Python with Robots

Use the CodeBot to build real-world projects with code.

Mission 1 - Welcome

Welcome to the CodeSpace Development Environment!

A virtual world for exploring robotics with code.

We're glad you're here!

You are about to experience a powerful learning and coding environment:

- Learn to code in **Python** by completing challenging **Missions**.
- Test your real-world programs in *simulation* or on a *physical* device.

Ready to begin your first *Mission?*

- Click the **NEXT** button...

Objective 1 - Mission Objectives

Objectives

Each Mission contains a series of **Objectives**. You're now reading an *Objective Panel.*

- Objectives are numbered on the ***Mission Bar*** to the right.
- Click the **number** to show or hide the Objective Panel.
- Use the icons at the *top* of the Mission Bar to choose from available *Missions* and *Packs.*

The *goals* to complete the Objective are below:

Goal:

- Click the **1** on the *Mission Bar* to close the Objective Panel →
  - Then click **1** *again* to bring it back!

Solution:

N/A

Objective 2 - Text Editor

Text Editor

On the left side of your screen is the **text editor**.

- You'll be typing in **Python code** here!
  - That's how you'll control your *physical* or *virtual* device.

Go ahead and *type something in!*

Goal:

- Complete this Objective by making any *change* in the **text editor**.

Solution:

N/A

***Objective 3* - Tool Box**

Your Coding Toolbox

As you work through each mission you'll be adding concepts to your toolbox.

- It's an important **reference** you will need in later missions!
- *And* when you are coding and debugging your own ***remixes.***

Collect 'em *ALL!*

When you see a tool, CLICK on it!

- You won't have anything in your toolbox unless you put it there.

Access Your Tools

You can always open up your toolbox later for reference.

- Just click the at the right side of the window.

Goal:

- Click the tool text above to open the Toolbox and then close the Toolbox.

Tools Found: Debugging

Solution:

N/A

***Objective 4* - Simulation Controls**

Simulation Controls

Below the 3D view is your *Simulation Toolbar*.

- There are controls to select a 3D *environment*.
- You can also control the *Camera* in the 3D scene, and more!
  - *This is a **virtual** camera for zooming around inside the sim, not your webcam!*
- You can manage with a trackpad, but a *mouse* is highly recommended for 3D navigation.

Click on the **Camera** menu below.

- Select **Help**
- *Click the inside the **Camera Help** window to close it.*

Want to *hide* these instructions?

- Click the at the upper-right corner.
- You can always bring an *Objective* back by clicking its number on the right side.
- Or you can *maximize* it by clicking

Goals:

- **Open** and **close** the *Camera Help*.
- **Rotate** the camera view around the *virtual device* in the 3D scene!

Solution:

N/A

***Quiz 1* - Your First Mission Quiz**

Question 1: Are you ready to learn some Python coding with your **physical** device?

✓ Yes. This is simple!

✗ I don't think I can.

✗ It looks too complicated.

Question 2: Select the two things you learned in this mission.

✓ How to open an objective

✓ How to move the camera

✗ How to run a half marathon

✗ How to control the weather

Mission 1 Complete

Welcome to CodeSpace!

You've completed your first ***Mission.***

You can always click the **Mission Select** icon at the upper right side of the window to go back to previous Missions.

You've learned the basics of *Missions* and *Objectives.*

- Now it's time to get to know your device!

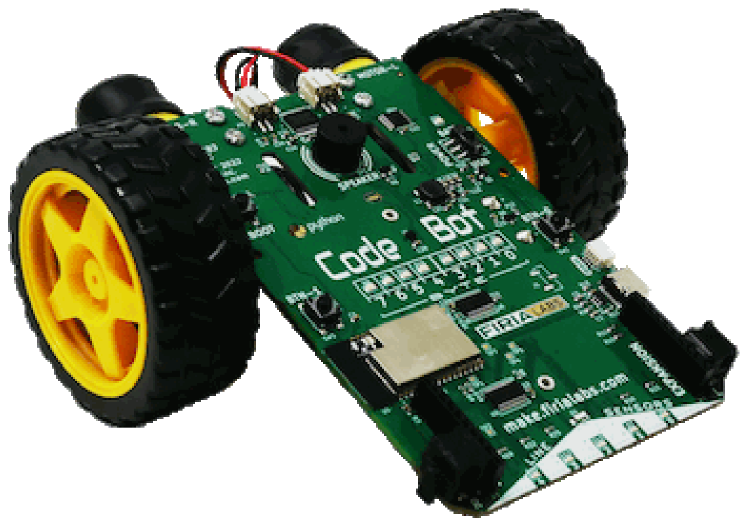
Mission 2 - Introducing CodeBot

CodeBot is a computer on wheels with lots of sensors and controls built-in. You will be writing **Python code** to bring this hardware to life!

There are quite a lot of hardware 🔧peripherals on CodeBot, including:

Outputs:

- LED lights
- Speaker
- Motors
- Infrared transmitters
- Expansion connectors for external device outputs
- USB data and filesystem output



Inputs:

- Line sensors
- Proximity sensors
- Motion and orientation sensor (accelerometer)
- Temperature sensor
- Wheel rotation sensors
- Pushbuttons
- Infrared receivers
- Expansion connectors for external device inputs
- USB data and filesystem input

Other:

- WiFi transceiver (CB3 Only)

One of the best things about CodeBot is that *all of that hardware* is completely controlled by code that you write. That means it's up to **you** to unlock the true potential of your robot.

Objective 1 - Motors

Motors - Programmable Electric Engines

CodeBot's 🔧motors power the *wheels* that move it around.

- They convert *electric power* to *mechanical rotation.*
- The picture at right shows a motor without its protective black cover, and with the gearbox open.

You'll soon be controlling those motors with Python code!

Locale the motors in the 3D View, and click on one of them...

> *To hide these instructions click the ✕ at the upper-right corner or press **CLOSE***

Hint:

- You *may* need to rotate your camera to the **back** of the 'bot!

Goal:

- Click one of the Motors in the 3D view

Tools Found: Motors

Solution:

N/A

***Objective 2* - LED Lights**

LEDs - Lighting the Way

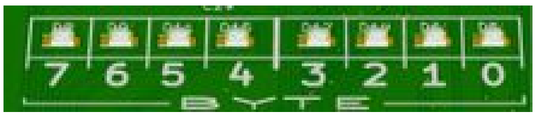
"Light Emitting Diodes" are tiny and efficient electronic components that produce light.

- There are 17 *visible light* 🔧LEDs on CodeBot
- ...and there are *8 more* LEDs that emit *infrared* light only robots can see ;-)

Like everything on CodeBot, they pretty much do nothing...

- Until **YOU** write some code to control them!
- *You'll be doing that in the next mission.*

Up close the LEDs look like little clear boxes:



Zoom In!

Use your mouse and the Camera controls to **zoom-in** for a closer look at the LEDs.

Goal:

- Click an LED on your *virtual CodeBot* in the 3d View!

Tools Found: LED

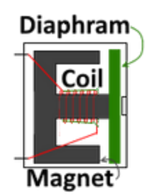
Solution:

N/A

***Objective 3* - Speaker**

Speaker - Make some Noise!

...or, make *beautiful* music. It's your choice.



- There's a real 🔧speaker aboard your 'bot.
  - Inside this little black cylinder is an electromagnet with a permanent magnet to pull against.
  - Hey, that's basically what's going on in the motors too!

Goal:

- Click on the Speaker in the 3D View

Tools Found: Speaker

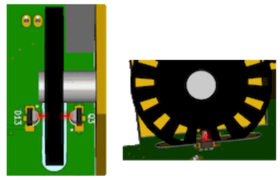
Solution:

N/A

***Objective 4* - Wheel Encoders**

Wheel Encoders

Your code can control the *power* applied to the motors, but to know exactly how far the wheels have turned you'll need to *sense rotation*. That's the job of these 🔧Encoders



View from beneath CodeBot

As the encoder disc rotates, an invisible IR (infrared) light beam passes through its slots. Your code can count the pulses of light to see how far the wheel has rotated.

Hint:

- The 🔧wheel encoders *can* be seen from the **top** of the 'bot.
- If you're having a difficult time, try looking *underneath* it!

Goal:

- Click on one of the black *Encoder Discs* in the 3D View

Tools Found: Wheel Encoders

Solution:

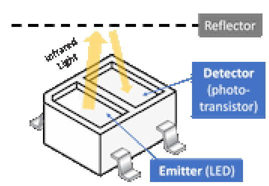
N/A

***Objective 5* - Pushbuttons**

Pushbuttons, Line Sensors, Proximity Sensors, Accelerometer, and *more...*

Okay, last objective for this "Intro" Mission... Then we start coding!

- As you've seen, there's a lot happening on your CodeBot.
- You'll explore all of it by writing Python code to complete Missions.
- ...and you're gonna ***need*** all those capabilities for the *challenges* we have in store!



Goal:

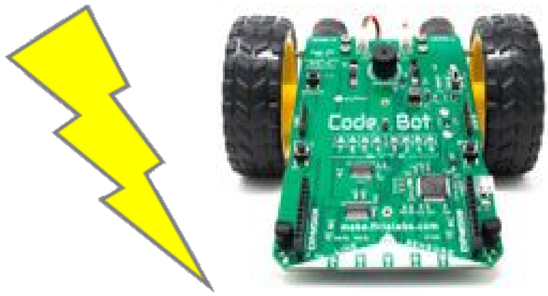
- Complete this objective by clicking on a 🔧CodeBot Button in the 3D View.
  - *(There are 3 of them to choose from!)*

Tools Found: Buttons

Solution:

N/A

***Quiz 1* - Don't Zap Your Bot!**



Static electricity is a charge that can build up when you walk across carpet in socks or take off a wool sweater. It causes the jolt and spark that happens sometimes when touching something grounded, like a faucet or lightswitch.

Hold your CodeBot by its **edges**, being gentle with the LEDs and other electronic components. They're all exposed on the board as with most other **Maker** computers, so you can *really* get to know them. More on that in the next few pages...

