

Introduction to Pybricks for FLL Teams

MD FLL Mentors Group
27 September 2025
Rev 18

Boyd Fletcher
flcoachboyd@gmail.com

Section Overview

- Section 1 – Introduction to Pybricks
- Section 2 – A comparison of Pybricks' Block Coding and Python modes
- Section 3 – How to use `hub_menu()`
- Section 4 – Overview of Core Pybricks Python Functions
- Section 5 – Building & Using a Pybricks Robot Library
- Section 6 – Robot Game
- Conclusion

Section 1 – Introduction to Pybricks

Introduction

- This presentation is intended as an introduction to Pybricks for FLL Coaches and team members
- This presentation will provide quick comparison of Pybricks Block Coding and Python modes followed by a core essentials overview of Pybricks Python
 - Line-by-line details for block coding and python code are in the slide notes for each slide
- This is part of a **new** series of presentations and documentation that is being developed by the MD FLL Mentors Group to help coaches and their teams succeed at FLL
 - The information will be posted on the <https://marylandfll.org> website later this year
- This presentation, examples, and my Python course are available here:
 - https://github.com/bcfletcher/lego_projects/tree/main/FLL/2025_MD_FLL_Coaches_Conf

Caveats

- Using Pybricks does not require you to know Python if you use the Block Coding mode
- Knowing at least one modern programming language (C/C++/C#, Java, Perl, JavaScript) will help with using Python in Pybricks
- Effective Pybricks programs require basic knowledge of Python
- Everyone needs to know the basic program structure rules for Python, This is covered in a separate presentation we have
- This course is NOT a complete overview of Pybricks – it is an introduction to get you and your team started using it.
- All the code in this course are released under the Apache License v2.0
 - <https://www.apache.org/licenses/LICENSE-2.0.txt>

Block Coding vs. Python

- In MD, many elementary schools teach Scratch which is an approach to block coding developed by MIT
 - While structurally different than the Lego and Pybricks Block coding, the concepts are similar
- For elementary school students with limited typing skills block coding is probably the best approach
- For middle schoolers, Python offers a lot of advantages
 - Prepares them for high school level programming classes in Python and Java
 - Python has much better capabilities to support finding bugs in your code (called debugging)
 - It is much easier to support code reuse and robot libraries in Python than block coding
 - Using Python lets you do nifty things like be able to run a bunch of steps (parts of the task that are working well) then “stopping” after a certain number of steps so the team can “go step-by-step” as they debug a new parts of the task

Why use Pybricks (1/3)

- It is awesome!
- Supports Lego Technic, Spike Prime/Robot Inventor, & EV3 Robotics Kits
- Supports both Block Coding (graphical) and Python coding
 - The backend of the Block Coding is actually Python so it provides a way to transition from Block Coding to Python
 - Pybricks' Python version is free and its block coding version is available for a small fee (to support its development)
- Designed specifically for Lego Robotics!
 - Robust and easy to use support for Spike Prime's Gyroscope
 - Simple but powerful commands to drive straight, turn in place, and drive & turn (called a curve/arc)
 - Very precise but easy to use control over the Hub, Motors, and Sensors

Why use Pybricks (2/3)

- Enables code reuse and good coding best practices by storing all code in .py (Python) text files
 - Supports importing of code libraries to improve sharing and reuse of code
 - Lego Spike Prime stores its code in a proprietary binary format and does **NOT** support using code libraries
- Pybricks uses an HTML/JavaScript (web) based interface (requires Chrome)
 - Can be “downloaded” to the computer so Internet access is not required
- Works on Windows, Macs, Linux, and most Chromebooks
- Supports use of text-based code editors like Microsoft’s free Visual Studio Code
 - See <https://code.visualstudio.com>
 - See <https://pybricks.com/project/pybricks-other-editors>

Why use Pybricks (3/3)

- Easy Multitasking

- Allows the robot to do multiple things at the same time
 - Like driving backward while turning and lifting an attachment while moving away from a mission model
- Frequently used to drive or turn while also moving the attachment into the right position
 - This can save valuable seconds during the robot game
- Used for stall detection for attachments to prevent the robot from “locking in place” while trying to move an attachment that is somehow gotten stuck on a mission

- `hub_menu()`

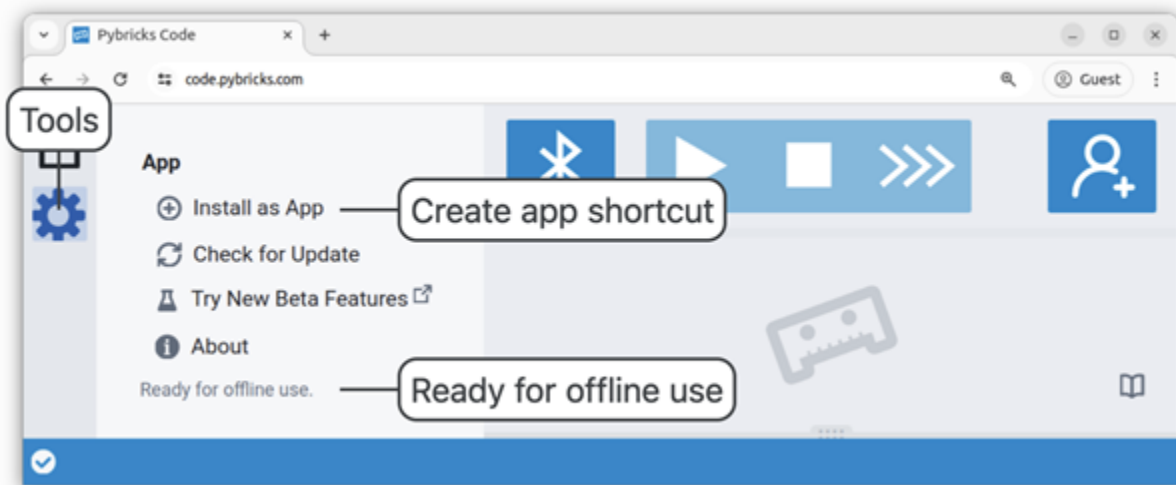
- Provides a way to run multiple tasks on the robot
 - Each task selected by use the <- -> keys on the hub and pressing the big button to run
- Loads all the tasks at one time onto the robot
- Replaces the task download capability in Lego Spike Prime application
- After each execution of a menu option, the menu starts from the beginning

Where do I get Pybricks?

- It is a open source project so the source code is available
- The software:
 - Using the Chrome web browser go to: <https://code.pybricks.com>
- The documentation:
 - <https://docs.pybricks.com/en/latest>
- The Getting Started Guide:
 - <https://pybricks.com/learn>
- Support Site:
 - <https://github.com/pybricks/support>

Installing Pybricks for Offline Use

- Go to `code.pybricks.com` using **Chrome**
- Press the Tools Gear icon, then press “**Install as App**”
- You will then only need Internet access to install new firmware on the robot and to update the app, which is only a few times a year
- You may need admin access on the computer to install the application



Pybricks Factoids (1/2)

- All distance measurements are in millimeters (mm)
- To move in the opposite direction you usually use a minus (-) sign before the distance or angle
- Both Block Coding and Python can use comments to document your code
- Use the `print()` function to help with debugging your code
 - The output of the print function is sent from the robot to the Pybricks Editor to be displayed in the output window.
 - It also works with Pybricks when using Visual Studio Code as the user interface.

Pybricks Factoids (2/2)

- To use Pybricks, the firmware (e.g., operating system) on your Lego Spike hub will need to be changed
 - You can revert back to Lego's firmware if you decide you don't like Pybricks
- Once the hub is updated to Pybricks and you have installed the Pybricks app in Chrome; an Internet connection is no longer needed
- Gyroscope makes driving long distances reliable
 - Better resilience against flaws in the game table
 - A “bump” at an mission doesn't necessarily ruin a run
 - **Always disable the gyro at the end of a task**
 - So if you pick the robot up it doesn't go crazy!
- Pybricks runs the open source MicroPython interpreter on the robot

Pybricks User Interface (UI)

Press to run your code, called the **Play** button

Press to stop your code, called the **Stop** button

Used to manually run Python commands on robot using the output window (not commonly used)

The list of programs currently loaded into the Pybricks User Interface

The Palette of available block coding commands

Your program in block code

Your program in Python - updated automatically as you change the block code

Output window that shows the output of your `print` statements, code syntax errors, and errors from the robot

Pybricks User Interface (UI)

Back up all the files in the app to disk. Places them a zip file with date stamp.

VERY USEFUL!!

Rename a file (in app not on disk)

