the robot drive forward for a specific number of seconds. We are passing in as parameters the number of seconds and the speed which we would like the robot to move. This is a pretty simple command. We multiply the number of seconds passed into the command by 50 and save this value. (The number 50 is used since our code runs 50 times a second). Within the “isFinshed” function, we reduce this number (decrement) each time and compare it to zero. When we get to zero the command is complete and we stop the motor in the “end” function.

We update the robot.py file to bind a joystick button to the command. When the #1 button is pressed we will call the “**AutoDriveXWheelCounts**” command passing in the parameters to turn the wheels at 30% speed for 1 second. The “**onTrue**” means when the button is pressed the command will run. There are other options like “WhileTrue” which will run the command while the button is pressed and stop running the command when the button is released.

**def \_\_configure\_button\_bindings(self) -> None:**

**button.JoystickButton(self.controller,1).onTrue(**

**AutoDriveXSeconds(self.drivetrainSubSys, 1, 0.3)**

**)**

**<< ACTION >>**

**Update the Code:**

* Update lines 23 and 30-33 of “robot.py” to bind the button to the command.

**[ Step 5 ]** [autodrivexseconds.py]

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32** | **import wpilib**  **import wpilib.drive**  **from commands2 import Command**  **from drivetrainsubsys import DriveTrain**  **class AutoDriveXSeconds(Command):**  **def \_\_init\_\_(self, drivetrain: DriveTrain, seconds: int, forwardSpeed: float):**  **self.drivetrain = drivetrain**  **self.seconds = seconds**  **self.forwardSpeed = forwardSpeed**  **self.counter = 50 \* seconds**  **self.addRequirements(self.drivetrain)**  **print ("Driving for ", seconds, " Command Instantiated")**  **def initialize(self):**  **super().initialize()**  **self.countDownCounter = self.counter**  **print ("Driving for Command Initialized !!!!")**  **def execute(self):**  **self.drivetrain.drive\_teleop(self.forwardSpeed, 0.0)**  **self.countDownCounter = self.countDownCounter - 1**  **def isFinished(self) -> bool:**  **if self.countDownCounter <= 0:**  **return True**  **else:**  **return False**  **def end(self, interrupted: bool):**  **self.drivetrain.drive\_teleop(0,0)** |
| --- | --- |

**[ Step 5 ] [ robot.py ]** - binding button to command

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66** | **import wpilib**  **from drivetrainsubsys import DriveTrain**  **from teleopdrivecmd import TeleopDrive**  **from commands2 import Command, RunCommand, TimedCommandRobot, button**  **from autodrivexseconds import AutoDriveXSeconds**  **from autodrivexwheelcounts import AutoDriveXWheelCounts**  **from typing import Tuple, List**  **class MyRobot(TimedCommandRobot):**  **def robotInit(self):**  **"""**  **This function is called upon program startup and**  **should be used for any initialization code.**  **"""**  **# Instantiate (create) subsystems**  **self.drivetrainSubSys: DriveTrain = DriveTrain()**  **self.controller = wpilib.Joystick(0)**  **self.\_\_configure\_button\_bindings()**  **# Configure commands**  **self.configure\_default\_commands()**  **print ("Robot Initialization (robotInit) Completed ")**  **def \_\_configure\_button\_bindings(self) -> None:**  **button.JoystickButton(self.controller,1).onTrue(**  **AutoDriveXSeconds(self.drivetrainSubSys, 1, 0.3)**  **)**  **def configure\_default\_commands(self) -> None:**  **self.drivetrainSubSys.setDefaultCommand(**  **TeleopDrive(self.drivetrainSubSys, self.controller)**  **)**  **def getAutonomousCommand(self) -> Command:**  **return AutoDriveXWheelCounts(self.drivetrainSubSys, 1, 0.5)**  **def autonomousInit(self):**  **"""This function is run once each time the robot enters autonomous mode."""**  **self.\_auto\_command = self.getAutonomousCommand()**  **if self.\_auto\_command is not None:**  **self.\_auto\_command.schedule()**  **print ("Autonomous Initialization (autonomousInit) Completed ")**  **def autonomousPeriodic(self):**  **"""This function is called periodically during autonomous."""**  **def teleopInit(self):**  **"""This function is called once each time the robot enters teleoperated mode."""**  **print ("TeleOpInit Initialization (teleopInit) Completed ")**  **def teleopPeriodic(self):**  **"""This function is called periodically during teleoperated mode."""**  **# Xaxis = self.controller.getRawAxis(0)**  **# Yaxis = self.controller.getRawAxis(1)**  **# print("Joystick: Forward Motion Axis: ", Yaxis, " Turning Motion Axis: ", Xaxis )** |
| --- | --- |

**Adding a Gyro Sensor to the DriveTrain**

**[ Step 6 ]**

We will now add a Gyro Sensor to the DriveTrain to control robot turning. A gyro is similar to a compass in that it determines the direction the robot is pointing. A gyro sensor can report the value to our python program. We use this capability to turn the robot to a specific direction without a human controlling the robot.