Especially when the air is very dry (cold or arid climates) it's good practice to touch some grounded metal (desk, doorknob) before handling the CodeBot to avoid damaging its sensitive components with static electric discharge.

Question 1: What should you do *before* handling your CodeBot?

✓ Touch some grounded metal

✗ Clean it with wet wipes

✗ Jumping jacks

***Objective 6* - Connect the USB**

Now, use the USB cable to connect the *CodeBot* to your computer.

> Caution: *Note*
>
> You may see a *window* pop-up when you plug in CodeBot.
>
> Feel free to close this window, you won't need it for CodeSpace.

Connecting the USB cable does two things:

1. It lets your computer communicate with the ***CodeBot***.
2. It provides 5 volt DC power to the CodeBot.

Make sure your USB cable is connected now!!

Hint:

-

See the port the cable is plugged into?

- Click that on the *virtual* 'bot!
- *You may need to rotate the camera!*

Goal:

- Click on the USB connection port in the 3D Scene.

Tools Found: USB

Solution:

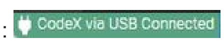
N/A

Objective 7 **- Link to CodeSpace**

The CodeBot must be linked to your browser before it can be used with CodeSpace.

Connection Steps

1. Make sure the USB cable is connected *both* to your PC and the CodeBot.
2. Click the **red** bar **below the code editor** to open the USB connection dialog.
   - The *connection bar* looks like this: 

   - The bar should look like *this* if your device is already connected: 

3. Select "**CodeBot**" from the *device list* that pops up.
   - See the video at right for an example.
4. Click the **Connect** button in the pop up.

Goal:

- Link your CodeBot to CodeSpace.
  - **Hint:** Make sure only one CodeX or CodeBot is connected.

Solution:

N/A

Objective 8 **- Save the Code!**

Time to create a file!

When you type code into the **text editor** panel on the left, it is automatically saved to your personal file-system in CodeSpace cloud!

Code is stored in files on a computer just like any other document.

- Each code file should have a **name** that states its purpose.

You should make a new file for each objective. Here's how:

1. Click the **File** menu button above the code editor.
2. Click *New File...*
3. Type in the name you'd like to give your new file.
4. Click the **Create** button.

Your new file should open in your code editor!!

Goal:

- Create a new file named: `LightsOn`
    - If this file is already in your file system go ahead and use the *New File...* button anyway!
    - Double check your capitalization!!

Solution:

N/A

Objective 9 **- The CodeTrek**

Check out the CodeTrek!!

The CodeTrek is a **CodeSpace** tool that gives you:

- A starting point for your program.
- Detailed information about lines of code you need to write.
- Explanations of coding topics.
- Holes (TODOs) for you to fill in on your own!

TODOs

A `# TODO:` is an instruction in a code comment.

- A 🔧comment is code that *doesn't get run*, you'll learn about them in-depth later.
- TODOs are used in the real world all the time!
    - They tell you to come back here because there is still work **TO DO**!!
    - Most code editors recognize `# TODO` and highlight it in your code!!

Click the **CodeTrek** button below to learn more about the code for an objective.

CodeTrek:

```
from botcore import leds
```

The CodeTrek will give you information about lines of code or give you more knowledge on a topic.

```
# TODO: Light USER LED 0
```

A `# TODO:` tells you to come back here because there is still work **TO DO**!!

- TODOs are used in the real world all the time!
    - Most code editors recognize `# TODO` and highlight the line in your code!!

Hint:

- There are **two** *steps* in the CodeTrek.
- Make sure you see them *both* by hitting the "NEXT" button!

Goal:

- Open the CodeTrek to learn about your code with the button.

Tools Found: Comments

Solution:

N/A

Objective 10 **- Lights On!**

Now it's time for **you** to type in some code!

Caution

- Capitalization matters! Your code is **case sensitive** *("Number" is **not** the same as "number"!)*.
- Punctuation is important!

(*Relax*, you're not going to break anything, but programming languages are very strict!)

Check the 'Trek!

Run It!

Watch for the **LEDs** in the center of CodeBot.

- They're labeled **BYTE** and numbered 7-6-5-4-3-2-1-0.
- These ***8 LEDs*** are dedicated to the **USER**.
  - Use them to display general status about anything!
- The other LEDs on CodeBot are located near peripherals.
  - Your code can use those LEDs to indicate sensor activity!

Try Your Skills: *Illuminati Confirmed*

CodeTrek:

```
from botcore import leds
```

The first line tells Python to import the **leds** object from the **botcore** library.

- **botcore** provides direct access to CodeBot's hardware!

```
leds.user_num(0, True)
```

The *second* line uses **leds** from *botcore!*

The `user_num()` function controls the red "BYTE" LEDs by *number*.

- Each LED can be ON or OFF, represented by boolean `True` or `False` values.
- There are **8** *user* LEDs, numbered 0-7.

Goals:

- `import` the `leds` object from the `botcore` library.
- **Light** the *USER LED* at index `0`.

Tools Found: Punctuation, Syntax Highlighting, Underscore, CPU and Peripherals, import, bool

Solution:

```
from botcore import leds #@1
leds.user_num(0, True) #@2
```

<u>Mission 2 Complete</u>

You've completed another mission!

...and you're at the start of a fantastic **adventure**. Your journey will take you to greater heights - more missions are ahead to *challenge* and *amaze* you!

Mission 3 **- Time and Motion**

In this project you'll get CodeBot *moving*!

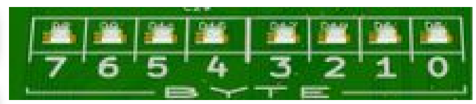
- When you're writing code for CodeBot you're doing *Physical Computing*. From **cars** to **stage lights**, code is at the heart of many things that get you moving!
- You'll use Python's **time** library to precisely control the timing of your bot's actions.

    Get your motors running!

Project Goals:

- Flash CodeBot's LEDs in a controlled sequence.
- Make a *Light Show* using all the LEDs.
- Learn how to use the *CodeSpace* ***debugger*** .
- Power up the **motors** to *move* and *rotate* your 'bot.
- Write code to drive in a specified pattern.
- Use pushbutton inputs to control the action.

Objective 1 **- LED Sequencer**



Create a New File!

Run It!

What do you notice when you run this code:

- Can you see each LED turn ON in sequence?
- Do you think they come on at *exactly the same time*?
- ...Or, do you think they activate one-at-a-time, but *really quickly*?

CodeTrek:

```
from botcore import leds
leds.user_num(0, True)
leds.user_num(1, True)
leds.user_num(2, True)
leds.user_num(3, True)
```

Goal:

- Light up *USER LEDS* `0` through `4` in *sequence*.

Tools Found: Editor Shortcuts

Solution:

```
from botcore import leds
leds.user_num(0, True)
leds.user_num(1, True)
leds.user_num(2, True)
leds.user_num(3, True)
```

Objective 2 **- The Debugger**

Inside the Mind of the Computer!

Computers are fast. Even a small 🔧CPU like CodeBot's can execute *millions* of operations per second!

The **CodeSpace debugger** lets you **Step** your program *one line at a time*, at your own speed, so you can understand *exactly* what the computer is doing and debug your code.

Note: Each line of code runs ***after* the** Step **button is clicked.**

Concept: *Stepping*

You can execute the code **one line at a time** by using the **STEP** button.

This is a *very* powerful tool for debugging your code. Be sure to use it whenever you need to understand more clearly what the code is doing!

Debug: *Try stepping through your code!*

Rather than pressing the **RUN** button, you can press **DEBUG** and have the computer *step* through your code.

Try it yourself and you'll see that **each LED *does* light one-at-a-time!**

1. Press the **DEBUG** button to re-load your program and wait at the first line.
2. Keep pressing **STEP IN** to execute each line of code in turn.
3. The highlighted line executes **after** you click **STEP IN**.
4. Then the *next* line of code is highlighted, waiting and ready to go...
5. Check CodeBot's LEDs *after* each STEP!

Goal:

- Step into your code with the CodeSpace Debugger.
- First click **DEBUG** then use the **STEP IN** button to *step* through your code.

Tools Found: CPU and Peripherals, Debugging

Solution:

```
from botcore import leds
leds.user_num(0, True)
leds.user_num(1, True)
leds.user_num(2, True)
leds.user_num(3, True)
```

Objective 3 - Slow it Down

When you *step slowly* through the code, the LEDs light in sequence. So you just need a way to delay the computer a little after it shows each Image.

```
from time import sleep
sleep(1.0)
```

The `sleep(1.0)` above causes CodeBot to delay for 1 second before continuing.

- You can try different times, like 0.5 or 3.14 seconds!
- Notice you have to import **sleep** from Python's **time** library.
    - Do that *just once* at the top of your program.

Check the 'Trek!

Add a line with `sleep(1.0)` on the *next* line of code **after** each `leds.user_num()`.

Run It!

Watch CodeBot's LEDs when you press the RUN button.

CodeTrek:

```
from botcore import leds
# TODO: Import the sleep object
```

Import `sleep` *from* the `time` library!

```
from time import sleep
```

```

leds.user_num(0, True)
sleep(1.0)
```

Your code will **stop** here for `1.0` second!

```
leds.user_num(1, True)
sleep(1.0)
leds.user_num(2, True)
sleep(1.0)
leds.user_num(3, True)
```

Goal:

- **Wait** `1` second *between* each `leds.user_num()` call using `sleep(1.0)`

Tools Found: Timing, import

Solution:

```
from botcore import leds
from time import sleep

leds.user_num(0, True)
sleep(1.0)
leds.user_num(1, True)
sleep(1.0)
leds.user_num(2, True)
sleep(1.0)
leds.user_num(3, True)
```

Objective 4 - Variable Speed!

It would be fun to play with some different delay times to change the **speed** of those LEDs. Right now the number `1` appears *three* times in the code, and **all** must be changed to adjust the delay between LEDs lighting up.

- Wouldn't it be nice to set the delay in ***one place***?

Instead of repeating a *literal number* like `1` in your code, you can use a `name` instead. Read on to learn how much *easier* this makes it to **change** your delay!

Concept: *Variables!*

A variable is a *name* to which you assign some *data*. The *data* could be a number, a `True` or `False` boolean value, or any other type of information your program uses.