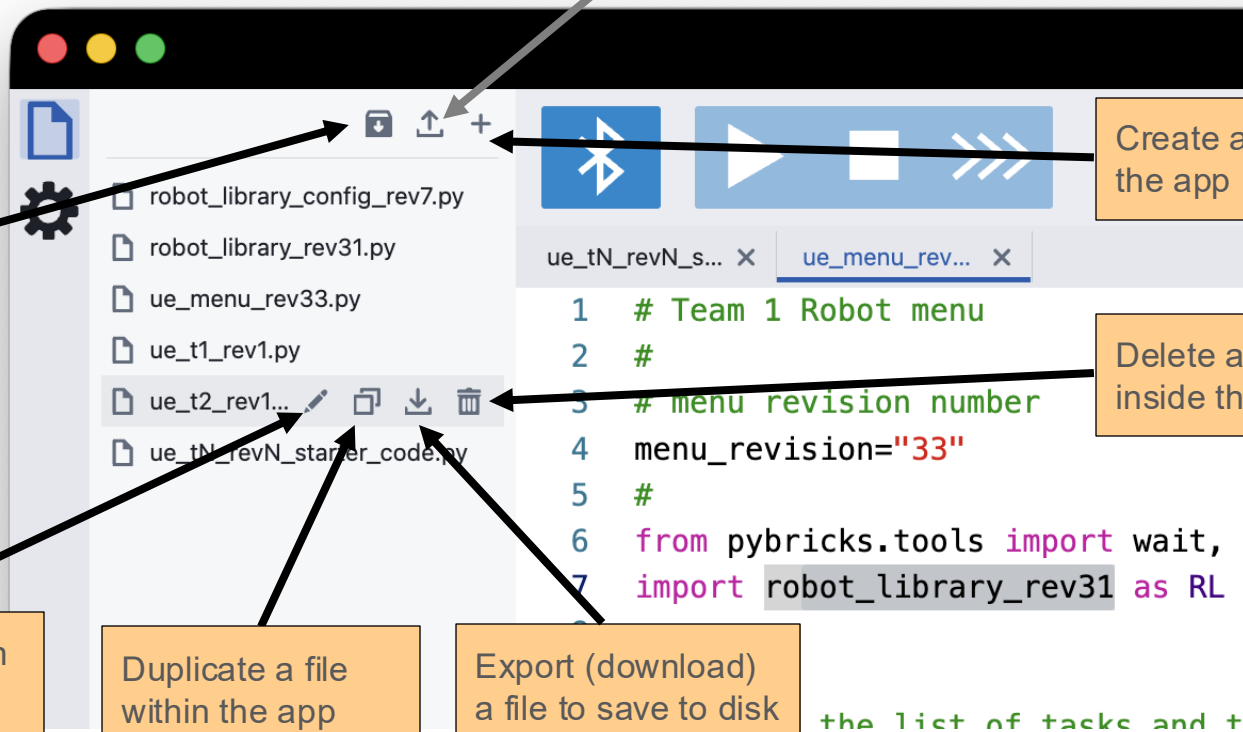
Duplicate a file within the app

Export (download) a file to save to disk

Import (upload) a file into the app

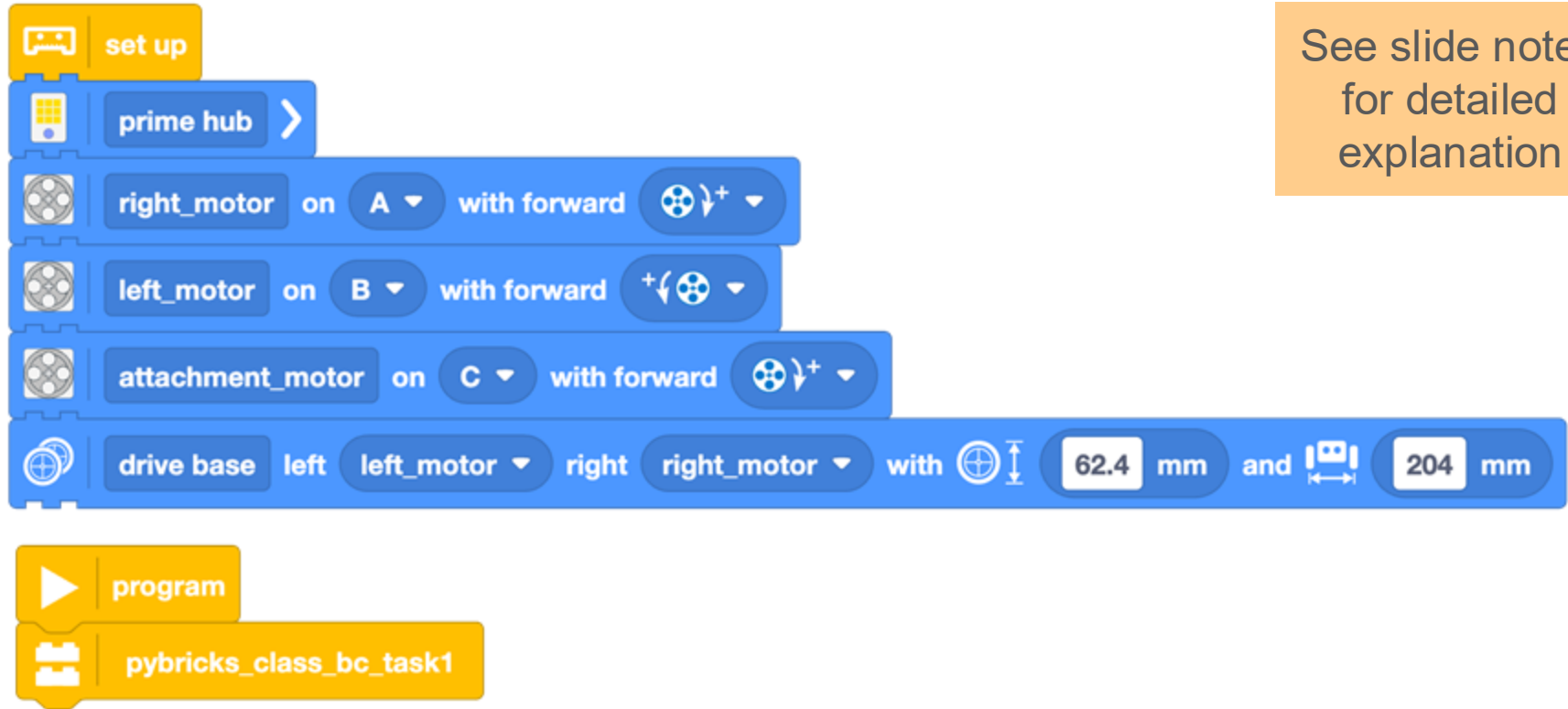
Create a new file in the app

Delete a file from inside the app



Section 2 - Pybricks Python vs. Block Coding Comparison

My First Program: Block Code - Initializing the Robot



See slide notes
for detailed
explanation

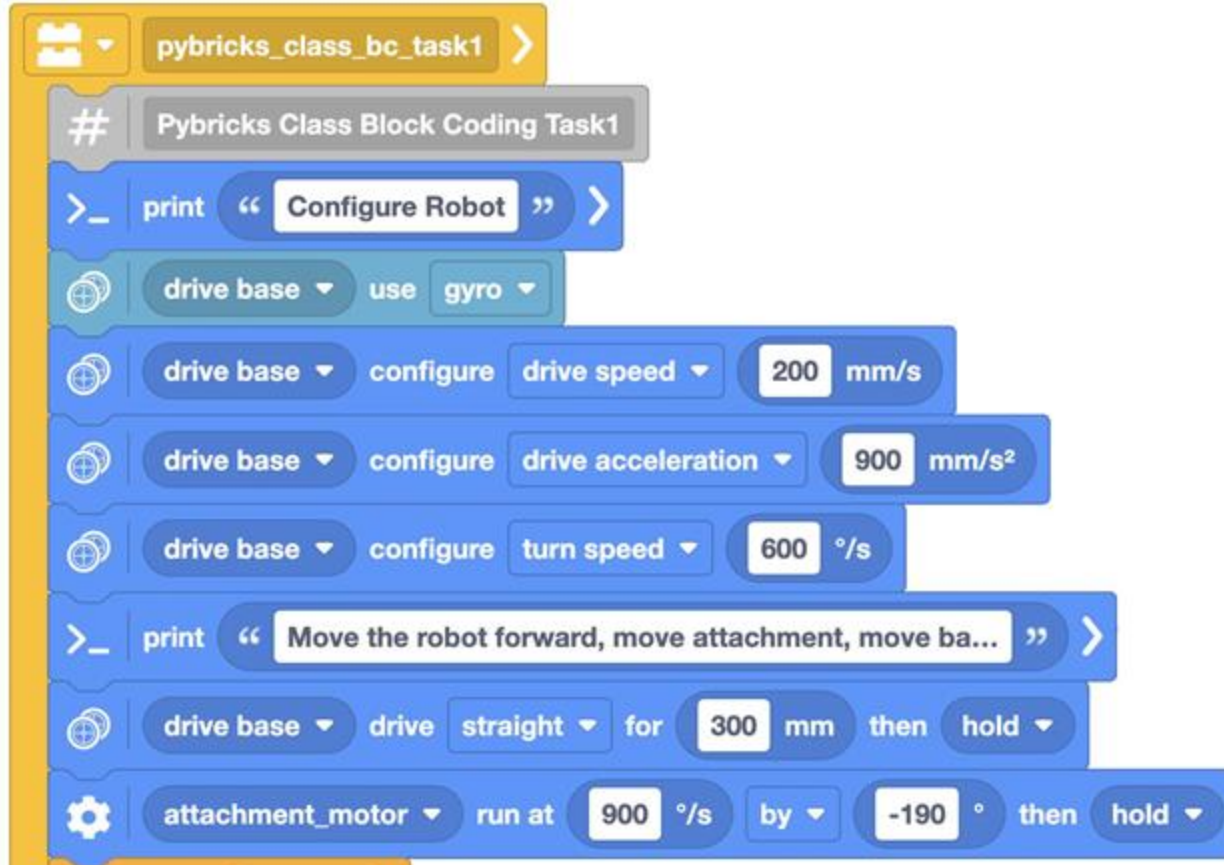
My First Program: Python - Initializing the Robot

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Direction, Port, Stop
from pybricks.pupdevices import Motor
from pybricks.robotics import DriveBase

# Set up all devices.
prime_hub = PrimeHub()
right_motor = Motor(Port.A, Direction.CLOCKWISE)
left_motor = Motor(Port.B, Direction.COUNTERCLOCKWISE)
attachment_motor = Motor(Port.C, Direction.CLOCKWISE)
drive_base = DriveBase(left_motor, right_motor, 62.4, 204)
```

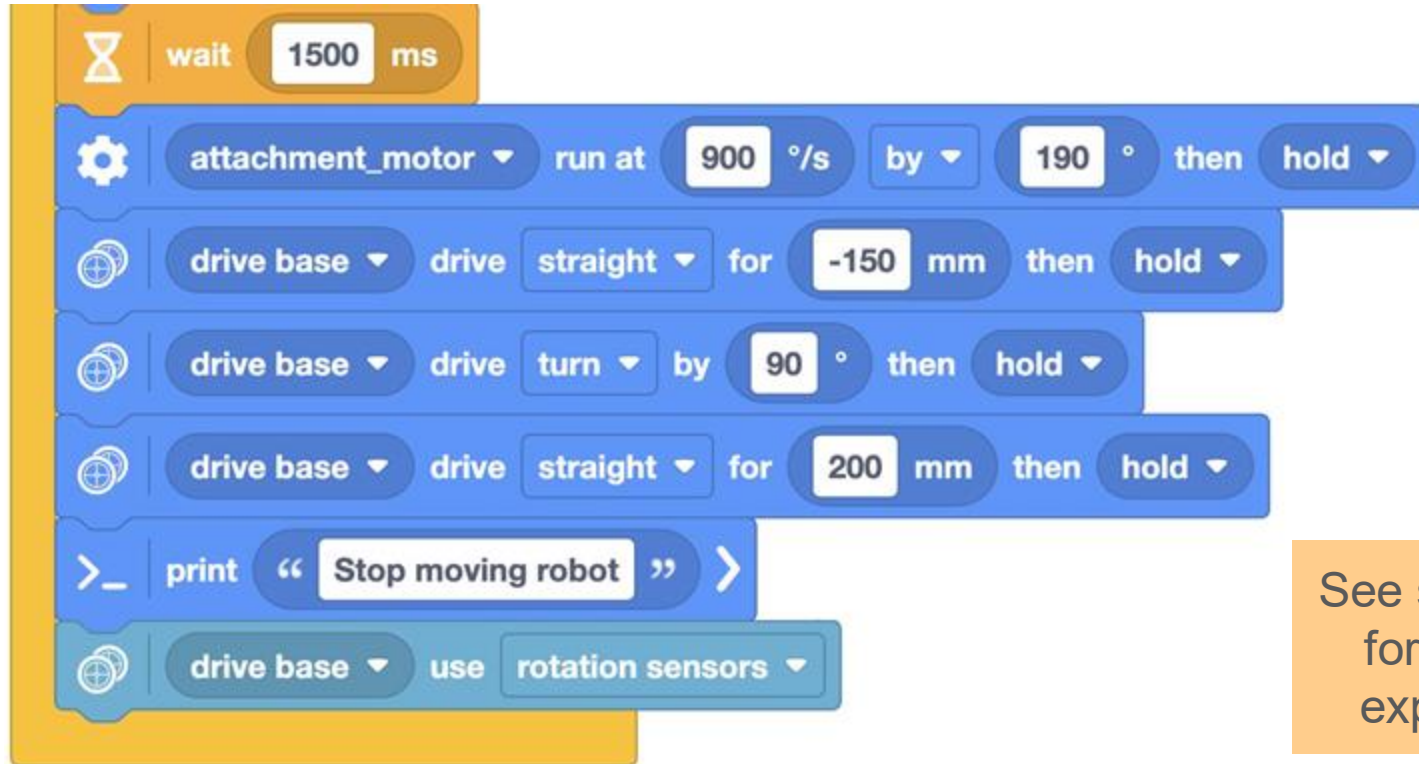
See slide notes
for detailed
explanation

My First Program: Block Code - Do something useful Part 1



See slide notes
for detailed
explanation

My First Program: Block Code - Do something useful Part 2



See slide notes
for detailed
explanation

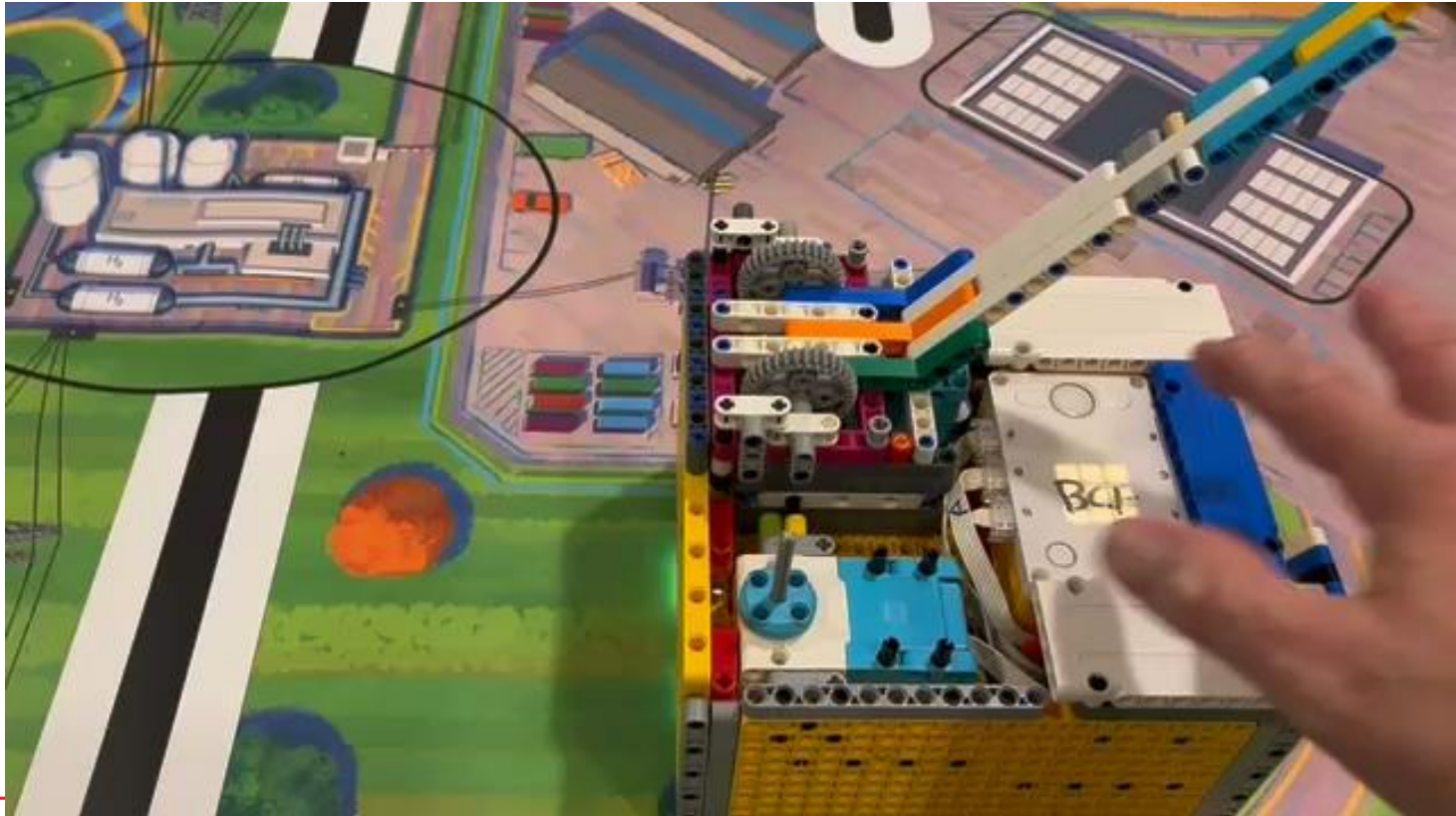
My First Program: Python