We will create a turn command (which accepts a turn direction and robot turn speed) and bind this to a button.

To make the robot turn when we press a button, we will:

* Add a Gyro Sensor to the DriveTrain (drivetrainsubsys.py)
* A new command to turn the robot a specific number of degrees (autoturnxdegrees.py)
* Update the main program, robot.py, to bind the joystick button to the turn command (robot.py)

**<< ACTION >>**

**Changes to the DriveTrain Class**

* Add lines 13 - 14: To add the gyro to the drivetrain we need to add the Gyro's manufacturer's python library plus a math library for some needed calculations
* Line 26: Instantiate the gyro as part of the drivetrain. We will call the gyro "self.gyro" with a Type Hint of navx.AHRS
* Line 117-131: Add functions to "Get" the current heading and a function to "Set" the gyro to zero prior to starting a turn

Since robots can turn to the left or right, we need to consider positive and negative values. Turns to the right (Clockwise) are considered negative values. To turn to the left, we request a positive turn value. A math calculation is performed to keep the "Get" value between 180 and -180.

**Adding a new AutoTurnXDegrees Command**

The AutoTurnXDegrees command accepts a value defining how much the robot should turn (positive to the left and negative to the right), the speed of the turn, along with the Drivetrain subsystem.

The command has \_\_init\_\_, initialize, execute, isFinished and end functions similar to other commands.

To determine which direction to turn, we subtract the current heading from the target heading. If this is positive we turn to the right and left if negative.

To determine when we have **completed the turn**, we monitor the difference between the current heading and target heading and when this gets below a specific value (currently set to 5 degrees) we stop the turn.

**<< ACTION >>**

**Update robot.py as follows:**

* Line 7: Import the new turn command
* Lines 32-34: Add button binding code to turn the robot when joystick button 2 is pressed.