Variables must be **defined** like this before they are used:

```
delay = 1.0
```

Check the 'Trek!

CodeTrek:

```
from botcore import leds
from time import sleep

delay = # TODO: Set delay to 1.0
```

Declare your **variable**!

Set `delay` to `1.0` with the line:

`delay = 1.0`!

```

leds.user_num(0, True)
sleep(delay)
```

When `delay` is equal to `1.0`, `sleep(delay)` is equal to `sleep(1.0)`!

```
leds.user_num(1, True)
sleep(delay)
leds.user_num(2, True)
sleep(delay)
leds.user_num(3, True)
```

Goals:

- **Declare** a **variable** named `delay` with a value of `1.0`.
- **Wait** `delay` second(s) *between* each `leds.user_num()` call using `sleep(delay)`

Tools Found: Variables, bool

Solution:

```
from botcore import leds
from time import sleep

delay = 1.0

leds.user_num(0, True)
sleep(delay)
leds.user_num(1, True)
sleep(delay)
leds.user_num(2, True)
sleep(delay)
leds.user_num(3, True)
```

Objective 5 - Light Show!

Use your LED control capabilities to make CodeBot shine!

- Turn LEDs **ON** and **OFF** to make a *flashing* display.
- Try smaller time values for quick changes.

Check the 'Trek!

Modify your code to turn each LED **OFF** after the delay.

Run It!

Can you make your light show ***flashier***?

- Try a smaller delay, like `delay = 0.1`
- Add more LEDs, up to `leds.user_num(7, True)`
- Why not add `leds.ls_num()` **line sensor LEDs** to the party!

CodeTrek:

```
from botcore import leds
from time import sleep

delay = 1.0

leds.user_num(0, True)
sleep(delay)
# TODO: Turn LED 0 OFF!
```

The second argument `leds.user_num` takes is `on`.

Since we want to turn the light **OFF**, simply pass `False`!

```
leds.user_num(0, False)
```

```

leds.user_num(1, True)
sleep(delay)
# TODO: Turn LED 1 OFF!
```

Just like the last step, turn user LED `1` **OFF**!

```
leds.user_num(1, False)
```

```

leds.user_num(2, True)
sleep(delay)
# TODO: Turn LED 2 OFF!
```

One last time! Turn LED `2` **OFF**!

```
leds.user_num(2, False)
```

```

leds.user_num(3, True)
```

Goal:

- *After* **sleeping**, turn **OFF** lit user LEDs using `leds.user_num(num, False)`.

Tools Found: Keyword and Positional Arguments, LED

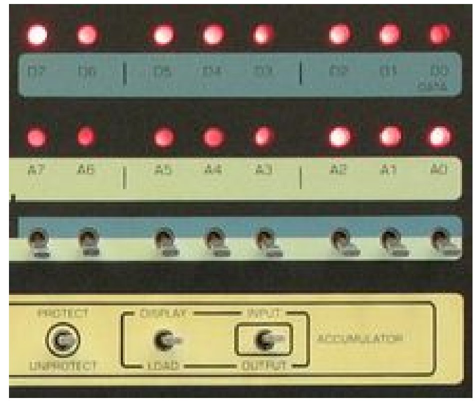
Solution:

```
from botcore import leds
from time import sleep

delay = 1.0

leds.user_num(0, True)
sleep(delay)
leds.user_num(0, False)

```

```
leds.user_num(1, True)
sleep(delay)
leds.user_num(1, False)

leds.user_num(2, True)
sleep(delay)
leds.user_num(2, False)

leds.user_num(3, True)
```

Objective 6 **- Bright Byte Lights!**

Old school? *Yes and No!*



Check out the picture on the right. It's from the front panel of one of the first "personal computers".

- Do you see the *8 DATA* LEDs?
- 8-bits of 🔧binary data is called a **BYTE** in Computer Science.

Our PCs and *mobile devices* have come a long way since then!

- CodeBot's 🔧CPU is fantastically powerful compared to those old machines :-)
- But just like PCs, phones, and ancient computers, at the core it fundamentally operates in 🔧binary.
- And since CodeBot happens to have a **BYTE sized** array of User LEDs, **binary** is a great way to program them!

Concept: *CodeBot LEDs*

All of the 🔧CodeBot LEDs can be controlled with Python code.

- **User** LEDs in the center of the 'bot.
- **Line Sensor** LEDs across the front edge, directly above the *line sensors*.
- **Prox** LEDs just in front of each *proximity sensor*.

You can control them all in a similar way, for example:

```
leds.user_num(0, True)
leds.ls_num(0, True)
leds.prox_num(0, True)
```

But they also have more powerful control functions, like the ability to display a value in 🔧binary.

Use Python's `0b` prefix to designate binary numbers *(1=ON, 0=OFF)*.

Ex: Light alternating *user* LEDs

```
leds.user(0b10101010)
```

Ex: Light all 5 *ls* LEDs

```
leds.ls(0b11111)
```

Click on the 🔧CodeBot LEDs tool to learn more.

Create a New File!

Use the **File** → **New File** menu to create a new file called ***BinaryLEDs***.

Check the 'Trek!

Create a new file and name it **BinaryLEDs**.

- Write code to control the Line Sensor LEDs on CodeBot, but this time do it with 🔧binary numbers.

The CodeTrek code uses 🔧binary values to animate the Line Sensor LEDs.

Run It!

Try to add your own *binary patterns* to the sequence!

CodeTrek:

```
from botcore import leds
from time import sleep

leds.ls(0b00100)
sleep(0.5)
leds.ls(0b01110)
sleep(0.5)
# TODO: Light ALL the ls LEDs using binary
```

Simply set each value in the **binary number** to `1`!

```
leds.ls(0b11111)
```

Goals:

- Create a **New File** named `BinaryLEDs`.
- Light the **middle** *ls* LED using **binary**.
- Light the **middle 3** *ls* LEDs using **binary**.
- Light **all** *ls* LEDs using **binary**.

Tools Found: Binary Numbers, CPU and Peripherals, CodeBot LEDs

Solution:

```
from botcore import leds
from time import sleep

leds.ls(0b00100)
sleep(0.5)
leds.ls(0b01110)
sleep(0.5)
leds.ls(0b11111)
```

***Quiz 1* - Checkpoint**

You're off to a great start!

- Controlling LEDs is the traditional starting point for lots of *physical computing* projects.
- Now take a minute or two to review what you've learned.

Question 1: Why would you add a delay (sleep) after you turn on each LED?

✓ So you can see them turn on one at a time.

✗ So they will turn on.

✗ To give the LEDs time to cool off.

Question 2: When you use the debugger, the line of code with the highlight:

- ✓ Will run the next time you press STEP.
- ✗ Ran the last time you pressed STEP.
- ✗ Is currently running.
- ✗ Will stop the program.

Question 3: The statement `sleep(1.5)`

- ✓ Pauses the program for 1.5 seconds.
- ✗ Pauses the program for 1.5 milliseconds
- ✗ Allocates 1.5 kilobytes of sleep space.

Question 4: What does `from time import sleep` do?

- ✓ Gives this code access to the "sleep" function from the "time" library.
- ✗ Sleeps from time to time.
- ✗ Allows this code to read the current time.

Question 5: Which LED does the following turn ON: `leds.ls(0b00100)`

- ✓ Line Sensor LED 2
- ✗ User LED 5
- ✗ Line Sensor LED 3
- ✗ Line Sensor LED 1

Objective 7 - Get Moving

It's time to power-up CodeBot's motors!!

- Make sure you have **batteries** loaded into your 'bot.
- Set the **POWER Switch** to **BATT** when you're using the motors.
  - Even when **USB** is *connected* this keeps your PC from having to supply all the power.

> **Caution**
>
> In the following steps, be **careful** as you run your programs! *You'll need some space* to let the 'bot move around. Also, it's helpful if your USB cable is long enough to allow a bit of movement.

> **Run It!**

CodeTrek:

```
from botcore import *
```

Importing `*` from a library imports *everything!*

- In this case, it gives your code access to *everything* in `botcore`, including `leds` and more!

```
from time import sleep

motors.enable(True)
```

You have to call `enable(True)` *before* the motors will move!

```
motors.run(LEFT, 50)
```

Start by powering just *one* of the motors at **50%**.

```
sleep(1.0)
# TODO: Disable the motors
```

Turn the motors **OFF** before the program ends by passing `False` to `motors.enable`.

```
motors.enable(False)
```

Goals:

- **Enable** your CodeBot's **motors** using `motors.enable(True)`.
- **Power** the `LEFT` motor at `50%`.
- **Disable** your CodeBot's **motors** using `motors.enable(False)`.

Tools Found: Motors, import, Reboot

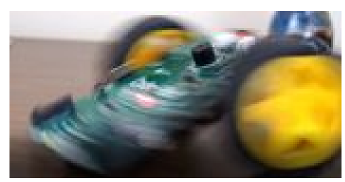
Solution:

```
from botcore import *
from time import sleep

motors.enable(True)
motors.run(LEFT, 50)
sleep(1.0)
motors.enable(False)
```

Objective 8 - Rotation Time!

You will need to use **both motors** for this one.

- Spin the wheels in opposite directions to *rotate* your 'bot.
- To rotate in-place, both wheels must have the *same speed*.
- Just change the direction so one is **negative** and the other **positive**.



Check the 'Trek!

Modify your program a little, and you'll have it!

- Check out the `# Comments` - they are *optional* for you to type.

Concept: *Comments and Readability*

In the CodeTrek code, did you notice the `# comment` lines?

- As you write code, imagine that someone who has never seen it before will have to read it and figure it out.
- A year from now, you might even pick up your **own** code and say: "what was I thinking!?"

Readability in code means making it for **humans** to understand.

- Use Comments - notes in the code about what you're doing.
- Use descriptive names for things.
- Use whitespace in keeping with the accepted *style* of the language.

In Python, anything that follows a `# to the end of the line`

...is a comment, meaning it is *ignored* by the computer.

Run It!

You'll need to **plug in the USB cable** and *RUN* your code as usual after any change.