```
def pybricks_class_bc_task1():
    # Pybricks Class Block Coding Task1
    print('Configure Robot')
    drive_base.use_gyro(True)
    drive_base.settings(straight_speed=200)
    drive_base.settings(straight_acceleration=900)
    drive_base.settings(turn_rate=600)
    print('Move the robot forward, move attachment, move back, and turn')
    drive_base.straight(300)
    attachment_motor.run_angle(900, -190)
    wait(1500)
    attachment_motor.run_angle(900, 190)
    drive_base.straight(-150)
    drive_base.turn(90)
    drive_base.straight(200)
    print('Stop moving robot')
    drive_base.use_gyro(False)

# The main program starts here.
pybricks_class_bc_task1()
```

See slide notes
for detailed
explanation

Example Running



Blocking Coding vs. Python (1/2)

- Most of the core functionality of Pybricks Python is available in Block Code
- Block Code will generate Python “on-the-fly” as blocks are placed, moved, and removed
- When saving block code to disk, there is a comment line (#) at the beginning of the file.
 - **DO NOT** edit it or Pybricks will not be able to read the block code file

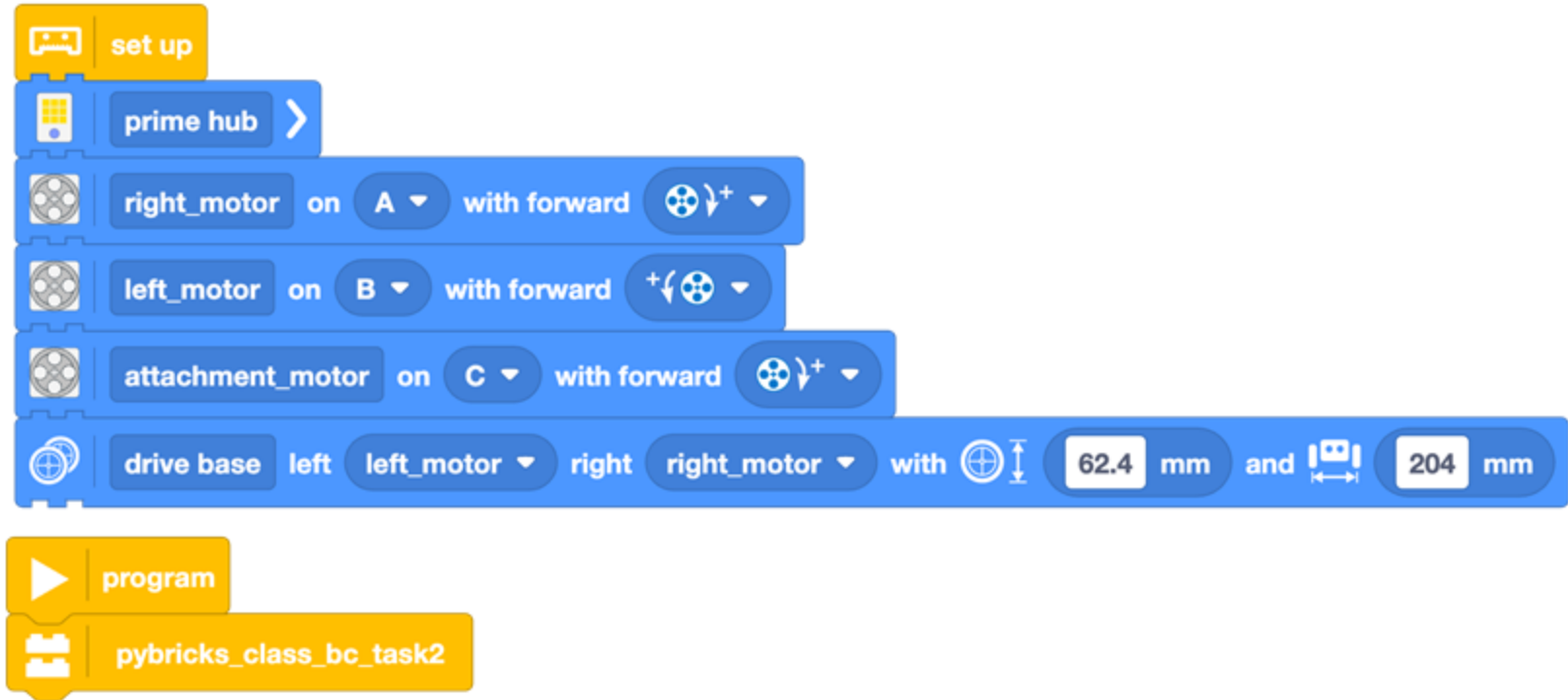
Blocking Coding vs. Python (2/2)

- Due to how block coding mode structures the file it does not work easily with **hub_menu()**
 - Basically you can only run the task once - which will cause problems in FLL competition if the robot glitches and you need to rerun the task
- Block Coding is best used as a transition from Scratch like coding to Python
- It can also be used to rapidly test out ideas for missions and tasks then save the file, remove the comment line, and then switch over to straight Python files for competition

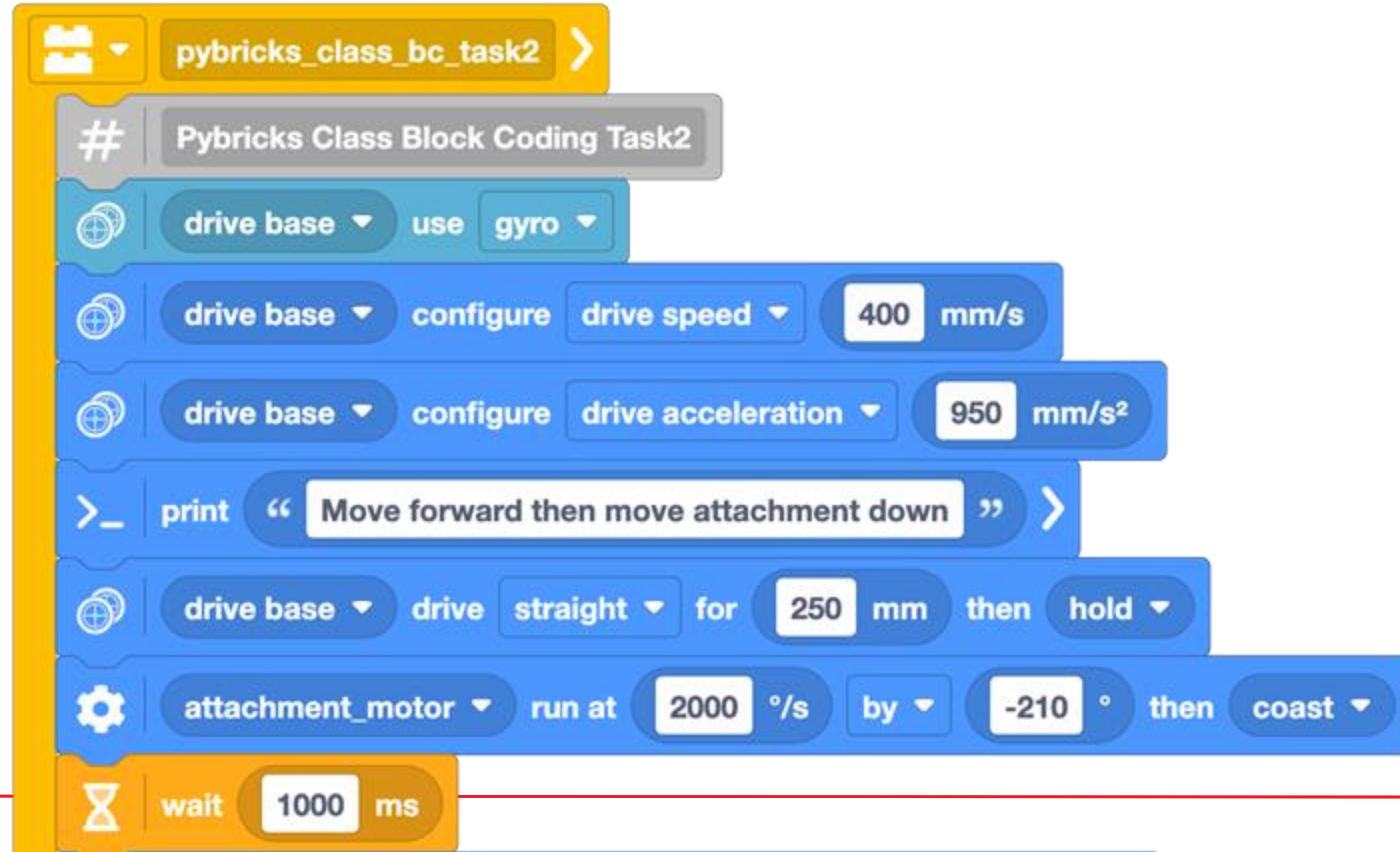
More Examples

- The next two examples demonstrate how use multitasking and variables in block coding and Python
- The two Python examples have been modified to use a simple robot library so they can be used to demonstrate how to use `hub_menu()`. The core logic is the same as the block code.

pybricks_class_bc_task2(): Part 1



pybricks_class_bc_task2(): Part 2



pybricks_class_bc_task2(): Part 3

