FIRST ROBOTICS COMPETITION

REEFSCAPE℠ PRESENTED BY HAAS Gene Haas Foundation

FIRST® DIVE℠
presented by Qualcomm

firstinspires.org/robotics/frc

2025 *FIRST*® Robotics Competition

KitBot Python Software Guide

1 Contents

2 Document Overview

This document will take you through how to get your 2025 KitBot up and running using the provided Python example code. To avoid content duplication this document frequently links to WPILib and/or RobotPy documentation for accomplishing specific steps along the way. In addition to getting you up and running with the provided code, this document will walk through the structure of that code so you can understand how it operates. Finally, we'll walk through some of the most likely changes you may wish to make to the code and provide concrete examples of how to make those modifications.

To get started with the example code, or to make some of the modifications described, minimal understanding of Python is required. The code and modification examples provided will likely provide enough of a pattern to get you going. To understand the walkthrough, or to make modifications not described in this document, a more thorough understanding of Python is likely required. The Introduction to Python in the RobotPy documentation will get you started, and also has some links to more comprehensive resources.

This document, and the provided example code, assumes the use of the SPARK MAX controllers provided in the rookie Kickoff Kit.

3 Getting Started with your KitBot code

3.1 Wiring your robot

Use the WPILib Zero-to-Robot wiring document to help you get your robot wired up. The KitBot wiring and code is documented with the Control System components that have recently come in the Rookie Kit of Parts (i.e. REV PDH and Spark MAX controllers), the KitBot wiring and code can be adapted to other electronics but that adaptation is not covered by these documents.

3.2 Configuring hardware and development environment

Before you are able to load code and test out your robot, you will need to configure your hardware (roboRIO, radio, etc.) and get your development environment set up. Follow the WPILib Zero-to-Robot guide steps 2 through 4 and the RobotPy installation steps (make sure to install the REV package as well) to get everything set up and ensure you can deploy a basic robot project.

3.3 Opening the 2025 KitBot Example

The 2025 KitBot example code is provided in individual zip files for each language on the KitBot webpage. The Python code contains two complete projects which illustrate different ways of creating Commands. Some description of the difference can be found in **Error! Reference source not found.** below. To open the C++ code:

1. Download and unzip the Python example code. Make sure to unzip or copy to a permanent location, not in a temporary folder.
2. Open **WPILib VS Code** using the Start menu or desktop shortcuts
3. In the top left click **File->Open Folder** and browse to the "C++" folder inside of the unzipped example code, then open the desired one of the two example projects, and then click **Select Folder.**

3.4 Changing to CAN control

If you have wired your SPARK MAX motor controllers using CAN, you will need to do some further configuration and code modification before proceeding. If you are using PWM, skip down to Section 3.5 to deploy and test the code.

3.5 SPARK MAX firmware update and CAN IDs

Before using the SPARK MAXs with CAN control, they each need to be assigned a unique ID..

1. Install the REV Hardware Client
2. With the robot powered off, connect a USB cable between the computer and the SPARK MAX USB port. Leaving the robot powered off ensures only the single SPARK MAX is powered and avoids changing the IDs on unintended devices.
3. Update the firmware on the SPARK MAX

4. Set the CAN ID and Motor Type (you can skip the current limit) and save the settings
    a. CAN IDs for each device can be found in Constants.java. You can either set the devices to match these IDs, or set the IDs as desired (some teams set the CAN ID = the channel number the device is attached to on the PD) and then update these constants.
    b. Note: If you wish to "Spin the motor" as described on that page, make sure the robot is in a safe state to do so (wheels not touching the ground or table, all hands clear of the robot).
5. Repeat for all 5 devices on the robot. IF you've wired up the 6th Spark MAX you likely want to set it to a non-conflicting ID as well.
6. While not required, if using the REV PDH you may wish to check that it has the latest firmware at this time as well. Do not change the ID of the PDH off of the default, each device type has a separate ID space and your PDH will not conflict with your SPARK MAX even if set to the same ID.