Physical Interaction: *Now unplug USB cable, and REBOOT!*

Unplug the USB cable and press the REBOOT button to test out your program.

- Does your 'bot spin around a full 360° circle?
- Try making it spin longer! Faster! Sloooower...

CodeTrek:

```
from botcore import *
from time import sleep

motors.enable(True)

# Run LEFT motor at 50% power forward
```

This is a `# comment`!

In **Python**, this line will be *skipped* over!

```
motors.run(LEFT, 50)

# TODO: Run RIGHT motor at 50% power backward
```

Backwards?!

- **Negative** integers makes the motor run backwards.
- *Therefore* `motors.run(RIGHT, -50)` will turn the `RIGHT` motor backward at `50`% power!

```

# Rotate for 1 second
sleep(1.0)

motors.enable(False)
```

Goals:

- Run the `LEFT` motor **forward** at `50`% power.
- Run the `RIGHT` motor **backward** at `50`% power.

Tools Found: Comments, Blank Lines and Whitespace, Reboot

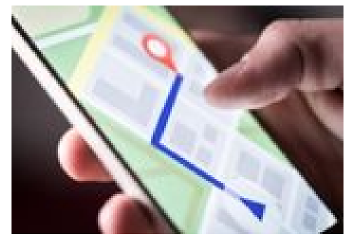
Solution:

```
from botcore import *
from time import sleep

motors.enable(True)

```

```
# Run LEFT motor at 50% power forward
motors.run(LEFT, 50)

# Run RIGHT motor at 50% power backward
motors.run(RIGHT, -50)

# Rotate for 1 second
sleep(1.0)

motors.enable(False)
```

Objective 9 - First Navigation Challenge!

Your first of *many* such challenges to come...



Program CodeBot to Drive in a *Square*

Requirements:

- Your bot's journey must start and end at *approximately* the same spot.
    - (you'll learn precise control later, just get *close* this time!)
- The sides of the square must be about 1 foot (30cm) in length.

Concept: *Algorithms*

You're facing a problem to solve with code!

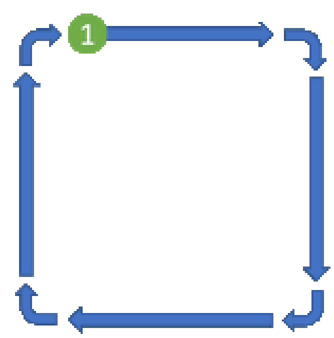
- It's going to require a series of steps. (a *sequence*)
- You'll likely make use of 🔧library functions along the way.

    You're going to be making an **Algorithm**, dude!

Algorithms are *precise sequences of instructions* that the computer can follow exactly, one step at a time.

Begin your SQUARE algorithm by *breaking the problem down into steps*

1. Enable the motors
2. Go forward 1 foot
3. Turn right 90°
4. Go forward 1 foot
5. Turn right 90°
6. Go forward 1 foot
7. Turn right 90°
8. Go forward 1 foot
9. Turn right 90°
10. Disable the motors
11. Finished!



A few steps... but not *too* complicated, right?

- Are you ready to **code** this?
- Wait! First, one more *pro-tip*...

Concept: *Divide and Conquer*

Break your problem down into bite-sized pieces

Some steps in an **algorithm** may *sound* simple, but really they're hiding a few steps of their own! For example:

Go forward 1 foot

There is no built-in command for CodeBot to do that! So you need *another* ***algorithm*** to do just this step. (you might call it a *sub-algorithm*!)

```
# Go forward 1 foot
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(3.0)  # TEST THIS! Not sure what value is needed here...
```

When you're facing complexity, remember: 🔧Divide and Conquer!

Create a New File!

Use the **File** → **New File** menu to create a new file called ***NavSquare***.

Check the 'Trek!

Run It!

This one will take some testing and experimentation to get right!

CodeTrek:

```
from botcore import *
from time import sleep

# Navigate in a SQUARE pattern
# TODO: ...just add code!
```

Goals:

- Call `motor.run` *atleast* **6 times**.
- Call `sleep` *atleast* **6 times**.

Tools Found: import, Divide and Conquer, Variables

Solution:

```
from botcore import *
from time import sleep


# Run LEFT motor at 50% power forward
motors.run(LEFT, 50)

# Run RIGHT motor at 50% power backward
motors.run(RIGHT, -50)

motors.run(LEFT, 50)
motors.run(RIGHT, -50)

motors.run(LEFT, 50)
motors.run(RIGHT, -50)

# Rotate for 1 second
sleep(1.0)
sleep(1.0)
sleep(1.0)
sleep(1.0)
sleep(1.0)
sleep(1.0)
motors.enable(False)
```

Objective 10 - Choose Your Adventure!

Your final ***Time and Motion*** series project is to make your **NavSquare** program more ***user-friendly***.

Imagine that you gave your 'bot to someone for testing...

Hypothetical User Feedback:

- "As soon as I run the program it starts moving. *Yikes!* I want it to only move if I **push a button** first."
- "It always goes the **same direction** around the square. *Boring!* I want to use push-buttons to choose **right turns** or **left turns**."

Push-Button Controls

You'll need to learn about 🔧CodeBot buttons to satisfy this feature request!

- The **botcore** function you'll need is called `buttons.was_pressed()`
- But you *also* need a way to change the 🔧control flow of your program!

Python's `if` statement is what you need:

Can you follow the *algorithm* below? Check out the 🔧control flow tool for an explanation!

```
if buttons.was_pressed(0):
    # turn left.
elif buttons.was_pressed(1):
    # turn right.
else:
    # stop - no button was pressed.
```

Concept: *Control Flow and Branching*

The `if condition` statement tells Python to only run the block of code 🔧indented beneath it **if** the *condition* is `True`.

- `elif` is short for "else if"

Be sure all code you want to run inside a block is 🔧indented *at the same level*.

- The **colon :** at the *end* of `if` expressions introduces a new block.
- So always 🔧indent the next line after a **colon**!

Pro Tip: *Use the TAB key to indent!*

In the spirit of 🔧divide and conquer, *test* your knowledge of 🔧CodeBot buttons and 🔧control flow with a new standalone program before you add it to your **NavSquare** program.

Create a New File!

Use the **File** → **New File** menu to create a new file called ***WhatIf***.

Check the 'Trek!

The provided code will make sure you understand how `buttons` and the `if` statement work!

Run It!

Watch for the ***USER*** LED sequence **4** ... **3** ... and make sure you press a button *during the countdown*.

- Just a momentary *press* will do. No need to hold down the button.

Debug

Try **stepping** through this code.

- Press *BTN-0* or *BTN-1* while the debugger is waiting on a line.
  - Is the button press still detected when your code reaches the `if` statements?
- Use the debugger to step through the `if`, `elif`, and `else` when **NO** button is pressed.
  - Does it completely skip the `if` statement?
  - *OR* does it test the `if buttons.was_pressed():` and just skip the indented code beneath it?

Does `buttons.was_pressed()` detect a button press that happens *while the debugger is **stopped*** on a line of code?

CodeTrek:

```
from botcore import *
from time import sleep

# Give user 2 seconds to press a button.
# Use LEDs to show "countdown"
leds.user_num(4, True)
sleep(1.0)
leds.user_num(3, True)
sleep(1.0)

# Set LEDs based on which button was pressed
if buttons.was_pressed(0):
```

> If button `0` was pressed, the *following* indented code will be **executed!**
>
> - In *this case*, `leds.user_num(0, True)` would be called.

```
    leds.user_num(0, True)
elif buttons.was_pressed(1):
    leds.user_num(7, True)
```

> If button `0` was **NOT** pressed **and** button `1` WAS pressed, execute `leds.user_num(7, True)`.

```
else:
    leds.user(0b00000000)
```

> If button `0` was **NOT** pressed **and** button `1` was **NOT** pressed, execute `leds.user(0b00000000)`.

Hint:

- Make sure you press the **STEP IN** button *all the way through* the program!
- You'll know you're there when the program *ends!*

Goals:

- Call `leds.user_num(0, True) if` **BTN-0** is *pressed.*
- **DEBUG** your program and use the **STEP IN** button.

Tools Found: Buttons, Branching, Indentation, Divide and Conquer

Solution:

```
from botcore import *
from time import sleep

# Give user 2 seconds to press a button.
# Use LEDs to show "countdown"
leds.user_num(4, True)
sleep(1.0)
leds.user_num(3, True)
sleep(1.0)

# Set LEDs based on which button was pressed
if buttons.was_pressed(0):
    leds.user_num(0, True) #@1
elif buttons.was_pressed(1):
    leds.user_num(7, True) #@2
```

```
else:
    leds.user(0b00000000) #@3
```

Objective 11 - Button it Up!

Now that you've mastered CodeBot buttons and control flow you can complete your **NavSquare** project.

- Start with an **LED Countdown** so the user has time to press a button.
- Then use `if`, `elif`, and `else` based on `buttons.was_pressed()` to choose
  - **LEFT** turns
  - **RIGHT** turns
  - or **STOP**

Type in the Code

Use the *File → Browse Files...* menu to re-open your **NavSquare** program.

Add code so it begins just like the previous step:

- An **LED Countdown** at the start, to give the user time to press a button.
- An `if` ...algorithm to select *navigation direction* based on *push buttons*.

```
# Navigate in a SQUARE pattern with push-button control.
# TODO: ...just add code!
```

Don't forget to use the Editor Shortcuts if you need to copy a block of code.

- In a future project you will learn ways to reduce repetition, but for now just make it work!

Run It!

Test your code thoroughly!

- Be sure to **test** all **3** user commands:
  - BTN-0 → ***LEFT TURNS***
  - BTN-1 → ***RIGHT TURNS***
  - No button → ***STOP***

"If it's not tested, it's broken"

- Bruce Eckel

CodeTrek:

```
from botcore import *
from time import sleep

# Use LEDs to show "countdown"
leds.user_num(4, True)
sleep(1.0)
leds.user_num(3, True)
sleep(1.0)
```

Start with the LED countdown from last objective.

```

if buttons.was_pressed(0):
    # TODO: Turn LEFT in a square!
```

When the user hits *BTN-0*, run your square algorithm **LEFT**.

```
elif buttons.was_pressed(1):
    # TODO: Turn RIGHT in a square!
```

When the user hits *BTN-1*, run your square algorithm **RIGHT**.

```
else:
    # TODO: Stop the motors!
```

If the user does *nothing*, **STOP** the motors.