```
>_ print "Move backward while moving the attachment up" >
```

multitask until all done

- drive base straight for -250 mm then hold
- attachment_motor run at 150 %/s by 210 ° then hold

>_ print "Stop moving robot" >

drive base use rotation sensors

Multitask code section

pybricks_class_basic_task2()

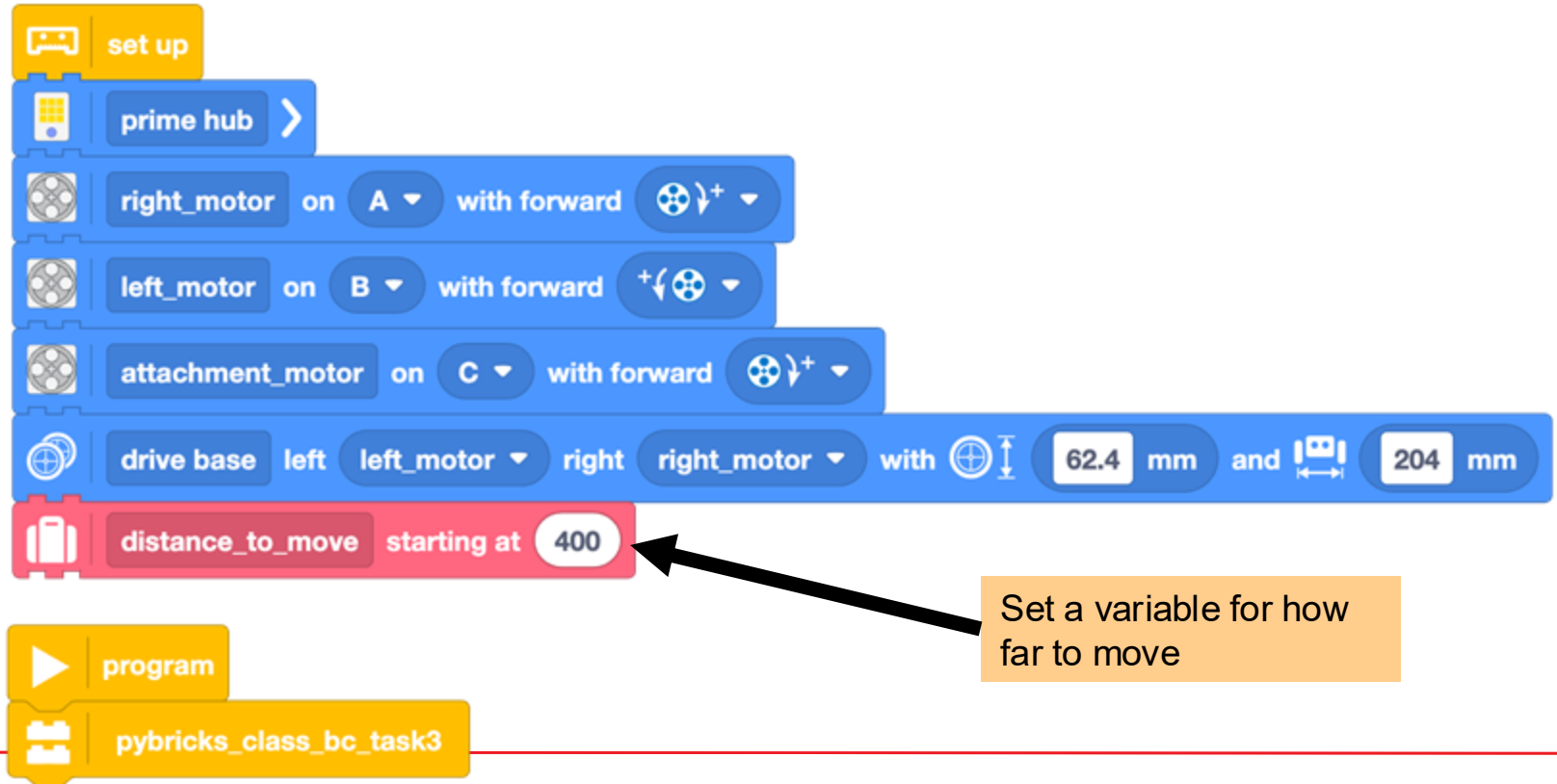
Pybricks Class Task 2

```
from pybricks_class_small_lib_rev3 import *  
async def pybricks_class_task2():  
    # Pybricks Class Block Coding Task2  
    print("Running Pybricks Class Task 2")  
    drive_base.use_gyro(True)  
    drive_base.settings(straight_speed=400, straight_acceleration=950)  
    print('Move forward then move attachment down')  
    await drive_base.straight(250)  
    await attachment_motor.run_angle(2000, -210, Stop.COAST)  
    await wait(1000)  
    print('Move backward while moving the attachment up')  
    await multitask(  
        drive_base.straight(-250),  
        attachment_motor.run_angle(150, 210)  
    )  
    print('Stop moving robot')  
    drive_base.use_gyro(False)  
#run_task(pybricks_class_task2())
```



Multitask code

pybricks_class_bc_task3(): Part 1



Set a variable for how far to move

pybricks_class_bc_task3(): Part 2

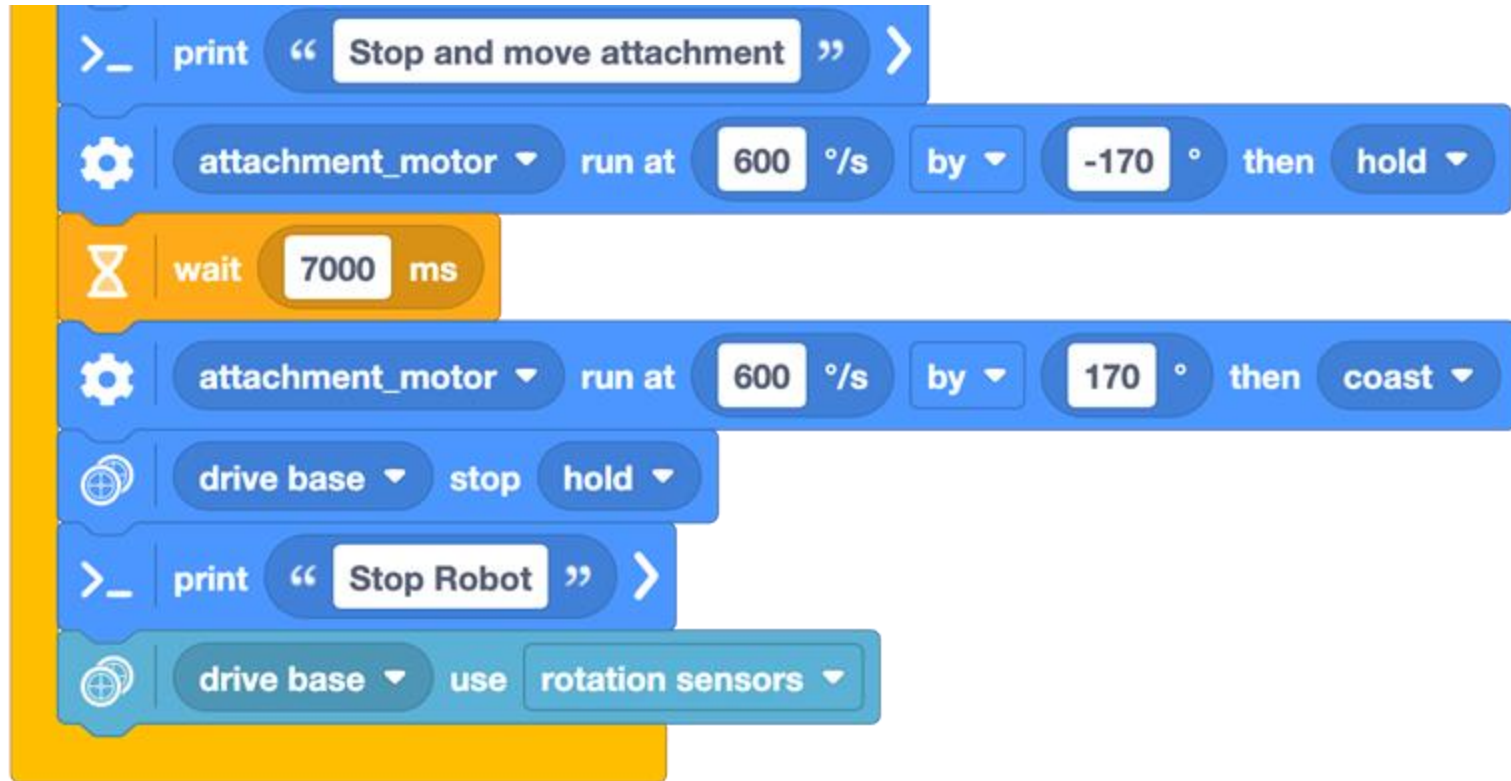
The screenshot shows a PyBricks code editor with the following script:

```
# Basic driving and attachment demonstration for a technic hub based robo...  
drive base use gyro  
drive base configure drive speed 500 mm/s  
>_ print "Drive Forward by" distance_to_move "mm"  
drive base drive straight for distance_to_move then brake  
distance_to_move set to -350  
>_ print "Drive Backward by" distance_to_move "mm"  
drive base drive straight for distance_to_move then brake
```

Annotations:

- Use the variable in **RED** ovals
- Changed value of variable

pybricks_class_bc_task3(): Part 3



pybricks_class_basic_task3()

Pybricks Class Task 3

from pybricks_class_small_lib_rev3 import *

async def pybricks_class_task3():

Pybricks Class Block Coding Task3

print("Running Pybricks Class Task 3")

distance_to_move = 400

drive_base.settings(straight_speed=200, straight_acceleration=600)

print('Drive Forward by', distance_to_move, 'mm')

await drive_base.straight(distance_to_move, then=Stop.BRAKE)

distance_to_move = -350

print('Drive Backward by', distance_to_move, 'mm')

await drive_base.straight(distance_to_move, then=Stop.BRAKE)

print('Stop and move attachment')

attachment_motor.control.limits(speed=950, acceleration=950)

print('Stop Robot')

drive_base.use_gyro(False)

#run_task(pybricks_class_task3())

Importing our library

Setting the variable

Examples of using it

Changed value of variable

Section 3 – How to use `hub_menu()`

Hub Menu or “how do I store and run multiple tasks on my robot”

- In the previous “My First Program” example (BC & Python) and Tasks 2 and 3 (BC version), the program being loaded will be the only program on the robot
- Lego Spike robots can store multiple programs
- During competition the team will want to store their tasks as individual programs and be able to use the hub menu to select the task
- The `hub_menu()` function is used to configure the robot for multiple tasks
 - Use with Pybricks Block Coding is not straightforward and has limitations

Using `hub_menu()` (1/2)

- `hub_menu()` should be its own dedicated task
 - It should be in its own file in the Pybricks user interface
- The robot setup in “Initializing the Robot” slide above should be moved to a separate “library”
 - Ex: `pybricks_class_library_rev1.py`
 - This is necessary to prevent Python from trying to create more than one instance of the Prime Hub
- In Pybricks, when you press the Play Button while in the `hub_menu()` code and it will push code for the library and all the tasks used the `hub_menu()` to the robot
 - The reason is due to the `import` statements at the beginning of the `hub_menu()` program

Using `hub_menu()` (2/2)

- Each task's Python file must have:
 - A unique name for the task's main function:
 - Ex: `pybricks_class_basic_task1()`
 - The `run_task(pybricks_class_basic_task1)` must be commented out if you want to run the task via `hub_menu()`
 - Ex: `#run_task(pybricks_class_basic_task1)`
 - Each task needs to import the library at the top of the file
 - Ex: `from pybricks_class_basic_lib_rev1 import *`
- The following `hub_menu()` example is for the Python code not the Block Code

pybricks_class_basic_task1() – Revised for hub_menu()

Pybricks Class Task 1

from pybricks_class_basic_lib_rev1 import *

def pybricks_class_basic_task1():

Pybricks Class Block Coding Task1

print("Running Pybricks Class Task 1")

print('Configure Robot')

drive_base.use_gyro(True)

drive_base.settings(straight_speed=200)

drive_base.settings(straight_acceleration=900)

drive_base.settings(turn_rate=600)

print('Move the robot forward, move attachment, move back, and turn')

drive_base.straight(300)

attachment_motor.run_angle(900, -190)

wait(1500)

attachment_motor.run_angle(900, 190)

drive_base.straight(-150)

drive_base.turn(90)

drive_base.straight(200)

print('Stop moving robot')

drive_base.use_gyro(False)

#run_task(pybricks_class_task1())

Basic Robot Library (pybricks_class_basic_lib_rev1.py)

```
from pybricks.hubs import PrimeHub
from pybricks.pupdevices import Motor, ColorSensor, UltrasonicSensor, ForceSensor
from pybricks.parameters import Button, Color, Direction, Port, Side, Stop
from pybricks.robotics import DriveBase
from pybricks.tools import wait, StopWatch, multitask, run_task, hub_menu
```

Set up all devices.