Now that all your devices are configured, you can do a preliminary check that your CAN bus is wired properly using the REV Hardware client. While plugged into any REV device on your CAN bus with a USB cable, power on the robot and you should see all the other devices listed in the left pane of the REV Hardware Client, under the CAN Bus heading. If you don't see all of the devices, you likely have one or more issues with your CAN bus wiring:

1. Verify that your CAN bus starts with the roboRIO and ends with a 120 ohm resistor, or the built in terminator of a Power Distribution Hub or Power Distribution Panel (with the termination set to On using the appropriate jumper or switch).
2. Check that your CAN bus connections all match yellow-yellow and green-green.
3. Check that all CAN wire connections are secure to each other and that the connectors are securely installed in each SPARK Max
4. If you're still having trouble, moving the USB connection around to different devices and seeing what each device can "see" on the bus can help pinpoint the location of an issue.

3.6 Deploying and testing the KitBot Example

To deploy the example to your robot, you will need to follow the deploy instructions in the RobotPy documentation.

Note: As of the latest release of RobotPy and REVLib, tests would frequently hang and not complete. You likely need to add '—skiptests' to the deploy command in order to skip tests and allow deploy to succeed.

Warning: Make sure you have space in all directions when operating a robot. Even with known code, the robot may move with unexpected speed or in unexpected directions. Be prepared to Disable (Enter) or E-stop (Spacebar) the robot if necessary. The 2025 KitBot code contains a very simple autonomous routine that will move the robot forwards at ½ speed for 1 second when the robot is enabled in Autonomous mode.

3.7 Configuring Gamepads

The code is set up to use the Xbox controller class. The Logitech F310 gamepads provided in the Kit of Parts will appear like Xbox controllers to the WPILib software if they are configured in the correct mode. To set up the controllers, check that the switch on the back of the controller is set the 'X' setting. Then when using the controller, make sure the LED next to the Mode button is off, if it is on press the Mode button to toggle it. When the Mode button is on, the controller swaps the function of the left Analog stick and the D-pad.

3.8 What does the code do?

The provided code implements the following robot controls in Teleoperated:

- Driver controller is an Xbox Controller in Slot 0 of the Driver Station
    - Controls the robot drivetrain using Split-stick Arcade Drive
        - Y-axis (vertical) of left stick controls forward-back movement of drivetrain
        - X-axis (horizontal) of right stick controls rotation of drivetrain
- Operator controller is an Xbox controller in Slot 1 of the Driver Station
    - Controls the gamepiece roller using the triggers and buttons
        - Left Trigger – Runs the roller motor inward (tries to push the Coral back up the ramp) at variable speed.
        - Right Trigger – Runs the roller motor outward (tries to eject the Coral) at variable speed.
        - A-Button – Runs the roller motor outward at specific power.

Because of the way the code is implemented, it is not recommended to press the triggers at the same time as the behavior will be difficult to predict. Pressing the left and right triggers at the same time will add their values together. For example, pressing the left trigger down halfway, and the right trigger down all the way, the result will be the motor spinning outward at ½ speed.

4 Overall Code Structure

The provided code utilizes the Command-Based programming structure provided by WPILib. This structure breaks up the robot's actuators into "subsystems" which are controlled by "commands" or collections of commands (aptly name "command groups"). The Command-Based structure may be a bit overkill for a robot of this complexity, but it scales very well for teams looking to add additional functionality to their KitBot. Additionally, this code structure was used by over 60% of teams in 2024, increasing the likelihood that teams around you may be able to provide assistance before or during the event.

To read more about the Command-Based structure, see the Command-Based Programming chapter of the WPILib documentation.

4.1 Ways of creating commands

There are multiple ways that you can define commands within the Command-Based structure. This project uses two of these different types to provide exposure to what they would look like in a full robot project. The common ways of creating commands that are utilized in this project are:

- Traditional: Command/group is defined as its own class in its own file.
- Factory: Command/group is defined via a "Command Factory method" in the subsystem or a separate static command class.

Traditional	Factory
Generally easier to understand	**Slightly high learning curve**
Modularity – Commands can be long depending on the complexity	**Modularity – Commands are broken down into small pieces and strung together into groups**
Boilerplate – Command classes require subclassing and can be long	**Boilerplate – Commands are written as methods in the subsystem, meaning less unnecessary code**
Organization – Having many Command classes can cause clutter and slow programmer's efficiency	**Organization – Commands are grouped together based on the subsystems they require, leading to fewer/none dedicated Command classes**
Debugging – Easier to debug due to more defined structure and traditional logic	**Debugging – More difficult to debug due to the lambda functions and command compositions**

5 Code Walkthrough

5.1 Subsystems

As described in the What is Command-Based Programming article, "'subsystems' represent independently-controlled collections of robot hardware (such as motor controllers, sensors, pneumatic actuators, etc.) that operate together".