- `motors.enable(False)`

Goals:

- **Start** with an **LED Countdown** to give the user time to press a button.
- Alternate `leds.user_num` and `sleep` ***atleast* twice.**
- Use:
- `if buttons.was_pressed(0):`
- `elif buttons.was_pressed(1):`
- `else:`
- Call `motor.run` ***atleast* 6 times.**

Tools Found: Buttons, Branching, Editor Shortcuts, LED, Motors

Solution:

```
from botcore import *
from time import sleep

# Give user 2 seconds to press a button.
# Use LEDs to show "countdown"
leds.user_num(4, True)
sleep(1.0)
leds.user_num(3, True)
sleep(1.0)

# Set LEDs based on which button was pressed
if buttons.was_pressed(0):
    leds.user_num(0, True) #@1
elif buttons.was_pressed(1):
    leds.user_num(7, True) #@2
    motors.run(LEFT, 100)
    motors.run(RIGHT, 100)
    motors.run(LEFT, 100)
    motors.run(LEFT, 100)
    motors.run(RIGHT, 100)
    motors.run(LEFT, 100)
    motors.run(RIGHT, 100)
    motors.run(RIGHT, 100)
else:
    motors.enable(False)
```

<u>Mission 3 Complete</u>

You've learned some fundamental computer science and robotics principles:

- Controlling LEDs and Motors with specific timing and sequencing.
- Using Python language `import` libraries.
- Reading CodeBot buttons inputs.
- Changing the control flow of your programs on the fly.

This code is *for real!*

Yeah, *robots* rock this kind of code. ***But so do:***

- Digital coffee makers and espresso machines.
    - Beans, water, heat, timing and sequencing - *sweet!*
- Music sequencers.
- Electric toothbrushes.
- ...and more!

Try Your Skills

Suggested Re-mix Ideas:

- Make CodeBot drive in a **circle**.
- Make the LEDs flash in a pattern of your choice
    - ***...while*** CodeBot is driving in a square!

Mission 4 **- Animatronics**

You have been hired by a major *Theme Park*!

- Your task is to create a new ***Animatronic Robot Exhibition***.

Gotta write some *Python code* and get the show running with **CodeBot**!

At a coffee shop meeting, the manager gave you a classic *Napkin Sketch* of what she's looking for.

Your notes from the meeting:

- The robot starts out "Asleep", constantly blinking RED LEDs in a "cool" pattern.
- Each guest **presses a button** as they enter the small *entrance room*.
- When **5** guests have entered, the show starts!
- Move forward 3 feet.
- Spin around about 360° while making "cute robot sounds".
- Play a greeting "Fanfare" sound.

After that, a cast member will reposition and reboot the robot to be ready for the next group of guests.

Project Goals:

- Blink red **'user'** LEDs ***constantly*** in a "cool" pattern.
- Count to **5 guests** using CodeBot buttons BTN-0.
  - *Upgrade 1:* **Show count** on green 'LS' LEDs.
  - *Upgrade 2:* ***Beep*** when a button is pressed.
- When count is 5, drive forward 3 feet.
- Make "cute robot sounds" while rotating 360°
- Play a short "fanfare" tune over the speaker

Objective 1 **- Forever Flashing**

Create a New File!

Concept

A `while condition:` statement tells Python to repeat the block of code indented beneath it as long as the given *condition* is **True**.

In the code above we used the *literal* value `True` as the condition, so we have an **infinite loop** - one that never ends, because `True` is always... **True**!

Run It!

CodeTrek:

```
from botcore import *
from time import sleep

# Define variables for blink delay and LED number.
```

```
delay = 0.5
n_led = 0

while True:
```

> `while: condition` repeats the 🔧indented block of code as long as the `condition` is `True`.
>
> - To run **infinitely**, simply set `condition` to `True`!

```
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)
    sleep(delay)
```

Goals:

- *Declare* 🔧variables `n_led` and `delay`.
- Use a `while True:` 🔧loop.

Tools Found: CodeBot LEDs, Loops, Indentation, bool, Variables

Solution:

```
from botcore import *
from time import sleep

# Define variables for blink delay and LED number.
delay = 0.5
n_led = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)
    sleep(delay)
```

Objective 2 - A Cool Pattern

Check the 'Trek!

Concept: *Updating a Variable*

You can assign a new value to a 🔧variable at any time.

It's very common for the *new* value to be based on the *old* value of the variable!

- That's what is happening with this code:

```
# Add +1 to n_led, and store result in n_led.
n_led = n_led + 1
```

Does it look odd to have `n_led` on *both* sides of the *assignment* statment?

- Just remember that everything to the **right** of the *equals* runs ***first***.
- So the assignment happens in **two** stages.

For example, if `n_led` was `0` before: 1. Do the **right hand side**: `n_led + 1` ⟶ `0 + 1` ⟶ `1` 1. Next, do **assignment**: `n_led` ⟵ `1`

So after the update, `n_led` is `1`.

Run It!

CodeTrek:

```
from botcore import *
from time import sleep

delay = 0.5
n_led = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)
    sleep(delay)

    # TODO: Add +1 to n_led, and store result in n_led.
```

Take the value of `n_led` and increase it's *value* by `1`.

```
n_led = n_led + 1
```

Goal:

- Add +`1` to `n_led` and *store the result* in `n_led`.

Tools Found: Loops, Variables, Branching, Indentation

Solution:

```
from botcore import *
from time import sleep

delay = 0.5
n_led = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)
    sleep(delay)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
```

***Objective 3* - Fixing A Cool Pattern**

Squashin' Bugs

Debug

Concept: *double equals sign*

Why is there a "double equal" sign in the code?

- A **single equal** means **"assignment"**.
  - Like ***assigning*** `n_led = 0` above your loop.
- A **double equal** is a 🔧comparison operator, just like `>` and friends.

Run It!

CodeTrek:

```
from botcore import *
from time import sleep
```

```

delay = 0.5
n_led = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)
    sleep(delay)


    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
```

Indentation Note:

- The `if` statement is indented to be inside the `while` loop.
  - But *under* the `if` statement is a **second** level of indentation: a block of code that runs only when the `if` expression is `True`.

```
        # TODO: reset n_led
```

Set `n_led` back to `0` to *restart the cycle!*

```
n_led = 0
```

Goals:

- Use `if n_led == 8:` in your `while` loop.
- *Within* the `if n_led == 8:` code block:
- **Reset** `n_led` to `0`

Tools Found: Branching, Comparison Operators, undefined

Solution:

```
from botcore import *
from time import sleep

delay = 0.5
n_led = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)
    sleep(delay)


    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0


```

Objective 4 - Counting the Guests - part 1

Here's your next **objective** in this project:

- Count CodeBot buttons **BTN-0** presses up to **5 guests**.
  - When the count reaches **5** you need to **break** out of the loop so CodeBot can do the next move - *driving forward!*

Is there a Python statement to *break* out of a loop?

Glad you asked! To break out of a loop, use a statement called... wait for it...

`break`

This simple statement ***exits the nearest enclosing loop.***

Check the 'Trek!

Make a small addition at the end of the code *inside* your `while` loop.

- To start with, just **test** the `break` statement you just learned about.
  - Break out of the loop immediately when BTN-0 is pressed.
- *Mind your indentation!*

Run It!

Test it out!

- Are you able to `break` out of the loop by pressing BTN-0?

CodeTrek:

```
from botcore import *
from time import sleep

delay = 0.1
n_led = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)
    # No sleep here!


    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
      n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
      # TODO: Break out of the loop
```

Use the `break` statement to **stop** the loop!

Goal:

- Use the `break` statement.

Tools Found: Buttons, Indentation, Loops

Solution:

```
from botcore import *
from time import sleep

delay = 0.1
n_led = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)
    # No sleep here!

```

```
    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
      n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
      break
```

Objective 5 **- Counting the Guests - part 2**

Check the 'Trek!

Run It!

How's it counting?

Test your program a few times!

- Not just *once or twice*!
  - Do the complete *5-count* at least *four times*, watching the **green** LEDs closely each time you press **BTN-0**.

Caution: *Contact Bounce*

Have you noticed that sometimes a ***single*** *button-press* causes **more than one count**?

Pushing a button causes two *metal pieces to contact* each other, allowing electric current to flow.

- At a microscopic level, those metal "contacts" **bounce** a little before settling down.

In the next step you will add code to **debounce** the button-press.

CodeTrek:

```
from botcore import *
from time import sleep

delay = 0.1
n_led = 0
n_guests = 0
```

Dont forget to *assign* `0` to the **variable** `n_guests` up here!

```

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

   # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        # TODO: Use line sensor LEDs to show guest count
```

Display variable `n_guests` by lighting the *corresponding* **LED number** with the function:

```
leds.ls_num(n_guests, True)
```

```
        # TODO: Increment n_guests
```

Just like `n_led` above! Update **variable** `n_guests` by adding `1`.

```
n_guests = n_guests + 1
```

```

        if n_guests == 5:
            break
```

Goals:

- Increment `n_guests` when 🔧button `0` is pressed.
- **Display** the guest count using the 🔧Line Sensor LEDs.

Tools Found: CodeBot LEDs, Buttons, Variables

Solution:

```
from botcore import *
from time import sleep

delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

   # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break
```

Objective 6 **- Beep Beep, I'm a Bot!**

Audible Button Feedback Tones

Now it's time for you to code the second *"upgrade"* goal of this project:

- **Upgrade 2:** ***Beep*** when a button is pressed.

Concept: *CodeBot Speaker*

This project uses just two basic functions of CodeBot's 🔧Speaker

- `spkr.pitch(440)`
  - Start playing a continuous tone at a given frequency in Hertz (ex: 440Hz)
- `spkr.off()`
  - Stop all sounds.