```
prime_hub = PrimeHub()
right_motor = Motor(Port.A, Direction.CLOCKWISE)
left_motor = Motor(Port.B, Direction.COUNTERCLOCKWISE)
attachment_motor = Motor(Port.C, Direction.CLOCKWISE)
drive_base = DriveBase(left_motor, right_motor, 62.4, 204)
```

Load all Pybricks functions you will need


Initialize the Hub, Motors, and DriveBase

- This library is imported into the `hub_menu()` task and the each of three robot mission tasks
- It instantiates the `PrimeHub`, 3 `Motors`, and `DriveBase` objects so they can be used by `hub_menu()` and the three mission tasks
- The objects represent the hardware so if you try to instantiate them more than once, you will get an error

hub_menu () Example Part 1 (pybricks_class_basic_menu_rev3.py)

```
# Pybricks Class Menu
# menu revision number
menu_revision="3"
#
# import necessary libraries and functions
from pybricks_class_small_lib_rev3 import *

print("Robot Menu, rev#", menu_revision)
print("Load Task1")
from pybricks_class_task1_rev3 import pybricks_class_task1
print("Load Task2")
from pybricks_class_task2_rev3 import pybricks_class_task2
print("Load Task3")
from pybricks_class_task3_rev3 import pybricks_class_task3
```



Importing our library

Import each of the 3 robot tasks into the `hub_menu ()` so it can call (run) them

hub_menu() Example Part 2

Run this loop forever

```
print("Running Hub Menu")
while True:
    selected = hub_menu("M","1","2","3","X")
    try:
        if selected == "M":
            # does nothing, used to show menu is active
            break
        if selected == "1":
            print("Task 1 Selected")
            wait(1000)
            run_task(pybricks_class_task1())
        elif selected == "2":
            print("Task 2 Selected")
            wait(1000)
            run_task(pybricks_class_task2())
        elif selected == "3":
            print("Task 3 Selected")
            wait(1000)
            run_task(pybricks_class_task3())
        elif selected == "X":
            # does nothing used to show end of menu
            break
    except BaseException as menuException:
        print("Stop was Pressed or a Critical Error Occurred.")
        print(menuException)
        break
```

- Set the characters that will be displayed on the hub menu.
- 'M' is used to show the menu is active
- Each number is associated with a task
- 'X' is used to indicate end of the menu

Run each mission task if the right number on the hub menu (1,2, or 3) is selected

wait(1000) is a 1 sec delay so fingers can get out of the way before the robot starts moving

hub_menu() Example: Part 1

```
# Pybricks Class Menu
# menu revision number
menu_revision="1"
#
# import our library
from pybricks.tools import hub_menu
from pybricks_class_library_rev1 import *

# The name of each task file minus the .py file extension
from pybricks_class_task1_rev1 import pybricks_class_task1
from pybricks_class_task2_rev1 import pybricks_class_task2
from pybricks_class_task3_rev1 import pybricks_class_task3

print("Robot Menu, rev#", menu_revision)
```

See slide notes for detailed explanation

This hub_menu example is intended to work with the Pybricks Python Examples below.

hub_menu() Example: Part 2

This is

```
while True:
    selected = hub_menu("M", "1", "2", "3", "X")
    try:
        if selected == "M":
            # does nothing, used to show menu is active
            break
        if selected == "1":
            wait(1000)
            run_task(pybricks_class_task1())
        elif selected == "2":
            wait(1000)
            run_task(pybricks_class_task2())
        elif selected == "3":
            wait(1000)
            run_task(pybricks_class_task3())
        elif selected == "X":
            # does nothing, used to show end of menu
            break
    except BaseException as menuException:
        print("Stop was Pressed or a Critical Error Occurred")
        print(menuException)
        break
```

See slide notes for detailed explanation

This hub_menu example is intended to work with the Pybricks Python examples above.

Example of all 3 Mission Running



Section 4 – Overview of Core Pybricks Python Functions

Introduction to Pybricks Functions

- Pybricks provides excellent documentation and examples and should be considered the authoritative source for information
- However, there are a large number of commands
- Most are not needed to be effective in solving FLL missions
- The next set of slides covers the commands we use in our robots
- Not all features of all commands are covered
- When in doubt use the official Pybricks documentation
- The following slides include a combination of the official Pybricks documentation with additions/clarifications

Components of a Pybricks program

Pybricks Programs usually consist of the following parts (in order):

1. Import Pybricks Libraries

- The python command important libraries and functions **from** for use by our program
- Ex: `from pybricks.hubs import PrimeHub`

2. Import any support libraries the team wrote that you need

3. Create any support functions that you need for the new task

4. Create the "main" function, usually named after the task

- Ex: `submerged_task1()`

5. Add `run_task()` at end of program which is used to execute your "main" function on the robot

- Ex: `run_task(submerged_task1())`
- **Note:** Putting `run_task()` in your task's code is normally not done when `hub_menu()` is being used. You use a comment (`#`) at the beginning of the line to disable it when using `hub_menu()`.

await()

- For any Pybricks command that takes time to complete **before** moving on to the next command you **must** prepend with **await**
 - Used before nearly all commands to the robot that result in the robot actually doing something (e.g., moving a motor, reading a sensor, making beeps) and all commands involving a time element
- In the Pybricks documentation, if await precedes the command then it must be used with **await**
- Failure to do this **will** result in incorrect and unpredictable behavior with the robot

Pybricks Multitasking

- Pybricks supports running multiple functions concurrently
 - Pybricks uses the term “coroutine” which also called a function
- Any Pybricks command that is prefixed with **await** can be used in multitasking and **await** must be used with the command or the results will be indeterminate
- **run_task**(*function_name*) is used to run one of your functions from start to end while blocking all other functions
 - When using **hub_menu()** the **run_task()** in your tasks should be commented out (put a # at the beginning of the line)
- **multitask**(*function_name1*, *function_name2*, *function_nameN*) is used to run multiple functions at at time
 - By default, **multitask()** waits for all functions to complete
 - If you add the **race=True** parameter, **multitask()** will wait until one function completes and then it will stop the rest

Miscellaneous Tidbits

- Positive/Negative integers are used to move forward/backward for distance, angles, and up/down for attachments
 - But the orientation of the motor determines if positive is forward or backward
- Attachments should have a defined AND consistent start position
 - Teams should be encouraged to always verify the attachments' start position for each run
 - Our teams "wiggle" each attachment forward/backward and then set it in the start position for each run

Motor()

Motor(port, positive_direction=direction, gears=None or [list of gears], reset_angle=True|False, profile=None)

- Where **direction** is `Direction.CLOCKWISE` or `Direction.COUNTERCLOCKWISE`
- `Gears` is the list of gears linked to the motor. The gear connected to the motor comes first and the gear connected to the output comes last.
- Gears are represented by number of teeth. So `[12,36]` would be a 12-tooth gear connected to a 36-tooth gear. Gears are primarily used in attachments.
- Normally you have two drive motors, one must be moving `COUNTERCLOCKWISE` and the other `CLOCKWISE`
 - By default motors move `CLOCKWISE`
 - You can use `+/-` in the angle in commands like `run_target()` to change the direction
- `Motor()` has a number of functions associated with it.
- `Motor.stalled()` function is used to determine if a motor (usually an attachment has gotten stuck

Motor() Examples

- `leftMotor = Motor(LeftDriveMotorPort, Direction.COUNTERCLOCKWISE, reset_angle=True)`
- `rightMotor = Motor(RightDriveMotorPort, Direction.CLOCKWISE, reset_angle=True)`
- `leftAttachment = Motor(LeftAttachmentMotorPort, Direction.CLOCKWISE, reset_angle=True)`
 - `if leftAttachment.stalled() == True` then *doing something about it.*

DriveBase()

`DriveBase(left_motor, right_motor, wheel_diameter, axle_track)`

- `left_motor` ([Motor](#)) – The motor that drives the left wheel, use `Motor()` to create the `left_motor` object
- `right_motor` ([Motor](#)) – The motor that drives the right wheel, use `Motor()` to create the `right_motor` object
- `wheel_diameter` ([Number](#), *mm*) – Diameter of the wheels.
- `axle_track` ([Number](#), *mm*) – Distance between the points where both wheels touch the ground.
- Used to control the drive motors for the robot
- **IMPORTANT:** Review the “*Measuring and validating the robot dimensions*” section in the DriveBase documentation on how to test and tweak the `wheel_diameter` and `axle_track` settings

DriveBase () Functions

- **DriveBase.settings(*straight_speed*, *straight_acceleration*, *turn_rate*, *turn_acceleration*)**
 - Used to configure speed, acceleration
 - You need to use the `DriveBase.Settings()` function to set the speed and acceleration values for below `arc()`, `turn()`, and `straight()` functions
- **DriveBase.use_gryo(Boolean)**
 - - enables the gyroscope
- **await DriveBase.arc(*radius*, *angle=None*, *distance=None*, *then=Stop.HOLD*, *wait=True*)**
 - Used to drive in an arc based on degrees or distance
 - You can use either **Angle** or **Distance** but not both
- **await DriveBase.turn(*angle*, *then=Stop.HOLD*, *wait=Boolean*)**
 - Used to turn the robot a specific number of degrees
- **await DriveBase.straight(*dist*, *then=Stop.HOLD*, *wait=True*)**
 - Used to drive straight in mm
- Distances and Angles can be Positive (right) or Negative (left) – unless your hub is situated differently, then Pos/Neg might be opposite

DriveBase () Examples

- `driveBase = DriveBase(leftMotor,rightMotor, wheel_diameter=62.4, axle_track=204)`
- `driveBase.settings(DefaultStraightSpeed=400, DefaultTurnRate=300, DefaultStraightAcceleration=[500,100], DefaultTurnAcceleration=300)`
 - Accelerations values are for acceleration and deceleration. This sets the speed at 400 mm/s, turn rate= 300 mm/s, Straight Acceleration at 500 mm/s² and Deceleration at 100 mm/s², both turn accelerations at 300 mm/s².
- To drive straight for 1000mm slowly
 - `driveBase.settings(straight_speed=100, straight_acceleration=100)`
 - `await driveBase.straight(1000 ,then=Stop.HOLD, wait=True)`

Gyroscope

- The gyroscope (gyro for short) sensor on the hub is used to provide automatic real-time corrections to bumps and anomalies on the course
- To enable (turn-on) the gyro use: `driveBase.use_gyro(True)`
- To disable (turn-off) the gyro use: `driveBase.use_gyro(False)`
- **ALWAYS** remember to disable the gyro after every mission
 - Failure to do so could be catastrophic in the Robot Game and would require turning off the robot
 - Simple solution is disable the gyro in `hub_menu()`
 - Ideally create a function that stops the gyro and stops all the robot motors
- Remember to enable the gyro at the start of the mission, some missions might require the gyro to be disabled

Motor.run_target()

- **await run_target(*speed*, *target_angle*, *then=Stop.HOLD*, *wait=True*)**
 - Runs the motor at a constant speed towards a given target angle.
 - The direction of rotation is automatically selected based on the target angle. It does not matter if speed is positive or negative.
 - **speed** (*Number*, deg/s) – Speed of the motor.
 - **target_angle** (*Number*, deg) – Angle that the motor should rotate to.
 - **then** (*Stop*) – What to do after coming to a standstill.
 - **wait** (*bool*) – Wait for the motor to reach the target before continuing with the rest of the program.
 - **run_target** is commonly used for attachments to provide precise control over the how far it moves

then=Stop.????

- Used to control the end condition of a motor
- Multiple Pybricks commands have the **then=Stop.????** argument
- In FLL it is typically used for attachment motors to control whether or not they hold their position.
 - For example, if you want to lift an object and carry it to a different location, the **stop.HOLD** would be used to hold the attachment off the mat. If you used **stop.COAST** or **stop.BRAKE** the attachment will not stay in position.
- Valid values are:
 - **stop.COAST**
 - Let the motor move freely.
 - **stop.COAST_SMART**
 - Let the motor move freely. For the next relative angle maneuver, take the last target angle (instead of the current angle) as the new starting point. This reduces cumulative errors. This will apply only if the current angle is less than twice the configured position tolerance.
 - **stop.BRAKE**
 - Passively resist small external forces.
 - **stop.HOLD**
 - Keep controlling the motor to **hold** it at the commanded angle.
 - **stop.NONE**
 - Do not decelerate when approaching the target position. This can be used to concatenate multiple motor or drive base maneuvers without stopping. If no further commands are given, the motor will proceed to run indefinitely at the given speed.

Motor.run_target() Examples

- **await multitask(attachment.run_target(250,190, then=Stop.COAST), isStalled(attachment), race=True)**
 - Moves an attachment at 250 mm/s by 190 degrees then coasts the attachment at that position while moving the attachment it is also checking to see if the attachment motor has stalled. If it stalls it stops. If stall detection wasn't used it could lock the robot.
 - For example, in the Submerged season the team used this approach when using an attachment to release the shark in Mission 2 and flipping the coral reef in Mission 3. Without stall detect the attachment would frequently stall locking the robot in position.
- **await attachment.run_target(250,120, then=stop.HOLD, wait=True)**
 - Moves an attachment motor at 250 mm/s by 100 degrees then holds the attachment at that position ("floating" in the air)

Section 5 – Building & Using a Pybricks Robot Library

Why a Robot Library?

- “I thought you said Pybricks is easy to learn and use....”
- It is...but kids end up repeatedly using the same sequence of commands over and over but just with different values
- A Robot Library:
 - Allows students to focus on solving the robot missions
 - Combines multiple functions (e.g., steps) into a single step
 - Enables easier debugging of robot programs
 - Simplifies the robot control into English like phrases that are easy for kids to understand

Why a Robot Library?

- The robot library combines those commands into functions, which:
 - Reduces errors of manually reusing the same block of code over and over
 - Provides a mechanism to embed configurable debugging (diagnostic) statements that can be displayed when testing
 - Supports code reuse which is a major programming principle
 - Allows changes to the underlying implementation without (necessarily) changing how it is used
 - For example: to improve debugging functionally, to enable step-by-step execution, to improve error condition handling, to improve performance, improve input validation
- Teaches and enforces the concept and value of code reuse

Changes from Actual Code

- The code examples in these slides differ slightly from the actual code
- Comments may have been removed so the code could fit on the slide
- Whitespace (spaces/tabs) has been changed to so the code could fit on the slide. So indentations are likely wrong.
- Some debugging statements may have been removed to improve readability
- This particular robot library does NOT make use of light sensor. It only uses the sensors on the Lego Spike Prime Hub.

Support Functions

- **`initializeRobot()`**
 - Configures the robot
- **`stopEverything()`**
 - Disables the Gyroscope and stops the motors
- **`initializeRobotForTask()`**
 - Called at the beginning of each task to set the robot
- **`isStalled(attachment)`**
 - Internal function to detect if an attachment has got stuck, not directly called by users of library

Movement Functions (1/2)

- **driveRobot(dist, speed, accel)**
 - Drive robot forward or backward at a specific speed and acceleration
- **driveRobotAndLift(dist, accel, attachmentStr, degrees, aSpeed, stopMode, resetAngle, stallDetect)**
 - Drive robot forward or backward while lifting an attachment up or down
- **turnRobot(dir, turnAccel, turnRate)**
 - Turn robot in a specific direction at a specific speed (rate) and acceleration
- **turnRobotAndLift(dir, turnAccel, turnRate, attachmentStr, degrees, aSpeed, stopMode, resetAngle, stallDetect)**
 - Turn robot while lifting up or down while lifting an attachment up or down

Movement Functions (2/2)

- **driveRobotInArc(radius, mode, angleOrDist, speed, accel, turnAccel, turnRate)**
 - Drive robot forward or backward in arc for a specific angle or distance at a specific speed, acceleration, and turn rate
- **driveRobotInArcAndLift(radius, mode, angleOrDist, attachmentStr, degrees, aSpeed, stopMode, resetAngle, stallDetect)**
 - Drive robot forward or backward in arc for a specific angle or distance at a specific speed, acceleration, and turn rate while lifting an attachment up or down
- **moveAttachment(attachmentStr, degrees, speed, stopMode, resetAngle, stallDetect)**
 - Move an attachment by a specified number of degrees at specific speed with optional stall detection

Robot Library Header