For the 2025 KitBot we have 5 motors that make up 2 subsystems, the Drivetrain, and the Roller. The 4 motors in the drivetrain always need to be working together to move the robot around the field and the Roller motor must spin to manipulate Coral. Sometimes the boundaries between subsystems may not be so clear, if you have an arm with a shoulder and wrist joint and a set of motorized wheels on the end, is that all one subsystem or multiple? The general rule of thumb to follow is think about what actions, or commands, you might have to control the subsystems. Do you think you might want the two pieces to be controlled independent of each other (i.e. run the intake in or out while moving the arm or wrist?). If you're unsure, err towards more smaller subsystems; you can always make commands that require multiple subsystems but if you end up wanting separate commands to control a single subsystem at the same time, you'll have to refactor the subsystem to split it up.

5.1.1 CANDriveSubsystem

5.1.1.1 Imports

```
5   import commands2
6   import wpilib
7   import wpilib.drive
8   import rev
9
10  import Constants
```

This section declares what other modules we need to reference within this code (imports). A common practice is to add imports as you go; as you find yourself referencing a module you have not yet imported, you can add an import for that class.

In this subsystem we import the commands2 library, the WPILib library, and specifically the drive module in WPILib. An additional import is needed for the REV Robotics third party library. This module provides the functionality for controlling SPARK Maxs over CAN. You can find information about the objects and methods available from the "rev" module in the RobotPy documentation. The last thing we import is the "constants" module from this project where we declare things like port numbers and timings specific to the KitBot.

5.1.1.2 Class declaration and Constructor

```
14  class CANDriveSubsystem(commands2.Subsystem):
15      def __init__(self) -> None:
16          super().__init__()
17
18          # spark max motor controllers in brushed mode
19          self.leftLeader = rev.SparkMax(
20              Constants.LEFT_LEADER_ID, rev.SparkBase.MotorType.kBrushed
21          )
22          self.leftFollower = rev.SparkMax(
23              Constants.LEFT_FOLLOWER_ID, rev.SparkBase.MotorType.kBrushed
24          )
25          self.rightLeader = rev.SparkMax(
26              Constants.RIGHT_LEADER_ID, rev.SparkBase.MotorType.kBrushed
27          )
28          self.rightFollower = rev.SparkMax(
29              Constants.RIGHT_FOLLOWER_ID, rev.SparkBase.MotorType.kBrushed
30          )
31
32          # this is the differential drive instance which allows us to control
33          # the drive with joysticks
34          self.drive = wpilib.drive.DifferentialDrive(
35              self.leftLeader, self.rightLeader
36          )
37
38          # set can timeouts. This program only sets parameters on startup and
39          # doesn't get any parameters so a long timeout is acceptable. Programs
40          # which set or get parameters during runtime likely want a timeout
41          # closer or equal to the default.
42          self.leftLeader.setCANTimeout(250)
43          self.rightLeader.setCANTimeout(250)
```

The first line of this image is the class declaration. This declares the name of our class and says that it's an extension of the Subsystem class. All subsystems should extend this class which provides some utility functions regarding setting the name of the subsystem, registering it with the scheduler, and sending information about it to the dashboard.

The next section is the constructor. Here we initialize any variables contained in the subsystem. In the case of our drivetrain, we initialize the 4 motor controllers, set configuration for them, and collect the leaders for each side into a WPILib DifferentialDrive object which describes the whole drivetrain.

5.1.1.3 Methods

```
83      # function to drive with joystick inputs
84      def arcadeDrive(self, xSpeed: float, zRotation: float) -> None:
85          self.drive.arcadeDrive(xSpeed, zRotation)
```

The remainder of the subsystem class is methods. Here we define any methods that commands may need to call to get status from or perform actions on the subsystem. For our simple drivetrain, the only method we need is the arcadeDrive method which simply passes the parameters through to the same

method on the DifferentialDrive object. In the "Inline Commands" project, this method will be a little more complicated, instead returning a command which runs the drivetrain.