Check the 'Trek!

Armed with the above knowledge, you are ready to add beeps to your code!

Hints:

- It's like *blinking* an LED, but with ***sound:*** **ON**→*delay*→**OFF**.
- Insert your **beep** code right below the `if buttons.was_pressed(0):`.

Run It!

Sounding good?

- Sounds add a whole new dimension to the project.
- I think it improves the User Interface since guests will know for sure they've been counted.

CodeTrek:

```
from botcore import *
from time import sleep

delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0


    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        # TODO: Play a tone at 440Hz
```

Use the function `spkr.pitch(frequency)` to *play a sound* on your 'bot's speaker!

To play a sound at 440Hz:

```
spkr.pitch(440)
```

```
        sleep(0.1)
        # TODO: Turn the speaker off
```

Stop **all** sound from your 'bot's speaker with the function `spkr.off()`

```

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1


        if n_guests == 5:
            break
```

Goal:

- When button `0` is **pressed**:
- Play a *continuous* tone at 440Hz using `spkr.pitch(frequency)`.
- Call `sleep(0.1)`.
- Turn off the speaker using `spkr.off()`.

Tools Found: Speaker, UI, Buttons

Solution:

```python
from botcore import *
from time import sleep

delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0


    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1


        if n_guests == 5:
            break
```

<u>*Objective 7* - Beep Beep 2</u>

One more thing - *Debouncing the Button*

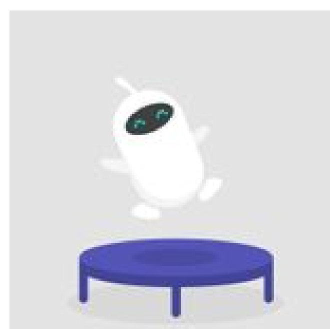
- We now have an additional **delay** right after the *button press is detected*, but the problem is still present!

Concept: *was_pressed() Back Story*

Consider how `buttons.was_pressed(0)` works. It actually does two things:

- Return `True` if a button has been pressed.
  - Button presses are monitored by a 🔧CPU *interrupt handler*.
- Reset the internal status of the button to `False`.
  - ...so it won't return `True` again unless the button was pressed again since last `was_pressed(0)`.

But when the **button bounces**, here's the sequence:



1. User presses button... *now in slooow moootiiooon...*
2. First Contact!
3. was_pressed(0) → `True  # we detected the first press!`
4. It's all good. The internal status of the button is reset to `False`.
5. Bounce!!
6. The CPU *interrupt handler* saves the *was_pressed* status.

Oh No! ...Next time around the loop when we call `was_pressed(0)` it will remember this bounce :-(

Concept: *Debounce*

Debouncing a button is simple:

1. Detect a button press
2. Delay long enough for the bouncing contacts to settle down.
3. Reset internal button press status.

You're *already* doing the first two steps:

- You detect a button press with `buttons.was_pressed(0)`.
- The **beep** lasts for 0.1 seconds, which is plenty of time for bouncing to settle down.

But **how** do you ***reset the internal button press status ??***

Easy! Just call `buttons.was_pressed(0)` again.

- It really doesn't matter whether it returns `True` or `False`...
  - The important thing is that `was_pressed()` ***resets the internal status***.

Check the 'Trek!

Now that you understand what's happening, the fix is *one simple line of code*.

- Insert a call to `buttons.was_pressed(0)` just after your **beep** code.

Run It!

- Test a few runs, and you'll notice the button presses are *spot-on*!

CodeTrek:

```
from botcore import *
from time import sleep

delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0


    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # TODO: After delay, DEBOUNCE the button
```

Reset the *internal* button press status by calling `was_pressed` again!

```
buttons.was_pressed(0)
```

```

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1


        if n_guests == 5:
            break
```

Goal:

- **Debouce** the button by calling `buttons.was_pressed(0)` ***after*** the speaker delay.

Tools Found: CPU and Peripherals, Buttons

Solution:

```
from botcore import *
from time import sleep

delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0


    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1


        if n_guests == 5:
            break
```

Quiz 1 - Checkpoint

Your project is going well!

- *Animatronics* is a great way to expand your ***coding skills***.
- Now take a minute or two to review what you've learned.

Question 1:

```
n = 7
n = n + 1
```

What is the value of `n` ***after*** the statement `n = n + 1` runs?

✓ 8

✗ 7

✗ 6

✗ 1

✗ 'm'

Question 2: Will the following program turn the LED on?

```
from botcore import leds
while False:
    leds.user_num(0, True)
```

✓ No.

✗ Yes.

Question 3:

```
from botcore import leds
from time import sleep

i = 0
while i < 3:
    leds.user_num(0, True)
    sleep(1.0)
    leds.user_num(0, False)
    sleep(1.0)

i = i + 1
```

How many times will the LED flash when the code above runs?

✓ Infinite times. The increment is outside the loop.

✗ Two times.

✗ Three times.

Question 4: The `buttons.was_pressed(0)` function returns `True` when:

✓ The button has been pressed since was_pressed(0) was last called.

✗ The button has been pressed since the program started.

✗ The button was pressed in the last 100 milliseconds.

Objective 8 - Moving Forward

Check the 'Trek!

Run It!

Adjust the parameters of `sleep()` and `motors.run()` *(by changing the values passed into them)* until your 'bot is moving forward like a *runway model* making a full-turn and stopping at the end of the catwalk!

- Press CodeBot's **reboot** button to *quickly* restart and try again.

CodeTrek:

```
from botcore import *
from time import sleep

# Sweep LEDs, counting guests as they arrive
delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0
```

```

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break


# Move forward 3 feet
```

This will take some *trial and error.*

Take your time!

```
motors.enable(True)
# TODO: run left motor
# TODO: run right motor
# TODO: sleep just long enough...
```

Reference the *previous* mission if you need a **refresher!**

Your code for driving 3 feet **forward** *may* look like this:

```
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(2)
```

```

# Spin 360 degrees
```

Similarly to driving 3 feet,
spinning **360** degrees is going to take some *trial and error.*

Run your motors in *opposite* directions!

```
# TODO: run both motors in opposite directions
# TODO: sleep just long enough...

# Stop
motors.enable(False)

```

Goals:

- Run *both* 🔧motors **forward,** then `sleep`.
  - Try to drive *3 feet* forward.
- Run *both* 🔧motors in **opposite** directions, then `sleep`.
  - Try to do a *360!*

Tools Found: Indentation, Motors, Parameters, Arguments, and Returns

Solution:

```
from botcore import *
from time import sleep

# Sweep LEDs, counting guests as they arrive
```

```
delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break


# Move forward 3 feet
motors.enable(True)
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(2)

# Spin 360 degrees
# TODO: run both motors in opposite directions
# TODO: sleep just long enough...
motors.run(LEFT, 50)
motors.run(RIGHT, -50)
sleep(1)

# Stop
motors.enable(False)
```

Objective 9 - Cute Robot Sounds

Type in the Code

Run It!

How does it sound?

- Interesting... but...
- Pretty ***quick***, right?

Maybe you can ***stretch it out*** and make it sound more "robot-like" if you loop your new sound!

Run It!

CodeTrek:

```
from botcore import *
from time import sleep

```

```
# Outer loop to play sound 10 times
count = 0
while count < 10:
```

This loop will iterate until the variable `count` is *greater than* `9`.

```
    # TODO: Iterate the count variable
```

Iterate the variable `count`!

```
count = count + 1
```

```

    # Sweep the frequency from 100-1000
    f = 100
    while f < 1000:
        f = f + 1
        spkr.pitch(f)

spkr.off()

# Sweep LEDs, counting guests as they arrive
delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break

# Move forward 3 feet
motors.enable(True)
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(2.0)

# Spin 360 degrees
motors.run(LEFT, -50)
motors.run(RIGHT, -50)
sleep(0.5)

# Stop
motors.enable(False)
```

Goals:

- *Sweep* the **frequency** from `100`-`1000` using a while loop with the `condition`:
- `while f < 1000:`
- *Sweep* the **frequency** from `100`-`1000` `10` times using a while loop with the `condition`:

- `while count < 10:`

Tools Found: Speaker, Editor Shortcuts, Loops, import, Indentation, Iterable, Variables

Solution:

```
from botcore import *
from time import sleep

# Outer Loop to play sound 10 times
count = 0
while count < 10:
    count = count + 1

    # Sweep the frequency from 100-1000
    f = 100
    while f < 1000:
        f = f + 1
        spkr.pitch(f)

spkr.off()

# Sweep LEDs, counting guests as they arrive
delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break

# Move forward 3 feet
motors.enable(True)
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(2.0)

# Spin 360 degrees
motors.run(LEFT, -50)
motors.run(RIGHT, -50)
sleep(0.5)

# Stop
motors.enable(False)
```

Objective 10 **- Really Cute Sounds**

This time try for something a little more **melodic**.

- Rather than **sweeping** the pitch, go for a ***random "beep" - "bloop"*** effect with single tones.

To make *random* tones you'll be using the **random** Python module, so look for a new 🔧import statement.

Concept: *random*

Python's random module makes it easy to work with random numbers.

One function it provides is `randrange(start, stop)`. This generates a random integer that's greater than or equal to `start` and less than `stop`. *See the complete docs for more details.*

```
from random import randrange

# Get random number from [1 to 8)
f = randrange(1, 8)
```

Type in the Code

Modify your code as follows:

- Leave the *outer loop* that counts to 10 as-is.
- Replace the *inner loop* which was **sweeping** the pitch from 100 to 1000 (Hz).
- In its place, do the following:
  - Pick a random pitch between 100 and 1000Hz (use `randrange()`).
  - Play that **pitch** for 0.1 seconds.

That's it!

Run It!

Try that one a few times...

- Pretty ***cute*** huh?
- Aw... it's *adorable!*

That's more like it.

Adjust the `while count < 10` to count **higher** if you need to *increase* the duration of **cute sounds**.

- Remember, it needs to play for as long as it takes CodeBot to **spin**.
- You might want to ***slow down*** the motors while spinning, to give **more time** for *sounds* to play!

Check the 'Trek!