```
# Basic FLL Robot Library
#
# This code is released under the Apache License v2.0
# https://www.apache.org/licenses/LICENSE-2.0.txt
# Developed against Pybricks 3.6.1
#
# Update this if you change the code
coreLib_revision="4"
coreLib_date="14 September 2025"
#
#
from pybricks.hubs import PrimeHub
from pybricks.pupdevices import Motor, ColorSensor, UltrasonicSensor, ForceSensor
from pybricks.parameters import Axis, Button, Color, Direction, Port, Side, Stop
from pybricks.robotics import DriveBase
from pybricks.iodevices import XboxController
from pybricks.tools import wait, StopWatch, multitask, run_task, hub_menu
from umath import sin, pi
from pybricks.hubs import PrimeHub
#
# change this for your configuration file
#
# define and set the debug levels
debugL1=True
debugL2=False
```

Change these when you update the library

This imports the majority of the Pybricks functions you will likely need if your programs.

debugL1

- Minimal debug output
- We typically use L1 for our robot tasks

debugL2

- Large amount of debug output

Both should be set to **False** for production (competition)

initializeRobot() (1/4)

```
def initializeRobot():  
    global hub  
    global leftAttachment  
    global leftAttachmentName  
    global rightAttachment  
    global rightAttachmentName  
    global attachmentsPresent  
    global driveBase  
    global leftMotor  
    global rightMotor  
    global leftDriveMotorPort  
    global rightDriveMotorPort  
    global leftAttachmentMotorPort  
    global rightAttachmentMotorPort  
    global leftAttachmentHasGears  
    global rightAttachmentHasGears
```

Declare a bunch of variables as global variables so they can be updated in the initializeRobot() function

```
# initialize the robot's  
hub = PrimeHub()  
# print some information about the robot library  
if debugL1: print("Basic Robot Library Rev#", coreLib_revision, "Release Date: ", coreLib_date)  
if debugL1: print(" Initializing Robot")  
if debugL1: print(" Using hub named: ", hub.system.name())
```

Initial the Lego Spike Prime hub so that it can be used

Purpose: Initialize the robot by configuring the Prime Hub, DriveBase, drive motors, and attachment motors

initializeRobot() (2/4)

Continued from Previous Slide

```
# set the hub LED display to the correct orientation
hub.display.orientation(Side.RIGHT) # Side.TOP is the normal value
```

```
# set this to true if there are attachments
attachmentsPresent=True
```

```
# the port locations for the light sensors, drive motors,
# and attachment motors
```

```
LeftDriveMotorPort=Port.B
RightDriveMotorPort=Port.A
RightAttachmentMotorPort=Port.C
LeftAttachmentMotorPort=Port.D
```

```
if debugL2: print("DEBUG-L2 - Assigning & Configuring Motors")
leftMotor = Motor(LeftDriveMotorPort, Direction.COUNTERCLOCKWISE,
                  reset_angle=True)
rightMotor = Motor(RightDriveMotorPort, Direction.CLOCKWISE,
                  reset_angle=True)
```

Tells the Prime Hub how the hub is oriented in/on the robot. See Pybricks docs for more detail

Assign the ports on the Prime Hub to the two drive motors and the two attachment motors

Initialize motors. The motor on the left must turn counterclockwise to make the robot go forward.

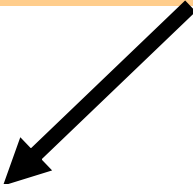
initializeRobot() (3/4)

Continued from Previous Slide

Initialize the DriveBase and configure wheelDiameter & axleTrack **that are specific for your robot!!** Both values are in millimeters (mm).

- axleTrack is the measurement between the centerpoints of the both wheel treads
- The axleTrack & wheelDiameter likely will need to tweaking.
- Read the Pybricks doc on DriveBase in the subsection "Measuring and validating the robot dimensions" on how tune these settings

```
#  
# Setup drive base and default performance parameters  
if debugL2: print("DEBUG-L2 - Configuring the DriveBase")  
wheelDiameter=62.4  
axleTrack=204  
driveBase = DriveBase(leftMotor,rightMotor, wheel_diameter=wheelDiameter, axle_track=axleTrack)
```



initializeRobot() (4/4)

Continued from Previous Slide

```
DefaultStraightSpeed=400
DefaultStraightAcceleration=[500,500]
DefaultTurnAcceleration=[500,500]
DefaultTurnRate=300
if debugL2: print("DEBUG-L2 - Configuring the Speed & Acceleration default settings")
```

Define some default speed and acceleration motor settings for your drive motors

```
driveBase.settings(straight_speed=DefaultStraightSpeed,
                    straight_acceleration=DefaultStraightAcceleration,
                    turn_acceleration=DefaultTurnAcceleration,
                    turn_rate=DefaultTurnRate)
```

Actually, configure the DriveBase to use the above settings

```
if attachmentsPresent == True:
    if debugL2: print("DEBUG-L2 - Configuring Attachment Motors")
    leftAttachment = Motor(LeftAttachmentMotorPort,Direction.CLOCKWISE,reset_angle=True)
    rightAttachment = Motor(RightAttachmentMotorPort,Direction.CLOCKWISE,reset_angle=True)
```

Assign and configure the two attachment motors

moveAttachment(attachmentStr, degrees, speed, stopMode, resetAngle, stallDetect) (1/2)

async def moveAttachment(attachmentStr, degrees, speed, stopMode, resetAngle, stallDetect) -> bool:

```
if attachmentStr == "left":
    attachment=leftAttachment
elif attachmentStr == "right":
    attachment=rightAttachment
else:
    print("moveAttachment() - ERROR, illegal attachment name of ",attachmentStr)
    return False

if resetAngle == True: resetAttachment(attachmentStr)

if stopMode == "hold":
    stopMode=Stop.HOLD
elif stopMode == "brake":
    stopMode=Stop.BRAKE
elif stopMode == "coast":
    stopMode=Stop.COAST
else:
    stopMode=Stop.HOLD
```

Motors have different stop modes.

- **HOLD** is used for picking up and carrying items.
- **HOLD** or **BRAKE** is good for lifting a lever or object that the robot doesn't have to carry.
- **COAST** is best for levers you have to push down to reduce stall risk.



Purpose: Move an attachment a specific number of degrees, with a configurable stop mode, and optional stall detection. Degrees are relative to the start position of the attachment.

moveAttachment(attachmentStr, degrees, speed, stopMode, resetAngle, stallDetect) (2/2)

Continued from Previous Slide

```
if (stallDetect == True):
```

```
    await multitask(attachment.run_target(speed, degrees, then=Stop.HOLD),  
                    isStalled(attachment), race=True)
```

```
else:
```

```
    await attachment.run_target(speed, degrees,  
                                then=stopMode, wait=True)
```

```
if debugL2: print("DEBUG-L2 - End moveAttachmentFC() final angle=", topAttachment.angle())
```

```
return True
```

Move the attachment a specific number of degrees while also checking to see if the motor has stalled using the `isStalled()` function. When `race=True` it will run both commands until one finishes then it kills the other one. So if the motor stalls it will kill the `run_target()` function

Move the attachment a specific number of degrees, then stop using `stopMode`, and wait for the command to complete

Purpose: Move an attachment a specific number of degrees, with configurable stop mode, and optional stall detection. Degrees are relative to the start position of the attachment.

driveRobot(dist, speed, accel)

drive robot forward or backward (-negative dist)


```
async def driveRobot(dist,speed, accel) -> bool :  
    if debugL2: print("DEBUG-L2 - driveRobot() dist=",dist," speed=", speed," accel=",accel)
```

configure the speed and acceleration values


```
driveBase.settings(straight_speed=speed,  
                    straight_acceleration=accel)
```

```
await driveBase.straight(dist, then=Stop.HOLD, wait=True)
```

```
if debugL2: print("DEBUG-L2 - driveRobot() done dist travelled=",driveBase.distance())  
return True
```



Set the straight speed and acceleration for the drive motors



Drive forward or backward a specific distance in millimeters, then stop and hold position, and hub will wait till this is done

Purpose: Drive forward or backward a specific speed and acceleration. Direction is based on if distance is negative. Normally a distance negative is backward but this would depend on how the robot is oriented.

`driveRobotAndLift(dist, accel, attachmentStr, degrees, aSpeed, stopMode, resetAngle, stallDetect)`

```
async def driveRobotAndLift(dist,dSpeed,accel,attachmentStr,degrees,aSpeed,resetAngle,stallDetect) -> bool:
```

```
    if attachmentStr == "left":
```

```
        attachment=leftAttachment
```

```
    elif attachmentStr == "right":
```

```
        attachment=rightAttachment
```

```
    else:
```

```
        print("driveRobotAndLift() - illegal attachment name of ",attachmentStr)
```

```
    return False
```

```
    if debugL2: print("DEBUG-L2 - driveRobotAndLift() dist=",dist," dSpeed=",dSpeed," accel=",accel,"  
                    attach=",attachmentStr," degrees=", degrees," aSpeed=",aSpeed," resetAngle=",  
                    resetAngle," stallDetect=",stallDetect)
```

```
    await multitask(driveRobot(dist, dSpeed, accel, False),
```

```
                    moveAttachment(attachmentStr, degrees, aSpeed, stopMode,resetAngle,stallDetect))
```

```
    if debugL2: print("DEBUG-L2 - driveRobotAndLift() done")
```

```
    return True
```

Move forward or backward

Move attachment

Purpose: Purpose: Drive forward or backward a specific speed and acceleration while moving an attachment. Normally a negative distance is backward but this would depend on how the robot is oriented.

turnRobot(dir, turnAccel, turnRate)

turn robot forward or backward (use negative number for
backward if robot facing forwards). if robot is facing
backwards, forward is then the negative number

async def turnRobot(dir, turnAccel, turnRate) -> bool :


if debugL2: print("DEBUG-L2 - turnRobot() dir=", dir, " turnAccel=", turnAccel, " turnRate=", turnRate)

driveBase.settings(turn_acceleration=turnAccel, turn_rate=turnRate)

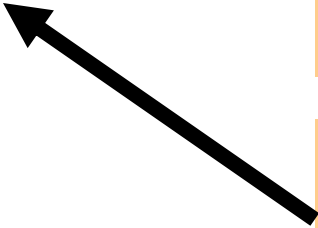
await driveBase.turn(dir, then=Stop.HOLD, wait=True)

if debugL2: print("DEBUG-L2 - turnRobot() done")

return True



Set the turn acceleration and turn rate for the drive motors



Turn the robot to a specific angle, hold position at end of turn, wait until the turn completes.

Purpose: Turn the robot to a specific angle.