**[ Step 6 ] [ drivetrainsubsys.py ]** - Adding Gyro

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66**  **67**  **68**  **69**  **70**  **71**  **72**  **73**  **74**  **75**  **76**  **77**  **78**  **79**  **80**  **81**  **82**  **83**  **84**  **85**  **86**  **87**  **88**  **89**  **90**  **91**  **92**  **93**  **94**  **95**  **96**  **97**  **98**  **99**  **100**  **101**  **102**  **103**  **104**  **105**  **106**  **107**  **108**  **109**  **110**  **111**  **112**  **113**  **114**  **115**  **116**  **117**  **118**  **119**  **120**  **121**  **122**  **123**  **124**  **125**  **126**  **127**  **128**  **129**  **130**  **131**  **132** | **import wpilib**  **from commands2 import Subsystem, Command**  **import wpilib.drive**  **from phoenix6.configs import TalonFXConfiguration**  **from phoenix6.hardware.talon\_fx import TalonFX**  **from phoenix6.signals.spn\_enums import (InvertedValue, NeutralModeValue, FeedbackSensorSourceValue)**  **from phoenix6.controls import DutyCycleOut**  **from phoenix6 import StatusCode**  **import navx**  **import math**  **class DriveTrain(Subsystem):**  **def \_\_init\_\_(self) -> None:**  **super().\_\_init\_\_() # Call the parent's (Super) initialization function**  **# Apply all the configurations to the left and right side Talon motor controllers**  **self.\_\_configure\_left\_side\_drive()**  **self.\_\_configure\_right\_side\_drive()**  **# Initialize Gyro to measure robot heading**  **self.gyro: navx.AHRS = navx.AHRS.create\_spi()**  **print ("DriveTrain Subsystem Initialization complete")**  **def \_\_configure\_left\_side\_drive(self) -> None:**  **self.\_left\_leader\_motor = TalonFX(1) # 1 is the CAN bus address**  **# Applying a new configuration will erase all other config settings since**  **# we start with a blank config so each setting needs to be explicitly set**  **# here in the config method**  **config = TalonFXConfiguration()**  **# Set the left side motors to be counter clockwise positive**  **config.motor\_output.inverted = InvertedValue.COUNTER\_CLOCKWISE\_POSITIVE**  **# Set the motors to electrically stop instead of coast**  **config.motor\_output.neutral\_mode = NeutralModeValue.BRAKE**  **# This configuration item supports counting wheel rotations**  **# This item sets the gear ratio between motor turns and wheel turns**  **config.feedback.feedback\_sensor\_source = FeedbackSensorSourceValue.ROTOR\_SENSOR**  **config.feedback.sensor\_to\_mechanism\_ratio = 10**  **# Apply the configuration to the motors**  **for i in range(6): # Try 5 times**  **ret = self.\_left\_leader\_motor.configurator.apply(config)**  **self.\_left\_leader\_motor.set\_position(0) # Reset the encoder to zero**    **def \_\_configure\_right\_side\_drive(self) -> None:**  **self.\_right\_leader\_motor = TalonFX(2) # 2 is the CAN bus address**  **# Applying a new configuration will erase all other config settings since**  **# we start with a blank config so each setting needs to be explicitly set**  **# here in the config method**    **config = TalonFXConfiguration()**  **# Set the left side motors to be counter clockwise positive**  **config.motor\_output.inverted = InvertedValue.CLOCKWISE\_POSITIVE**  **# Set the motors to electrically stop instead of coast**  **config.motor\_output.neutral\_mode = NeutralModeValue.BRAKE**  **# This configuration item supports counting wheel rotations**  **# This item sets the gear ratio between motor turns and wheel turns**  **config.feedback.feedback\_sensor\_source = FeedbackSensorSourceValue.ROTOR\_SENSOR**  **config.feedback.sensor\_to\_mechanism\_ratio = 10**  **# Apply the configuration to the motors**  **for i in range(6): # Try 5 times**  **ret = self.\_right\_leader\_motor.configurator.apply(config)**  **self.\_left\_leader\_motor.set\_position(0) # Reset the encoder to zero**  **########################### Drivetrain Drive methods #######################**  **def drive\_teleop(self, forward: float, turn: float):**  **# Slow the robot down by the reduction value**  **reduction = 0.5**  **turn = turn \* reduction**  **forward = forward \* reduction**  **speeds = wpilib.drive.DifferentialDrive.curvatureDriveIK(forward, turn, True)**  **# print ("Wheel Speeds: Left:", speeds.left, " Right: ", speeds.right)**  **self.\_left\_percent\_out: DutyCycleOut = DutyCycleOut(0)**  **self.\_right\_percent\_out: DutyCycleOut = DutyCycleOut(0)**  **self.\_left\_percent\_out.output = speeds.left**  **self.\_right\_percent\_out.output = speeds.right**  **self.\_left\_leader\_motor.set\_control(self.\_left\_percent\_out)**  **self.\_right\_leader\_motor.set\_control(self.\_right\_percent\_out)**  **########################### Drive Encoder methods #######################**  **def get\_left\_side\_encoder\_count(self):**  **self.\_left\_leader\_motor.get\_rotor\_position()**  **return self.\_left\_leader\_motor.get\_position().value**  **def get\_right\_side\_encoder\_count(self):**  **self.\_right\_leader\_motor.get\_rotor\_position()**  **return self.\_right\_leader\_motor.get\_position().value**    **def reset\_left\_side\_encoder\_count(self):**  **self.\_left\_leader\_motor.set\_position(0) # Reset the encoder to zero**  **def reset\_right\_side\_encoder\_count(self):**  **self.\_right\_leader\_motor.set\_position(0) # Reset the encoder to zero**  **########################### Drive Gyro methods #######################**  **def reset\_gyro (self):**  **self.gyro.reset()**  **def get\_gyro\_heading(self) -> float: # Turn clockwise = negative Gyro values**  **"""**  **Positive values are Counter clockwise, Negative values are clockwise (after negating)**  **"""**  **angle = math.fmod(-self.gyro.getAngle(), 360)**  **if angle < 0:**  **return angle if angle >= -180 else angle + 360**  **else:**  **return angle if angle <= 180 else angle - 360** |
| --- | --- |