Now you need to move this code to its proper place

- This code ***replaces*** the `sleep()` while *spinning*, so make sure to delete that line.
- Select the whole block of code, from `count = 0` to `spkr.off()`.
- Cut it with *CTRL-X*, then position your cursor *near the bottom* of your code where you've just started the **spin**.
- Use *CTRL-V* to paste the code where you need it.
- Oh, and **don't forget to leave** `from random import randrange` **at the top of your code**

Run It!

Wow! You are *almost* finished.

- Make any adjustments needed to your code.

CodeTrek:

```
from botcore import *
from time import sleep
from random import randrange
```

Don't forget to leave this line at the *top* of your file!

```

# Sweep LEDs, counting guests as they arrive
delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break

# Move forward 3 feet
motors.enable(True)
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(2.0)


# Spin 360 degrees
motors.run(RIGHT, -50)

# Play "cute sounds" while spinning
count = 0
while count < 10:
    count = count + 1

    f = # TODO: Choose a random frequency
```

Use the 🔧function `randrange(start, stop)` to generate the *random* frequency.

- `f = randrange(100, 1000)`

```
    spkr.pitch(f)
    sleep(0.1)

# Stop sounds
spkr.off()

```

```
# Stop motors
motors.enable(False)


```

Goals:

- 🔧Import `randrange` *from* `random`.
- Assign `f` to a 🔧random frequency using `randrange(start, stop)`
- *Copy* and *paste* your *"cute sounds"* code near the *bottom* of the file.
- Place it right *before* you turn **OFF** the 🔧motors.

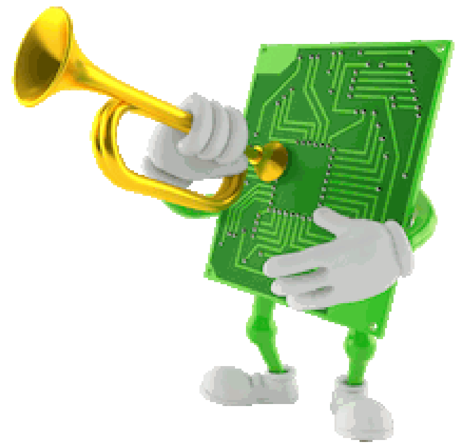
Tools Found: import, Random Numbers, int, Motors, Functions

Solution:

```
from botcore import *
from time import sleep
from random import randrange

# Sweep LEDs, counting guests as they arrive
delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break

# Move forward 3 feet
motors.enable(True)
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(2.0)


# Spin 360 degrees
motors.run(RIGHT, -50)

# Play "cute sounds" while spinning
count = 0
while count < 10:
    count = count + 1

    # Choose a random frequency
    f = randrange(100, 1000)
    spkr.pitch(f)
    sleep(0.1)

```

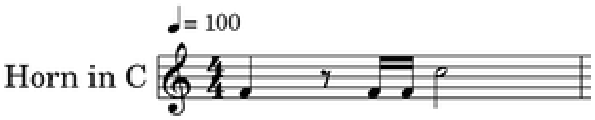
```
# Stop sounds
spkr.off()

# Stop motors
motors.enable(False)
```

***Objective 11* - FanFare!**



The final step in this project is to play a *Fanfare* tune to greet the guests.

- A call to the Park's *Director of Bands* got you a snippet of sheet music for CodeBot to play.



Notes

Sheet music is written with *notes!*

- In order to play a **note**, you just need to know the *corresponding frequency!*
- For example, **F4** is 349 Hz!

Lets play a note!

Once again, test out your new code near the top of the file!

- Right below your import statements is a good place to add the following.

Type in the Code

Write code to play an ***F4*** for ***0.4*** seconds.

- Be sure to call `spkr.off()` after playing the note.
- Also, add a `sleep(0.05)` after turning the sound off.[articulation]

```
from botcore import *
from time import sleep
from random import randrange

# Play the first note of Fanfare!
spkr.pitch(349)
sleep(0.4)
spkr.off()
sleep(0.05)
```

---

1. **Articulation gap** - gives some *separation* between notes, rather than *slurring* them. The 0.05 duration is just a guess, to give a little space between notes. To be precise you should *subtract* that from the overall note duration. But keep it simple for now!

Run It!

That was... ***NOTE worthy!***

- Now of course you *could* **copy and paste** those 4 lines a few more times, make changes, and complete the rest of the song.
- **But** even for this *short* song that would add up to **many** lines of code!
- Too bad there's not a function in `botcore` like `note(freq, duration)`.
  - A *single* line could replace all **4** of the lines above!

That function doesn't exist... ***yet!* You can define your OWN functions**

- Click on the **functions** tool above and learn the basics.
- *Then move on to test your knowlege!*

Check the 'Trek!

Define a *function* to play a single note:

- It should take 2 parameters: `def note(freq, duration):`
- **Select** your old *"note"* code and press **TAB** to indent it beneath the `def ...`
- Update `spkr.pitch()` and `sleep()` to use the arguments `freq` and `duration`.

After you **def**ine the new function, go ahead and *test* it with the first note of the *Fanfare*.

- Your program should *sound* the same as before, but now it's ready to **rock!**

Run It!

Try out your shiny new function!

- Being able to **def**ine custom functions will make it much easier for you to write more advanced programs.

CodeTrek:

```
from botcore import *
from time import sleep
from random import randrange

# Function to play a note with given frequency and duration
def note(freq, duration):
```

`note(freq, duration)` takes speaker **frequency** and sleep **duration** as arguments to *play a note!*

- It uses the code you wrote *earlier* in the objective.

You need to **define** a function ***before*** you can call it!

```
    spkr.pitch(freq)
    sleep(duration)
    spkr.off()
    sleep(0.05)
```

This *short* `sleep` will prevent the notes from *slurring*.

- If you feel like the notes aren't **distinct** enough, feel free to *increase* this value!

```

# TODO: Play the first note of Fanfare!
```

Call it just like you would `sleep` or `randrange`!

Before you turned your code into a function, you were playing ***F4*** for ***0.4*** seconds.

To **replicate** that *note*, simply call your *new* function:

- `note(349, 0.4)`

```

# Sweep LEDs, counting guests as they arrive
delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
```

```
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break

# Move forward 3 feet
motors.enable(True)
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(2.0)

# Spin 360 degrees
motors.run(RIGHT, -50)

# Play "cute sounds" while spinning
count = 0
while count < 15:
    count = count + 1

    # Choose a random frequency
    f = randrange(100, 1000)
    spkr.pitch(f)
    sleep(0.1)

# Stop sounds
spkr.off()

# Stop motors
motors.enable(False)
```

Goals:

- **Define** a function named `note` that has `freq` and `duration` as 🔧parameters.
- Play an ***F4** note* for ***0.4*** seconds by calling `note(349, 0.4)`.

Tools Found: import, Functions, Indentation, Keyword and Positional Arguments, Parameters, Arguments, and Returns

Solution:

```
from botcore import *
from time import sleep
from random import randrange

# Function to play a note with given frequency and duration
def note(freq, duration):
    spkr.pitch(freq)
    sleep(duration)
    spkr.off()
    sleep(0.05) #@1

# Play the first note of Fanfare!
note(349, 0.4)

# Sweep LEDs, counting guests as they arrive
```

```python
delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break

# Move forward 3 feet
motors.enable(True)
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(2.0)

# Spin 360 degrees
motors.run(RIGHT, -50)

# Play "cute sounds" while spinning
count = 0
while count < 15:
    count = count + 1

    # Choose a random frequency
    f = randrange(100, 1000)
    spkr.pitch(f)
    sleep(0.1)

# Stop sounds
spkr.off()

# Stop motors
motors.enable(False)
```

Objective 12 - Putting It All Together

Turning the *notes* into *Fanfare!*

Now, to **decode** the musical notation into something that you can write **Python code** for:

- There are just two ***pitches***: **F4** (349 Hz) and **C5** (523 Hz).
- Musical ***timing*** is in *"beats"*, and the table below breaks the tune into 16 *slices of time*.
    - For this tune, just make those 1/16 slices equal 0.1 seconds each.

Note	1/16 Beats	→	Frequency	Seconds
F4	4		349 Hz	0.4
rest	2		--	0.2
F4	1		349 Hz	0.1
F4	1		349 Hz	0.1
C5	8		523 Hz	0.8

Check the 'Trek!

Now to put it ALL together

- Using your new `note()` function, write the whole **Fanfare** tune.
- Use variables to *define* notes `F4` and `C5`.
- A musical **"rest"** is a *silent pause*. Simply `sleep()` for the specified duration.
- **Move the code to the *very end of your program*, where it belongs!**

Run It!

This should satisfy ***all*** the project *requirements!*

- In my version I added a *pause for effect* before the **Fanfare**.
- You may want to adjust things to your liking also!

There's **plenty** of room to *improve* this project further!

CodeTrek:

```
from botcore import *
from time import sleep
from random import randrange

# Sweep LEDs, counting guests as they arrive
delay = 0.1
n_led = 0
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break

# Move forward 3 feet
motors.enable(True)
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(2.0)

# Spin 360 degrees
motors.run(RIGHT, -50)

# Play "cute sounds" while spinning
count = 0
while count < 15:
    count = count + 1

    # Choose a random frequency
    f = randrange(100, 1000)
    spkr.pitch(f)
    sleep(0.1)
```

```

# Stop sounds
spkr.off()

# Stop motors
motors.enable(False)


# Pause for effect
sleep(0.5)

# Function to play a note with given frequency and duration
def note(freq, duration):
```

Move your function definition *down here!*

- This won't change the *functionality* of your program, it just *looks* ***better!***

```
    spkr.pitch(freq)
    sleep(duration)
    spkr.off()
    sleep(0.05)

# Define musical note frequencies
F4 = 349
C5 = 523
```

Define the **note frequencies** to make the code *more* readable.

```

# Play the Fanfare!
note(F4, 0.4)
sleep(0.2)
note(F4, 0.1)
# TODO: Play the missing note!
```

Oh dear!