5.1.2 CANRollerSubsystem

This class is the subsystem for the roller.

5.1.2.1 Package, Imports, Class Declaration, Member Variables, and Constructor

The first sections of this subsystem are very similar to the PWM Drive subsystem. See that section for more detailed description of each of these parts of the code. For the roller, a single motor is created and controlled independently, no motor controller group or drivetrain object is used.

5.1.2.2 Methods – Hardware Control

```
17      def runRoller(self, forward: float, reverse: float) -> None:
18          self.rollerMotor.set(forward - reverse)
```

The remainder of the subsystem class is hardware access methods. Here we define any methods that commands may need to call to get status from or perform actions on the subsystem. For our roller this includes methods to set the speed of the axle. Again, the Inline Commands project will have a more complicated line here which creates a command.

5.2 Commands

Commands are what tell the robot when to run the different components that are defined in their subsystems. Commands are "scheduled" for execution, performed, and then removed from the command scheduler. Each command has 4 distinct parts that are performed throughout its lifecycle:

- Initialize: performed when the command is initially scheduled. Anything put in the initialize section of a command will run right before the main body of the command.
- Execute: the main body of the command, which runs once every loop cycle (~20 ms)
- End: performed when the command is removed from the scheduler.
- isFinished: called once per loop cycle after the execute method. isFinished returns true when the condition for exiting the command is met. When isFinished returns true, the end method is called.

As we discussed in section 4, The two ways of writing commands that we focus on in this tutorial are Traditional commands which subclass the WPILib Command class, and command factories, which use command decorators to create commands. Regardless of the way your team chooses to make commands, this is the underlying structure that commands follow.

5.3 Traditional Commands

This subsection will focus on the Traditional command-based framework. Traditional commands are defined in their own classes by subclassing the generic WPILib Command. This means that we can directly override the behavior of the command methods.

5.3.1 DriveCommand

5.3.1.1 Imports and Constructors

```
import commands2

from subsystems.CANDriveSubsystem import CANDriveSubsystem


# Command class to drive the robot based on joystick inputs
class DriveCommand(commands2.Command):
    def __init__(
        self,
        xSpeed: lambda xSpeed: xSpeed,
        zRotation: lambda zRotation: zRotation,
        driveSubsystem: CANDriveSubsystem,
    ) -> None:
        self.driveSubsystem = driveSubsystem
        self.xSpeed = xSpeed
        self.zRotation = zRotation
        self.addRequirements(self.driveSubsystem)
        super().__init__()
```

The imports for this command include the commands2 library and the subsystem used. The class definition indicates that this command extends the commands2.Command class by including it in parentheses after the class name. The init method specifies the parameters our command takes, using lambdas so the values can change. It then stores each of these parameters into local variables for later access and specifies the subsystem(s) required by the command. A command should specify a requirement for each subsystem it controls the output of. If a command is going to be the default command for a subsystem, it should require that subsystem.

5.3.1.2 Methods – Command State Overrides

```
    def initialize(self) -> None:
        pass

    def execute(self) -> None:
        self.driveSubsystem.arcadeDrive(self.xSpeed(), self.zRotation())

    def end(self, interrupted: bool) -> None:
        pass

    def isFinished(self) -> bool:
        return False
```

In the execute method, we call the ArcadeDrive method from the drive subsystem which tells the motors to drive such that the robot moves according to the joystick inputs.

5.3.2 RollerCommand

The first section of this command is very similar to the Drive command. See section **Error! Reference source not found.**1 for more detailed description of each of the parts of the code.

5.3.2.1 Methods – Command State Overrides

```
36      def initialize(self) -> None:
37          pass
38
39      def execute(self) -> None:
40          self.rollerSubsystem.runRoller(self.forward(), self.reverse())
41
42      def end(self, interrupted: bool) -> None:
43          pass
44
45      def isFinished(self) -> bool:
46          return False
47
```

In the execute method, we call the runRoller method from the Roller subsystem, passing through the values that are given to the command.