**[ Step 6 ] [ autoturnxdegrees.py]**

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44** | **import wpilib**  **import wpilib.drive**  **from commands2 import Command**  **from drivetrainsubsys import DriveTrain**  **import math**  **class AutoTurnXDegrees(Command):**  **def \_\_init\_\_(self, drivetrain: DriveTrain, targetHeading: int, turnSpeed: float):**  **self.drivetrain = drivetrain**  **self.targetHeading = targetHeading**  **self.turnSpeed = turnSpeed**  **self.differenceBetweenTargetAndCurrent = 0**  **self.addRequirements(self.drivetrain)**  **self.drivetrain.reset\_gyro()**  **self.current\_heading = self.drivetrain.get\_gyro\_heading()**  **print ("Heading Change of ", self.targetHeading, " Command Initialized")**  **def initialize(self):**  **super().initialize()**  **self.drivetrain.reset\_gyro()**  **self.current\_heading = self.drivetrain.get\_gyro\_heading()**  **def execute(self):**  **self.currentHeading = self.drivetrain.get\_gyro\_heading()**  **self.differenceBetweenTargetAndCurrent = self.targetHeading - self.currentHeading**  **print ("Gyro Heading (Clockwise turn is negative: ", self.currentHeading,**  **" Target heading: ", self.targetHeading,**  **" Difference: ", self.differenceBetweenTargetAndCurrent )**  **if self.differenceBetweenTargetAndCurrent > 0:**  **# Would like to turn Clockwise**  **self.drivetrain.drive\_teleop(0.0, self.turnSpeed)**  **else:**  **self.drivetrain.drive\_teleop(0.0, -self.turnSpeed)**  **# Positive turn value moves robot counter clockwise**  **def isFinished(self) -> bool:**  **if (math.fabs(self.differenceBetweenTargetAndCurrent) < 5):**  **return True**  **else:**  **return False**  **def end(self, interrupted: bool):**  **self.drivetrain.drive\_teleop(0,0)** |
| --- | --- |

**[ Step 6 ] [ robot.py ]** - adding group command

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66**  **67** | **import wpilib**  **from drivetrainsubsys import DriveTrain**  **from teleopdrivecmd import TeleopDrive**  **from commands2 import Command, RunCommand, TimedCommandRobot, button**  **from autodrivexseconds import AutoDriveXSeconds**  **from autodrivexwheelcounts import AutoDriveXWheelCounts**  **from autoturnxdegrees import AutoTurnXDegrees**  **class MyRobot(TimedCommandRobot):**  **def robotInit(self):**  **"""**  **This function is called upon program startup and**  **should be used for any initialization code.**  **"""**  **# Instantiate (create) subsystems**  **self.drivetrainSubSys: DriveTrain = DriveTrain()**  **self.controller = wpilib.Joystick(0)**  **self.\_\_configure\_button\_bindings()**  **# Configure commands**  **self.configure\_default\_commands()**  **print ("Robot Initialization (robotInit) Completed ")**  **def \_\_configure\_button\_bindings(self) -> None:**  **button.JoystickButton(self.controller,1).onTrue(**  **AutoDriveXSeconds(self.drivetrainSubSys, 1, 0.3)**  **)**  **button.JoystickButton(self.controller,2).onTrue(**  **AutoTurnXDegrees(self.drivetrainSubSys, 90, 0.3)**  **)**  **def configure\_default\_commands(self) -> None:**  **self.drivetrainSubSys.setDefaultCommand(**  **TeleopDrive(self.drivetrainSubSys, self.controller)**  **)**  **def getAutonomousCommand(self) -> Command:**  **# Configure command to drive when autonomous is selected (1 second at 50% speed)**  **return AutoDriveXWheelCounts(self.drivetrainSubSys, 1, 0.5)**  **def autonomousInit(self):**  **"""This function is run once each time the robot enters autonomous mode."""**  **self.\_auto\_command = self.getAutonomousCommand()**  **if self.\_auto\_command is not None:**  **self.\_auto\_command.schedule()**  **print ("Autonomous Initialization (autonomousInit) Completed ")**  **def autonomousPeriodic(self):**  **"""This function is called periodically during autonomous."""**  **def teleopInit(self):**  **"""This function is called once each time the robot enters teleoperated mode."""**  **print ("TeleOpInit Initialization (teleopInit) Completed ")**  **def teleopPeriodic(self):**  **"""This function is called periodically during teleoperated mode."""**  **# Xaxis = self.controller.getRawAxis(0)**  **# Yaxis = self.controller.getRawAxis(1)**  **# print("Joystick: Forward Motion Axis: ", Yaxis, " Turning Motion Axis: ", Xaxis )** |
| --- | --- |

**Creating a Group of Commands**

**[ Step 7 ]**

We will now chain together a group of commands that the robot will run sequentially in autonomous mode. Our goal is to make a robot drive in a square pattern pausing between each action. We will call a single command and the robot will drive straight, pause, turn, pause four times.