```
turnRobotAndLift(dir, turnAccel, turnRate, attachmentStr, degrees,
aSpeed, stopMode, resetAngle, stallDetect)
```

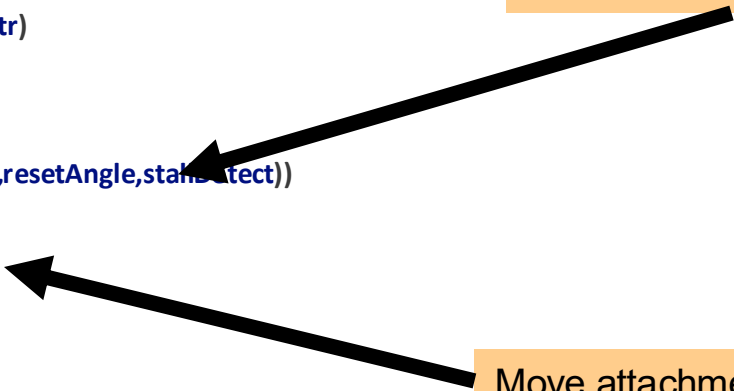
```
async def turnRobotAndLift(dir,turnAccel, turnRate,attachmentStr, ,aSpeed, resetAngle, stallDetect) -> bool:
```

```
    if attachmentStr == "left":
        attachment=leftAttachment
    elif attachmentStr == "right":
        attachment=rightAttachment
    else:
        print("turnRobotAndLift() - illegal attachment name of ",attachmentStr)
        return False

    await multitask(turnRobot(dir,turnAccel, turnRate,False),
        moveAttachment(attachmentStr, degrees, aSpeed, stopMode,resetAngle,stallDetect))

return True
```

Turn robot



Move attachment

Purpose: Turn the robot to a specific angle and move an attachment

`driveRobotInArc(radius, mode, angleOrDist, speed, accel, turnAccel, turnRate)`

`async def driveRobotInArc(radius, mode, angleOrDist, speed, accel, turnAccel, turnRate) -> bool :`

```
driveBase.settings(straight_speed=speed, straight_acceleration=accel,  
                   turn_acceleration=turnAccel, turn_rate=turnRate)
```

```
if mode == "arc":
```

```
    if debugL2: print("DEBUG-L2 - turnRobotInArc() radius=", radius,  
                    " mode=arc degrees=", angleOrDist)
```

```
    await driveBase.arc(radius, angle=angleOrDist, then=Stop.HOLD, wait=True)
```

```
if mode == "distance":
```


```
    if debugL2: print("DEBUG-L2 - turnRobotInArc() radius=", radius,  
                    " mode=distance distance=", angleOrDist)
```

```
    await driveBase.arc(radius, distance=angleOrDist, then=Stop.HOLD, wait=True)
```


```
if debugL2: print("DEBUG-L2 - turnRobotInArc() done")
```

```
return True
```

Drive the robot in arc by specific number of degrees then stop and hold position



Drive the robot in arc by specific number of millimeters then stop and hold position



`driveRobotInArcAndLift(radius,mode, angleOrDist, attachmentStr, degrees, aSpeed, stopMode, resetAngle, stallDetect) (1/2)`


`async def driveRobotInArcAndLift(radius,mode,angleOrDist,attachmentStr,degrees,aSpeed,resetAngle,stallDetect) -> bool:`

```
if attachmentStr == "left":
    attachment=leftAttachment
elif attachmentStr == "right":
    attachment=rightAttachment
else:
    print("driveRobotInArcAndLift() - illegal attachment name of ",attachmentStr)
    return False

if mode == "angle":

    await multitask(driveBase.arc(radius, angle=angleOrDist, then=Stop.HOLD, wait=True),
                    moveAttachment(attachmentStr, degrees, aSpeed, stopMode,resetAngle,stallDetect))
```

Drive robot in an Arc
based on degrees



Move attachment



Purpose: Drive robot in an arc (by degrees or distance) and move an attachment

`driveRobotInArcAndLift(radius, mode, angleOrDist, attachmentStr, degrees, aSpeed, stopMode, resetAngle, stallDetect) (2/2)`

Continued from Previous Slide

```
if mode == "distance":
```

```
    if debugL2: print("DEBUG-L2 - driveRobotInArcAndLift() radius=", radius,
                    " mode=distance distance=", angleOrDist,
                    " attach=", attachmentStr, " degrees=", degrees, " aSpeed=", aSpeed,
                    " resetAngle=", resetAngle, " stallDetect=", stallDetect)
```

```
    await multitask(driveBase.arc(radius, distance=angleOrDist, then=Stop.HOLD, wait=True),
                    moveAttachment(attachmentStr, degrees, aSpeed, stopMode, resetAngle,
                                   stallDetect))
```

```
if debugL2: print("DEBUG-L2 - driveRobotInArcAndLift() done")
return True
```

Drive robot in an Arc
based on distance

Move attachment

Purpose: Drive robot in an arc (by degrees or distance) and move an attachment

isStalled()

detect if a specific attachment is stalled
used with multitask() in moveAttachment functions
async def isStalled(attachment) -> bool :

if debugL2: print("isStalled()", attachment.stalled())

while attachment.stalled() == False:
 await wait(1)

if debugL1: print("stalled")

return attachment.stalled()

-> Bool - This is used to indicate that this function returns a Boolean value (True/False) .

Run a loop checking to see if the attachment is stalled. Wait one millisecond between checks. You might need to adjust this for your robot.

Returns a Boolean value, in reality this will only ever return a True because the function loops until it is stalled.

Purpose: This function is used to detect if an attachment is stalled (stuck) in position. It is called from the `multitask()` function used in other functions above.

resetAttachment (attachmentStr)

```
async def resetAttachment(attachmentStr):
    if attachmentsPresent == False: return
    if attachmentStr == "left":
        attachment=leftAttachment
    elif attachmentStr == "right":
        attachment=rightAttachment
    else:
        print("resetAttachment() - illegal attachment name of ",attachmentStr)
        return False

    if debugL2: print("DEBUG-L2 - Start resetAttachment() attachment= ",attachmentStr,
        " previous angle=",attachment.angle())

    attachment.reset_angle(0)

    if debugL2: print("DEBUG-L2 - End resetAttachment() attachment= ", attachmentStr,
        " new angle=",attachment.angle())
```

Checks to see if a valid attachment name was provided.

Example of a debugging statement

Resets the current angle of the attachment

Purpose: This function is used to reset the current angle of an attachment to a zero position. Primarily used in functions that need to make sure the angle provided is from a zero position not relative to previous angle.

initializeRobotForTask()

```
async def initializeRobotForTask():  
    if debugL1: print(" Initializing Robot for Task")  
  
    while hub.imu.ready() != True:  
        await wait(250)  
  
    if attachmentsPresent:  
        await resetAttachment("left")  
        await resetAttachment("right")  
  
    driveBase.use_gyro(False)  
    driveBase.reset(angle=0,distance=0)  
    hub.imu.reset_heading(0)  
    driveBase.use_gyro(True)  
  
    if debugL1: print(" Completed Robot Initialization")
```

Is the Hub's Inertial Measurement Unit stable.

Reset attachment to zero position

Disable Gyro temporarily

Reset driveBase and IMU to a known good position. Gyro must be disabled for the imu to do this

Enable Gyro

Purpose: This function is used to set the robot to a zero position and ensure everything that needs to be enabled or disabled at the beginning of a mission.

stopEverything()

```
# stops the motors and disables the gyroscope (if  
# you lift it off the table) this should be run at  
# the end of every task in the task and in menu  
# after each task
```

```
async def stopEverything():
```

```
    if debugL1: print("Stop All Motors & Gyro")
```

```
    driveBase.use_gyro(False)
```

```
    await wait(100)
```

```
    leftMotor.stop()
```

```
    await wait(100)
```

```
    rightMotor.stop()
```

```
    await wait(100)
```

```
    if attachmentsPresent:
```

```
        leftAttachment.stop()
```

```
        rightAttachment.stop()
```

```
    await wait(100)
```

```
    if debugL1: print("All motors should be stopped")
```

Disable the gyro so that if you pick up the robot it doesn't go nuts

Stop the drive motors

Stop the attachment motors

Purpose: Shuts down the motors and gyroscope so that robot can be safely picked and moved to a new position.

Using RobotLib

- Put the following line (updated for the correct revision) at the top of your `hub_menu()` and task/mission files. It imports the robot library, Pybricks libraries, and various variables that you will use in your programs
 - `import pybricks_class_library_rev4 as RobotLib`
- To use/access RobotLib or Pybricks variables and functions you have to use the `RobotLib.` prefix. Some examples:
 - `RobotLib.driveBase.settings(straight_speed=400, straight_acceleration=950)`
 - `await RobotLib.wait(1000)`
 - `await RobotLib.driveBase.straight(-250)`
 - `await RobotLib.rightAttachment.run_angle(150, 210)`

Four Pybricks library functions

 - `await RobotLib.driveRobot(-105, 400, 100)`
 - `await RobotLib.turnRobot(76, 200, 200)`

Two robot library functions
- You can mix direct Pybricks commands with RobotLib commands if needed

Examples of Using the Robot Library

Robot Lib Example (1/3)

```
# Robot Lib Demonstration
#
# task revision number
t1_revision="2"
#
import pybricks_class_library_rev4 as RobotLib
```

```
async def pybricks_class_task4():
    print ("Robot Library Demonstration, Suibmerged Demo rev=",t1_revision)
    await RobotLib.initializeRobotForTask()
```

```
    await RobotLib.wait(1000)
    if RobotLib.debugL1: print(" get and place coral tree")
    await RobotLib.driveRobot(285,750,500)
    await RobotLib.driveRobotAndLift(160,100,100,"left",1750,1000,"hold",False,True)
    await RobotLib.turnRobot(5,100,100)
    await RobotLib.driveRobotAndLift(-230,300,150,"left",1100,1000,"hold",False,True)
    await RobotLib.turnRobotAndLift(25,200,100,"left",1850,1000,"hold",False,True)
```

Imports the robot library which include the Pybricks libraries as **RobotLib**.

RobotLib is used to reference those variables and functions in your code

Example of debug statement

All library functions, variables, and Pybricks functions need to be prefixed with "**RobotLib.**" including the period after Lib

Robot Lib Example (2/3)

Continued from Previous Slide

```
if RobotLib.debugL1: print(" smash coral nursery")
await RobotLib.driveRobot(580,850,500)
await RobotLib.turnRobot(-15,200,200)
await RobotLib.moveAttachment("right",-175,1000,"hold",False,True)
await RobotLib.moveAttachment("right",0,1000,"hold",False,True)
await RobotLib.driveRobot(-105,400,100)
await RobotLib.turnRobot(76,200,200)
```

```
if RobotLib.debugL1: print(" lift mast")
await RobotLib.driveRobotAndLift(-80,100,100,"right",-170,300,"hold",False,True)
# drive slowly to light mast
await RobotLib.turnRobot(5,100,100)
await RobotLib.driveRobot(260,200,400)
# drive away from ship and lift robot arm out of the way
await RobotLib.driveRobot(-210,400,1000)
await RobotLib.moveAttachment("right",0,100,"hold",False,True)
await RobotLib.driveRobot(100,400,1000)
```



Robot Lib Example (3/3)

Continued from Previous Slide

```
if RobotLib.debugL1: print(" goto shark")  
await RobotLib.turnRobot(-142,200,200)  
await RobotLib.driveRobot(300,700,500)
```

```
if RobotLib.debugL1: print(" move away from shark")  
await RobotLib.driveRobot(-75,500,100)
```

```
await RobotLib.stopEverything()  
print("Task 1 - Completed")
```



run `stopEverything()`
to make sure robot is safe
to pick at end of task

Pybricks Library Menu (1/4)

Pybricks Class Menu

menu revision number

menu_revision="4"

import necessary libraries and functions

from pybricks_class_library_rev4 import *

print("Robot Menu, rev#", menu_revision)

print("Load Task 1")

from pybricks_class_task1_rev4 import pybricks_class_task1

print("Load Task 2")

from pybricks_class_task2_rev4 import pybricks_class_task2

print("Load Task 3")

from pybricks_class_task3_rev4 import pybricks_class_task3

print("Load Task 4")

from pybricks_class_task4_rev1 import pybricks_class_task4

Pybricks Library Menu (2/4)