5.3.3 AutoCommand

The AutoCommand is slightly different from the other two commands, as it requires a little bit more setup.

5.3.3.1 Imports and Constructor

```
10  # Command class to run the robot forwards at 1/2 power for 1 second in autonomous.
11  class AutoCommand(commands2.Command):
12      # Constructor for CAN command
13      def __init__(self, driveSubsystem: CANDriveSubsystem) -> None:
14          self.driveSubsystem = driveSubsystem
15          self.timer = wpilib.Timer()
16          self.seconds = 1.0
17          super().__init__()
```

The AutoCommand constructor takes a drive subsystem parameter, and instantiates two member variables to track the state of the auto routine. The number of seconds are hardcoded here but could be changed into a parameter if you wanted to make the command more flexible.

5.3.3.2 Methods – Command State Overrides

```python
# called when command is initially scheduled
def initialize(self) -> None:
    self.timer.restart()

# called every loop cycle (~20 ms) while command is running
def execute(self) -> None:
    self.driveSubsystem.arcadeDrive(0.5, 0.0)

# called after every execution to check if command is finished
def isFinished(self) -> bool:
    return self.timer.get() >= self.seconds

# called when command ends
def end(self, interrupted: bool) -> None:
    self.driveSubsystem.arcadeDrive(0.0, 0.0)
```

- Initialize: Start the timer. Uses "restart" to allow the command to run more than once without restarting the robot.
- Execute: Sets the drive motors to drive in reverse.
- End: Stop the timer and stop the drive subsystem.
- IsFinished: Returns true when the timer exceeds 1 second.

5.4 Factory Commands

This subsection will focus on the factory command-based framework. Commands are defined in their subsystems using WPILib inline commands and decorators.

5.4.1 Drive Command Factory

The drive command factory is a method in the Drive subsystem which returns a WPILib Command. We call in the RobotContainer to set it as the default command for the drive subsystem.

```python
    # method to drive the robot with joysticks using the differential drive method 'arcadeDrive'
    def arcadeDrive(
        self,
        driveSubsystem,
        xSpeed: lambda xSpeed: xSpeed,
        zRotation: lambda zRotation: zRotation,
    ) -> commands2.Command:
        return commands2.cmd.run(
            lambda: self.drive.arcadeDrive(xSpeed(), zRotation()),
            driveSubsystem,
        )
```

To create the command, the run method from commands2.cmd is used. This method creates a command that runs the supplied method during the execute() phase of the command (every ~20ms while the command is scheduled).

5.4.2 Roller Command Factory

This section is very similar to the Drive Command Factory section. If you want more details on how the command-based factories work, see Section 5.4.1.

```
    def runRoller(
        self,
        rollerSubsystem,
        forward: lambda forward: forward,
        reverse: lambda reverse: reverse,
    ) -> commands2.Command:
        return commands2.cmd.run(
            lambda: self.rollerMotor.set(forward() - reverse()),
            rollerSubsystem,
        )

```

5.4.3 Autos

The Autos file is an example of a "Static Command Factory". Your program should never create an Autos object (as shown by the constructor simply printing an error message) instead you call class methods statically using Autos.ExampleAuto() type syntax. This structure is one of the ways to define complex groups that involve multiple subsystems (though our example here is not complex and requires only a single subsystem).

```
class Autos(commands2.Command):
    def __init__(self) -> None:
        pass

    def exampleAuto(driveSubsystem: CANDriveSubsystem) -> commands2.Command:
        return commands2.cmd.run(
            lambda: driveSubsystem.arcadeDrive(0.5, 0.0)
        ).withTimeout(1.0)
```

Our example file only has a single autonomous routine to get. You could easily extend this pattern by adding additional methods to define more autonomous routines and you could select between them using a SendableChooser on the dashboard.

This simple autonomous routine instructs the robot to drive forwards for 1 second at 50% power by using the WithTimeout decorator to set a timeout of 1 second on the driving command. The different types of command compositions that are built-in via decorators and factory methods are described on the Command Compositions page.

5.5 Constants

This class contains named constants used elsewhere in the code. Sections are used to organize the constants into distinct groups delineated by comments.