To create the command group, we will:

* Add a command that causes the robot to pause (wait)
* Create a GroupCommand
* Update the Robot.py to call the group command when we switch to anonymous mode

**<< ACTION >>**

**Create a Wait command:**

* Create a file called “waitxseconds”
* Copy in the code listed below into the file. This command is very similar to the AutoDriveXSeconds command but sets the motor speed to zero in each loop.

**Create a GroupCommand command:**

* Create a file called “autocommandgroup.py”
* Copy in the code listed below into the file

The AutonomousCommandGroup command allows you to chain together a number of commands using the "addCommands" function. This single command can be updated easily.

**Updating robot.py to call the GroupCommand:**

* Line 8, Import the **commandgroup** command into the robot.py file.
* Lines 44-45: Update the getAutoCommand to return the AutonomousCommandGroup command

**[ Step 7 ] [ waitxseconds.py ]** - Wait command

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29** | **import wpilib**  **import wpilib.drive**  **from commands2 import Command**  **from drivetrainsubsys import DriveTrain**  **class WaitXSeconds(Command):**  **def \_\_init\_\_(self, drivetrain: DriveTrain, seconds: int):**  **self.drivetrain = drivetrain**  **self.seconds = seconds**  **self.counter = 50 \* seconds**  **self.addRequirements(self.drivetrain)**  **print ("Waiting for ", seconds, " Command Initialized")**  **def initialize(self):**  **super().initialize()**  **def execute(self):**  **self.drivetrain.drive\_teleop(0.0, 0.0)**  **self.counter = self.counter - 1**  **def isFinished(self) -> bool:**  **if self.counter <= 0:**  **return True**  **else:**  **return False**    **def end(self, interrupted: bool):**  **self.drivetrain.drive\_teleop(0.0, 0.0)** |
| --- | --- |

**[ Step 7 ] [ autocommandgroup.py ]**

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29** | **import commands2**  **from commands2 import PrintCommand**  **from autodrivexseconds import AutoDriveXSeconds**  **from autoturnxdegrees import AutoTurnXDegrees**  **from waitxseconds import WaitXSeconds**  **from drivetrainsubsys import DriveTrain**  **class AutonomousCommandGroup(commands2.SequentialCommandGroup):**  **def \_\_init\_\_(self, drivetrainsubsys: DriveTrain) -> None:**  **super().\_\_init\_\_()**  **print ("Within Autonomous SequentialCommand Group")**  **self.drivetrainsubsys = drivetrainsubsys**  **self.addCommands(AutoDriveXSeconds(self.drivetrainsubsys, 3, 0.3))**  **self.addCommands(WaitXSeconds(self.drivetrainsubsys,1))**  **self.addCommands(AutoTurnXDegrees(self.drivetrainsubsys,-90, 0.3))**  **self.addCommands(AutoDriveXSeconds(self.drivetrainsubsys, 3, 0.3))**  **self.addCommands(WaitXSeconds(self.drivetrainsubsys,1))**  **self.addCommands(AutoTurnXDegrees(self.drivetrainsubsys,-90, 0.3))**  **self.addCommands(AutoDriveXSeconds(self.drivetrainsubsys, 3, 0.3))**  **self.addCommands(WaitXSeconds(self.drivetrainsubsys,1))**  **self.addCommands(AutoTurnXDegrees(self.drivetrainsubsys,-90, 0.3))**  **self.addCommands(AutoDriveXSeconds(self.drivetrainsubsys, 3, 0.3))**  **self.addCommands(WaitXSeconds(self.drivetrainsubsys,1))**  **self.addCommands(AutoTurnXDegrees(self.drivetrainsubsys,-90, 0.3))**  **self.addCommands(PrintCommand("Done"))** |
| --- | --- |