```
print("Initialize Robot")
run_task(initializeRobot())

print("Running Hub Menu")
while True:
    selected = hub_menu("M", "1", "2", "3", "4", "X")
    try:
        if selected == "M":
            # does nothing. used to show menu is active
            break
        elif selected == "1":
            print("Task 1 Selected")
            wait(1000)
            run_task(pybricks_class_task1())
            run_task(stopEverything())
```

Pybricks Library Menu (3/4)

```
elif selected == "2":
```

```
    print("Task 2 Selected")
```

```
    wait(1000)
```

```
    run_task(pybricks_class_task2())
```

```
    run_task(stopEverything())
```

```
elif selected == "3":
```

```
    print("Task 3 Selected")
```

```
    wait(1000)
```

```
    run_task(pybricks_class_task3())
```

```
    run_task(stopEverything())
```

```
elif selected == "4":
```

```
    print("Task 4 - Robot Library Demo Selected")
```

```
    wait(1000)
```

```
    run_task(pybricks_class_task4())
```

```
    run_task(stopEverything())
```

Pybricks Library Menu (4/4)

```
elif selected == "X":  
    print("End of Menu")  
    break  
except BaseException as menuException:  
    print("Stop was Pressed or a Critical Error Occurred.")  
    print(menuException)  
    run_task(stopEverything())  
    break
```

Demonstration Robot Run for Submerged



Section 7 - Robot Game

Robot Game: Introduction

- Robot Game makes up only 25% of the team's FLL tournament total score
- The teams behavior during the robot game contributes to their Core Values score (which is another 25% of their score)
- A team does not need to win or even place in the top 5 in the robot game to move from a qualifier to the State tournament
- Teams focusing solely on the robot game will not advance out the qualifier or tournament

Robot Game: Rulebook & Videos

- All team members must read the Robot Game Rulebook (RGR)
- Failure to understand the scoring rules will result in unnecessary frustration in developing attachments and programming the robot
- **Do NOT read into the wording.** Take wording as literal.
 - If it doesn't say you can't, you likely can do it
 - However, breaking or dislodging a mission model may result in no points for the mission
- Watch the Model Setup and Activation Mechanism video
 - Posted to the same FLL website as the assembly instructions
- Watch Robot Game Missions video (usually on YouTube)
 - There are applications for Mac and Windows that will let you download the video so you can watch when an Internet connection is not available. Youtube is blocked at many schools.
 - For Mac, the HomeBrew yt-dlp app is available at <https://formulae.brew.sh/formula/yt-dlp>
 - For Windows/Linux/Mac, Anaconda youtube-dl app is available at <https://anaconda.org/conda-forge/youtube-dl>

Robot Game: Missions vs. Tasks (1 of 2)

- Robot game consists of solving missions
- Missions usually are focused on a single mission model
 - However, there are usually a couple of missions that take pieces from one or more mission models (or locations on the board) and move them to another mission model or location
- A common approach to the robot game is combining multiple missions (or parts of missions) into tasks
 - On Spike Prime robots, the hub menu is then used to run specific tasks
- Tasks should be optimized for the limited time available **(2 min 30 secs!)**
 - Returning to a home base is expensive and should be visited sparingly to swap attachments and add/remove mission model components needed for other tasks
 - Consider grouping tasks based on the action (e.g., push, pull, lift, pickup, flip) required for the task which usually means which attachments are connected to the robot

Robot Game: Missions vs. Tasks (2 of 2)

- Teams should be careful about assigning too many missions to a single task
 - If the task fails, it is usually not practical to grab the robot off the table (and thus losing a precision token) and rerun the task
 - Programs tend to expect objects and/or models to be in specific positions
 - Rerunning the task will likely fail, because the positions of the models are different
 - This could result in another precision token being used if the robot gets stuck or is off course
- Testing with multiple game boards is highly recommended
 - Competition boards are not perfect and do have imperfections including bumps, rises, and uneven surfaces that can throw off the robot - this is also why using the gyroscope is important
- Teams should consider building positioning rigs to ensure the robot is always started in the right location for each task

Robot Game: Approach to Missions

- There usually is more than one way to solve a mission
 - In most cases it is the end state that matters
- Many missions have an “obvious” solution but that might not be the “best or optimal” solution
 - Thinking “out of the box” should be encouraged! Many “wild-n-crazy” ideas might actually be the best ones!
- Look at the mission model from all angles
 - Are there other ways to flip, turn, or lift the mission?
 - Would a pull work better than a push?
- Rarely does FLL dictate how the result must be obtained
 - Exceptions are usually related to mission models that have an interaction with the other table/team
 - There is usually a mission model that will require some “interaction” with the other table/team

Robot Game: Team Organization

- The robot game is the portion of FLL where assigning roles is most critical
- The FLL teams have found there are typically three roles during the robot game
 - **Driver** (1 per side) - The team member responsible for initiating the robot for each task
 - **Rigger** (1 per side) - The team member responsible for assisting the driver with the robot
 - Focused on securing the previous mission's attachment and providing to driver the next attachment and any mission pieces needed for the next task
 - **Signer** (1 per team) - The team member responsible for signing the score sheet
 - The signer should be able to "quote" the RGR
- Teams with 5+ or members typically have two crews
 - Calling them the A and B teams might not be the best approach, try colors instead
 - Reality is there are team members that are more proficient at successfully running the Robot Game

Robot Game: “The Grand Unification”

- Many teams approach the robot game by having small groups within the team develop solutions for each mission
- Don't wait until the last minute to integrate those solutions into a set of tasks
- Teams should either consider:
 - Starting to integrate individual mission solutions into larger tasks at least 4 weeks out from competition
 - Have half the team working on integrating, testing, and turning completed missions while the other half develops code for new missions
- Remember the robot game lasts 2.5 minutes. You do not have to return to home at the end!
- It takes a lot of time to work out integration issues/bugs in the code and attachments
- It takes lots of rehearsals to get the team to consistently “run” the robot without errors
- Use a single robot for your integration testing
 - Switching between robots can increase testing complexity as the robots do have different “personalities” based on manufacturing differences and differences in “wear-n-tear” over the years
 - Once everything is working on one robot, duplicate the code and tweak (if necessary, usually is) for the second backup robot

Robot Game: At the Competition Table

- Before each game table match, it is critical for the students to review the board and ensure the table is setup correctly
 - Verify missions are not "broken", attached to the dual-lock, and are configured correctly
 - It is the responsibility of each team to verify the configuration of the board
 - Use the **signer** role as the person that notifies the referee the team is ok with the state of the board
 - There are no "do-overs" if the board is wrong once the match has started
 - **The coach is not involved in this activity!**
- Team needs to greet the other team and wish them well and thank the referee for volunteering
- Laptops/Tablets are NOT allowed at the robot game competition table
 - All tasks/missions must be loaded on the robot, the hub menu is used to change between tasks/missions
 - Remote controllers for the robot are not allowed

Parts....I need Parts!



- The minimum viable Lego kits you need are:
 - LEGO® Education SPIKE™ Prime Set, 45678
 - LEGO® Education SPIKE™ Prime Expansion Set, 45681
 - NOTE: Older Lego robots including EV3 and NXT are legal. Robot Inventor is the same hub as SPIKE Prime
- However, for a competitive robot most teams need additional Technic pieces
 - If you only have 2 large and 2 medium motors, use the large motors for the attachments - you need the torque!
- We have a extensive parts list and locations on where to buy them in our docs
 - Be careful about buying parts on Amazon and Ebay - not everything they sell are authentic parts and officially only Lego parts are allowed in competition

Robot Design Presentation

- Team is required to do a presentation on the design of the robot and attachments
- It is recommended that there should be some visual (e.g., a trifold or poster) that shows which missions use which attachments
 - Teams need to discuss why/how they selected the missions they did
 - The task/mission approach should be discussed, the attachment(s) used
 - Should provide a few of the coding examples
- Team **must** clearly discuss how they were involved in the robot and attachment design, build, testing, refinement, and integration activities
 - Use of previous season robots and attachments are ok, but the team needs to explain how they improved on it during the season to address the season's unique challenges
 - Consider using Lego's free BrickLink Studio "CAD Program" to document the design of the Robot and Attachments

Conclusion

Wait.... But I don't know Python

- There A LOT of free resources on the Internet to learn python
 - <https://www.w3schools.com/python>
 - <https://www.tutorialspoint.com/python>
 - www.python.org
 - docs.pybricks.com
- This Presentation, my Python for FLL presentation, and the code examples are available at:
 - https://github.com/bcfletcher/lego_projects/tree/main/FLL/2025_MD_FLL_Coaches_Conf

Summary

- This was a introduction into Pybricks
- It covered the key concepts and Pybricks APIs that teams will need to be successful using Pybricks
- Pybricks has excellent documentation both within the Application and the website.
- Pybrick's support website has lots of good discussions and examples
 - <https://github.com/pybricks/support>

Xbox Controller Example

Xbox Controller Example

- This is example of how to use an Xbox Controller with a Lego Spike Prime Robot
- It does not use the robot library
- This was primarily an experiment in how to use the Xbox Controller to get angles and distances to objects on an FLL Course
- It was inspired by a presentation from Julian Huss and Monongahela Cryptid Cooperative at the 2025 First Chesapeake Mentor Conference
 - <https://github.com/MonongahelaCryptidCooperative/FLL-Block-2024-2025/>

Xbox Example (1/5)

```
from pybricks.hubs import PrimeHub
from pybricks.iodevices import XboxController
from pybricks.parameters import Button, Direction, Port, Stop
from pybricks.pupdevices import Motor
from pybricks.robotics import DriveBase
from pybricks.tools import wait
```

Set up all devices.

```
prime_hub = PrimeHub()
right_motor = Motor(Port.A, Direction.CLOCKWISE)
left_motor = Motor(Port.B, Direction.COUNTERCLOCKWISE)
attachment_motor = Motor(Port.C, Direction.CLOCKWISE)
drive_base = DriveBase(left_motor, right_motor, 62.4, 204)
controller = XboxController()
```

Initialize variables.

```
mySpeed = 950
```

Xbox Example (2/5)

```
def xbox_controller_test():
    global mySpeed
    print('Connected')
    drive_base.use_gyro(False)
    drive_base.reset(0, 0)
    drive_base.settings(straight_speed=mySpeed)
    drive_base.settings(straight_acceleration=750)
    while not Button.X in controller.buttons.pressed():
        if Button.UP in controller.buttons.pressed():
            print('forward')
            drive_base.straight(10, then=Stop.NONE)
            wait(100)
        elif Button.DOWN in controller.buttons.pressed():
            print('backward')
            drive_base.straight(-10, then=Stop.NONE)
            wait(100)
        elif Button.LEFT in controller.buttons.pressed():
            print('left', drive_base.angle())
            drive_base.turn(-1, then=Stop.NONE)
            wait(10)
```


Xbox Example (3/5)

```
elif Button.RIGHT in controller.buttons.pressed():
    print('right', drive_base.angle())
    drive_base.turn(1, then=Stop.NONE)
    wait(10)
elif Button.A in controller.buttons.pressed():
    drive_base.stop()
    wait(100)
elif Button.B in controller.buttons.pressed():
    print('B_dist=', drive_base.distance())
    wait(100)
elif Button.Y in controller.buttons.pressed():
    print('Y_angle=', drive_base.angle(), ' hub_heading=', prime_hub.imu.heading())
    wait(100)
elif Button.RB in controller.buttons.pressed():
    attachment_motor.run_angle(900, -190, Stop.COAST)
    wait(100)
```

Xbox Example (4/5)

```
elif Button.LB in controller.buttons.pressed():
    attachment_motor.run_angle(900, 190, Stop.COAST)
    wait(100)
elif Button.MENU in controller.buttons.pressed():
    print('Dist/Angle Reset')
    drive_base.reset(0, 0)
    prime_hub.imu.reset_heading(0)
    drive_base.settings(straight_speed=10)
    mySpeed = 10
    wait(100)
elif Button.VIEW in controller.buttons.pressed():
    mySpeed = mySpeed + 10
    print('Up Speed=', mySpeed)
    if mySpeed > 960: mySpeed = 950
    drive_base.settings(straight_speed=mySpeed)
    wait(100)
```

Xbox Example (5/5)

```
elif Button.UPLOAD in controller.buttons.pressed():  
    mySpeed = mySpeed - 10  
    print('Down Speed=', mySpeed)  
    if mySpeed <= 10: mySpeed = 10  
    drive_base.settings(straight_speed=mySpeed)  
    wait(100)  
else:  
    wait(100)  
    print('Program Ended')  
    drive_base.stop()  
    attachment_motor.stop()  
    drive_base.use_gyro(False)
```