5.6 Robot

This file is identical to the default Command-Based template. You can find a description of the elements in the Robot class in the Structuring a Command-Based Robot Project article. While that article provides examples in Java, that should be pretty easily mapped to the Python equivalents.

5.7 RobotContainer

The RobotContainer class is where instances of the robot subsystems and controllers are declared and where default commands and mappings of buttons to commands are defined.

5.7.1 Imports

```
5   import commands2
6   import commands2.button
7
8   import Constants
9   from commands.Autos import Autos
10  from subsystems.CANDriveSubsystem import CANDriveSubsystem
11  from subsystems.CANRollerSubsystem import CANRollerSubsystem
```

The first section of code is the imports. In this case we need to import some elements from the commands module, the constants file from our project, and then all of our commands and subsystems.

5.7.2 Class definition and Constructor

```
class RobotContainer:
    """
    This class is where the bulk of the robot should be declared. Since Command-based is a
    "declarative" paradigm, very little robot logic should actually be handled in the :class:`.Robot`
    periodic methods (other than the scheduler calls). Instead, the structure of the robot (including
    subsystems, commands, and button mappings) should be declared here.
    """

    def __init__(self) -> None:
        # The driver's controller
        self.driverController = commands2.button.CommandXboxController(
            constants.kDriverControllerPort
        )
        self.operatorController = commands2.button.CommandXboxController(
            constants.kOperatorControllerPort
        )

        # The robot's subsystems
        self.drive = DriveSubsystem()
        self.launcher = LauncherSubsystem()
```

The first section of the constructor sets up some member variables in the class. For RobotContainer this generally includes all of your subsystems and control devices. This code uses the CommandXboxController class to represent the gamepads as it contains a number of helper methods that make it much easier to connect commands to buttons. Next is a call to configureBindings() which we will cover below. This method is used to set up button bindings and default commands.

```
    # Function to bind commands to buttons on the driver and operator controllers.
    def configureButtonBindings(self):
        self.driveSubsystem.setDefaultCommand(
            self.driveSubsystem.arcadeDrive(
                self.driveSubsystem,
                lambda: -self.driverController.getLeftY(),
                lambda: -self.driverController.getRightX(),
            )
        )
```

The configureBindings method is used to set up when each command should run. Splitting this into a separate method is done purely for organization purposes. You could do this all in the constructor, or go the other way and split up the default command setting into a different method from the button mapping.

The first thing in configureBindings is the default command for the drivetrain. We want a command to run on our drivetrain to allow us to drive the robot with joysticks whenever we don't have some other command using the drivetrain (like the exampleAuto command). To do this we use the setDefaultCommand() method of the subsystem. This sets the command that will run whenever the Scheduler sees nothing else running on that subsystem. The Traditional Commands version of this will look slightly different as it's using the separate command class instead of the method in the subsystem that returns a command.

For the forward/back movement we pass in the value from the Y-axis (vertical) of the left stick of the controller, but we negate it. This is because joysticks generally define pushing the stick away from you as negative and pulling the stick towards you as positive (a result of the original use being flight simulators). We want pushing the stick away from us to drive the robot forward so we negate the value. Similar for the turning value where we negate the X-axis (horizontal) of the right stick of the controller. The joystick considers pushing this to the right as a positive value, but the WPILib classes consider clockwise rotation (what would be expected when pushing the joystick right) as negative.

```
self.rollerSubsystem.setDefaultCommand(
    RollerCommand(
        lambda: self.driverController.getLeftTriggerAxis(),
        lambda: self.driverController.getRightTriggerAxis(),
        self.rollerSubsystem,
    )
)
```

This is the default command binding for the roller subsystem using Traditional Commands, the Inline Commands one will looks slightly different.

```
54          self.operatorController.a().whileTrue(
55              self.rollerSubsystem.runRoller(
56                  self.rollerSubsystem,
57                  lambda: Constants.ROLLER_MOTOR_EJECT_SPEED,
58                  lambda: 0,
59              )
60          )
```

Finally, we bind a command to the A button of the operator controller. The CommandXboxController class contains methods for each button which return “Trigger” objects. These “Trigger” objects then have further methods that narrow down the behavior we want to control the command such as toggles, initiating on change, or running only while the Trigger is true or false. In this instance we use “whileTrue()” to have our command run while the button is being held and stop when it is released. Because our RunRoller Command takes lambdas as parameters so they can change when using the joysticks, we need to use lambdas here as well even though the values are constant.