**[ Step 7 ] [ robot.py ]** - adding group command

| **01**  **02**  **03**  **04**  **05**  **06**  **07**  **08**  **09**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66**  **67**  **68**  **69**  **70** | **import wpilib**  **from drivetrainsubsys import DriveTrain**  **from teleopdrivecmd import TeleopDrive**  **from commands2 import Command, RunCommand, TimedCommandRobot, button**  **from autodrivexseconds import AutoDriveXSeconds**  **from autodrivexwheelcounts import AutoDriveXWheelCounts**  **from autoturnxdegrees import AutoTurnXDegrees**  **from autocommandgroup import AutonomousCommandGroup**  **from typing import Tuple, List**  **class MyRobot(TimedCommandRobot):**  **def robotInit(self):**  **"""**  **This function is called upon program startup and**  **should be used for any initialization code.**  **"""**  **# Instantiate (create) subsystems**  **self.drivetrainSubSys: DriveTrain = DriveTrain()**  **self.controller = wpilib.Joystick(0)**  **self.\_\_configure\_button\_bindings()**  **# Configure commands**  **self.configure\_default\_commands()**  **print ("Robot Initialization (robotInit) Completed ")**  **def \_\_configure\_button\_bindings(self) -> None:**  **button.JoystickButton(self.controller,1).onTrue(**  **AutoDriveXWheelCounts(self.drivetrainSubSys, 1, 0.3)**  **)**  **button.JoystickButton(self.controller,2).onTrue(**  **AutoTurnXDegrees(self.drivetrainSubSys, 30, 0.3)**  **)**  **def configure\_default\_commands(self) -> None:**  **self.drivetrainSubSys.setDefaultCommand(**  **TeleopDrive(self.drivetrainSubSys, self.controller)**  **)**  **def getAutonomousCommand(self) -> Command:**  **return AutonomousCommandGroup(self.drivetrainSubSys)**  **def autonomousInit(self):**  **"""This function is run once each time the robot enters autonomous mode."""**  **self.\_auto\_command = self.getAutonomousCommand()**  **if self.\_auto\_command is not None:**  **self.\_auto\_command.schedule()**  **print ("Autonomous Initialization (autonomousInit) Completed ")**  **def autonomousPeriodic(self):**  **"""This function is called periodically during autonomous."""**  **def teleopInit(self):**  **"""This function is called once each time the robot enters teleoperated mode."""**  **print ("TeleOpInit Initialization (teleopInit) Completed ")**  **def teleopPeriodic(self):**  **"""This function is called periodically during teleoperated mode."""**  **# Xaxis = self.controller.getRawAxis(0)**  **# Yaxis = self.controller.getRawAxis(1)**  **# print("Joystick: Forward Motion Axis: ", Yaxis, " Turning Motion Axis: ", Xaxis )** |
| --- | --- |

**Adding a Distance RangeFinder to the DriveTrain Subsystem**

**[ Step 8 ]**

The final update to the code is to add an ultrasonic RangeFinder to the robot code. This sensor can measure the distance to a wall (or other large object) in front of the robot.

This update will be left to the student with the following guidance. Perform the following steps to make the robot stop a specific distance from a wall:

* Add the RangeFinder to the drivetrain
* Create a command which we pass in the desired stopping distance from the wall and the speed to which the robot will approach the wall.
* Assign a button to trigger the command or make it part of a command group.

To add the rangefinder to the drivetrain, we need to instantiate the sensor within the \_\_init\_\_ function and create a “GET” function to ask for the distance to the wall. Here is the core code for these two actions.

**self.ultraSoundRangeFinder = wpilib.AnalogInput(0)**

**def get\_rangefinder\_distance\_in\_inches(self) -> float:**

**rawReturnedValue = self.ultraSoundRangeFinder.getAverageVoltage()**

The rangefinder distance is provided as a voltage feed into one of the RoboRio’s analog inputs. Here are typical voltage values measured at specific distances. Create a math function to convert from voltages to distance.

Range finder data:

1 ft = 0.6 Volts - Close distance measurable by the rangefinder

2 ft = 1.19 Volts

3 ft = 1.68 Volts

4 ft = 2.22 Volts

5 ft = 2.9 Volts

6 ft = 5.3 Volts

**Going Further:**

Programming robots is great fun. The concepts conveyed in this document are just the beginning of a new skill set. There are better approaches to controlling the robot than provided in the previous pages such as PID loops (Proportional-Integral-Derivative) and smart camera vision control. Learn the basics then learn advanced skills. Learning takes time and patience.