6 Making Changes

This section details some common possible changes you may want to make to the KitBot code and provides some references for how to approach making those changes.

6.1 Changing Drive Axis Behavior

Another easy change to make is to modify which axes of the controller are used as which part of the robot driving and how. The provided code does this mapping when setting up the drivetrain default command at the end of the constructor in RobotContainer.

The example code uses the Y-axis of the left stick to drive forward and back and the X-axis of the right stick to rotate. These can easily be swapped to the opposite sticks, or move just one so they are on the same stick! To review the available options, look for methods that return a **float** in the XboxController API Doc. To make this type of modification, locate the method call you wish to change, such as getLeftY(), and replace it with the new desired method, such as getRightY()

Example: changing the forward-back driving to the right stick Y-axis and leaving the rotation on the right X-axis

```
self.driveSubsystem.setDefaultCommand(
    self.driveSubsystem.arcadeDrive(
        self.driveSubsystem,
        lambda: self.driverController.getRightY(),
        lambda: self.driverController.getRightX(),
    )
)
```

You can also modify the axis values. One common modification is to cube the values. This preserves the sign of the value (positive stays postive, negative stays negative) and the maximum value (doesn't reduce the max speed of the robot) while providing less sensitivity at low inputs, potentially allowing for more precise control at low speeds. To make this type of modification, you can apply the modification to the axis where it's being captured.

Example: changing only the rotation axis to be cubed:

```
self.driveSubsystem.setDefaultCommand(
    self.driveSubsystem.arcadeDrive(
        self.driveSubsystem,
        lambda: self.driverController.getLeftY(),
        lambda: self.driverController.getRightX() ** 3,
    )
)
```

Another common modification is to scale the values down by default, but allow for the maximum value if a button is pressed (turbo mode). This type of modification can also be done at the point of capture, though as complexity grows, you may wish to shift from an inline command definition to a different type where you can define the command behavior more clearly.

Example: Scale the forward-back driving by 50% unless the right bumper is pressed

```
self.driveSubsystem.setDefaultCommand(
    self.driveSubsystem.arcadeDrive(
        self.driveSubsystem,
        lambda: self.driverController.getLeftY()
        if self.driverController.a().getAsBoolean()
        else self.driverController.getLeftY() * 0.5,
        lambda: self.driverController.getRightX(),
    )
)
```

This example uses a shorthand if else construction called the ternary operator. This operator allows us to write a simple "if" statement in a very compact way, if the right bumper is pressed, we pass the full value, if not we multiply it by .5.

6.2 Changing Drive Type

The last likely change we will cover is changing from Arcade Drive to Tank Drive. Unlike Arcade drive which maps one axis to rotation and one to forward/back, Tank drive maps one axis (generally the Y-axis) to each side of a differential drivetrain. To make this change, you'll have to reach beyond RobotContainer as the provided drivetrain subsystems don't expose a tank drive method. In the appropriate drivetrain subsystem (PWMDriveSubsystem or CANDriveSubsystem) make a new method called tankDrive(). This method should look a lot like the arcadeDrive method. Then, modify the default command mapping in RobotContainer to use this new method with the appropriate joystick axis.

Example:

```
42      def tankDrive (self, left: float, right: float) -> None:
43          self.drive.tankDrive(left, right)
```

OR

```
def tankDrive(
    self,
    driveSubsystem,
    xSpeed: lambda xSpeed: xSpeed,
    zRotation: lambda zRotation: zRotation,
) -> commands2.Command:
    return commands2.cmd.run(
        lambda: self.drive.tankDrive(xSpeed(), zRotation()), driveSubsystem
    )
```

6.3 Developing Autonomous Routines

The provided code contains a very basic autonomous mode that drives forwards at ½ power for 1 second. Additional autonomous modes can be developed, see the Hatchbot Inlined example or Hatchbot Traditional example for a project with multiple autos.

It's common (but definitely not required!) to have multiple autonomous routines that you may wish to run based on different starting locations or strategies. If you pursue this, the most common way to choose between them for each match is to select between them using a SendableChooser on the dashboard.