A **note** is *missing!*

- Reference the *table* in the instructions to *complete the song!*

```
note(C5, 0.8)

```

Goals:

- *Define* the **notes** by assigning the respective frequencies to the following variables.
- `F4`
- `C5`
- Play ***Fanfare*** using your `note` function and the `F4` and `C5` variables!

Tools Found: Variables, Functions, Readability

Solution:

```
from botcore import *
from time import sleep
from random import randrange

# Sweep LEDs, counting guests as they arrive
delay = 0.1
n_led = 0
```

```
n_guests = 0

while True:
    leds.user_num(n_led, True)
    sleep(delay)
    leds.user_num(n_led, False)

    # Add +1 to n_led, and store result in n_led.
    n_led = n_led + 1
    if n_led == 8:
        n_led = 0

    # Count guests when BTN-0 pressed
    if buttons.was_pressed(0):
        spkr.pitch(440)
        sleep(0.1)
        spkr.off()

        # After delay, DEBOUNCE the button
        buttons.was_pressed(0)

        leds.ls_num(n_guests, True)
        n_guests = n_guests + 1

        if n_guests == 5:
            break

# Move forward 3 feet
motors.enable(True)
motors.run(LEFT, 50)
motors.run(RIGHT, 50)
sleep(2.0)

# Spin 360 degrees
motors.run(RIGHT, -50)

# Play "cute sounds" while spinning
count = 0
while count < 15:
    count = count + 1

    # Choose a random frequency
    f = randrange(100, 1000)
    spkr.pitch(f)
    sleep(0.1)

# Stop sounds
spkr.off()

# Stop motors
motors.enable(False)


# Pause for effect
sleep(0.5)

# Function to play a note with given frequency and duration
def note(freq, duration):
    spkr.pitch(freq)
    sleep(duration)
    spkr.off()
    sleep(0.05)

# Define musical note frequencies
F4 = 349
C5 = 523

# Play the Fanfare!
note(F4, 0.4)
sleep(0.2)
note(F4, 0.1)
note(F4, 0.1)
note(C5, 0.8)

```

Mission 4 Complete

It's a great feeling when a plan comes together!

- You started with an *ambitious* set of **Goals**.
- A lot of **creativity** was needed to make this ***exhibition*** a success.
- And there were a few surprises along the way! Who knew you'd learn about:
    - **Debouncing** contacts on push-button switches.
    - Making ***pulse-laser-disruptinator*** sounds.
    - Translating **sheet music** to **Python** code.
    - ...and more!

And... This is real-world stuff!

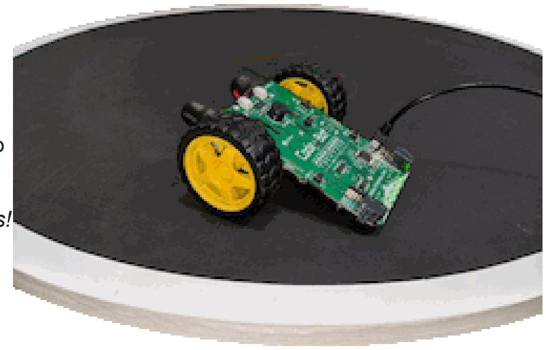
- From ***Movie FX***, to ***Art Installations***, to ***Theme Parks***, you'll find *coders* just like ***you*** making the magic happen!
- Counting button presses? How about traffic monitors with pressure switches to count traffic?
- Your *Python coding toolkit* is growing. A lot of cool applications you see every day are now within your ability to *craft with code!*

Try Your Skills

Suggested Re-mix Ideas:

- Make the **Flashing User LED sequence** sweep ***both ways***, first to the *left*, then to the *right*.
    - You could use an `if` statement inside your loop to decide whether to **increment** or **decrement** (subtract 1 from) `n_led`.
- Increase the *number* of **Guests** the exhibit can hold to **15**.
    - Display the count as a binary number on the **Line Sensor LEDs**, using `leds.ls(n_guests)`
- Make the **Audible Button Feedback Tones** *increase* to **higher frequencies** as the number of Guests increases.
- Use your `note()` function to compose and play an ***enchanting melody*** during the **Move Forward** part of the exhibit.

Mission 5 - Fence Patrol



In this project you'll gain an in-depth understanding of CodeBot's *Line Sensors*

- CodeBot has several ***sensors*** onboard, giving it the ability to *interact* with its environment.
- Those 5 high-performance 🔧Line Sensors let your Python code respond to changes as CodeBot moves across a surface.
- From *robot housekeepers* to *self-driving cars*, the **sensing** and **control** techniques you'll learn in this project apply to all kinds of *intelligent systems!*

Think about *what has driven CodeBot* in the previous projects:

- Your code used **timing** with the `sleep()` statement.
- And you've already done some **sensing**, by detecting 🔧CodeBot buttons.

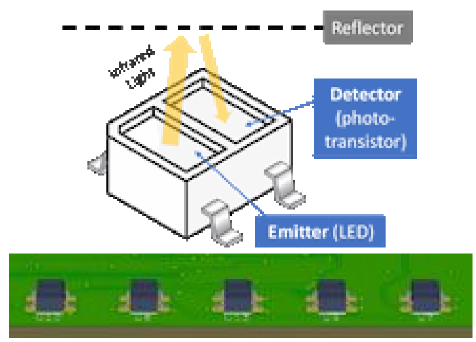
  You can still use *all* those tools, but these **sensors** really expand your 'bots abilities!

Project Goals:

- Read the 🔧Line Sensors and display the results on the green LEDs right above them.
- Use 🔧analog readings to measure the *contrast* between different surfaces.
- Make a *"contact counter"* to show each line-detect on the **User LEDs**.
- Teach CodeBot to *drive between the lines* - the **Fence Patrol** 'bot!

Objective 1 - Line Sensors Up-Close!

How do the 🔧line sensors work?



Take a look at the close-up diagram to the right:

- The **emitter** is like a flashlight, shining *invisible* light.
- The **detector** is like your *eyes* - judging how *bright* the reflection is.
- The **reflector** could be *anything*! A taped line on the floor, or any object placed near the *detector*.

The detected **brightness** level can *vary* based on:

- **Reflectivity** of the surface:
  - *Reflective* → shiny surfaces, white or light colors.
  - *Not-Reflective* → black or dark colors, empty space.
- **Distance** of the surface from the sensor.

So without further ado, on to the **API**...

Concept: *API*

Application Programming Interface

"The details of how your program interacts with different services it needs."

You've already been using APIs:

- The `motors`, `leds` and other parts of the **botcore** API.
- The `time` library is part of Python's amazing *standard library*, which offers lots of APIs to Operating System services and more!

Your code can **read** the *brightness* level of the reflected *infrared* light as an 🔧analog value with the function:

```
ls.read(num)  # Sensor 'num' can be 0, 1, 2, 3, or 4
```

- This function turns on the *emitter*, reads the *detector*, then turns the emitter back off.
- The value it returns is an 🔧integer between 0 and 4095, since the 🔧ADC (analog-to-digital) converter is 12 🔧bits resolution (2^{12} = 4096 numbers).

Create a New File!

Use the **File** → **New File** menu to create a new file called ***LineSense***.

Type in the Code

Try ***stepping through*** the following code.

- You will need to **step** through this in the debugger with the **Debug Panel** visible *and* the "Globals" dropdown **open**.
- That will allow you to see the `val` values the sensor is returning.

```
from botcore import *
while True:
    # Read line sensor 0
    val = ls.read(0)
```

Debug

Step through the code.

- Can you see a difference in `val` when you put a reflective object near the sensor?
- See how the distance of your finger from the sensor affects the reflected light?

Goals:

- Click the ☰ button at the lower-right to open the *console* panel.
- Enter **DEBUG** mode on the CodeBot by pressing the Debug Program button.
- Use the debugger **Step In** button to *step* through the code.

Tools Found: Line Sensors, Analog to Digital Conversion, int, Binary Numbers

Solution:

```
from botcore import *
while True:
    # Read line sensor 0
    val = ls.read(0)
```

Objective 2 - The Debug Console

Text messages from your *Code*

It's great to be able to use the **debugger** to inspect variables!

- But wouldn't it be nice to ***continuously display sensor values to a screen***?

Python's built-in `print()` function is made for that!

As it runs, your program can print text messages about what it's doing.

If you ***click*** on the console panel you can type-in **Python** statements directly. This is called the **REPL**.

Concept: *REPL*

"Read Evaluate Print Loop"

A name for the "command line" that languages like **Python** offer. If you were to code your own "REPL" in Python, it might look something like this:

```
while True:
    # Read a statement from the Keyboard. (press ENTER)
    # Evaluate the statement. (execute the Python code!)
    # Print the result to the Console.
```

Besides being a place to see `print()` statement *output*, the **REPL** is a great way to test out snippets of code, language features, and APIs as you decide how to use them in your code.

You'll have a chance to test some commands on the **REPL** later. For now, ***go back to your code in the Editor*** and try adding a `print()` statement to show the sensor value on the **Debug Console**!

Type in the Code

Add a `print()` statement to display `val` to the *Debug Console* each time through the loop.

```
from botcore import *
while True:
    # Read line sensor 0
    val = ls.read(0)

    # Display the sensor value to the Console
    print(val)
```

Run It!

When you run this, the values are going to *stream* by very quickly!

The `print()` statement can do a lot more than just display numbers.

- You can give it multiple arguments: strings, integers, etc.

Ex: `print("Your name is ", name, " and your address is ", address)`

Check the 'Trek!

Modify your code to add a **label** to your sensor value.

- The output lines should look something like: ***"Sensor 0 reading = 325"***

Run It!

Isn't it so much nicer to have a *description* along with all those numbers!

CodeTrek:

```
from botcore import *
while True:
    # Read line sensor 0
    val = ls.read(0)

    # Display the sensor value to the Console
    # TODO: Add a label to the printed sensor val
```

The `print` function can take *multiple* values!
Add a string as the first argument to your `print` statement.

```
print("Line Sensor 0 value = ", val)
```

Goals:

- Print the *value* of line sensor `0` to the console.

- Print an output that looks *something* like `"Sensor 0 reading = 325"` where `"325"` is the *value* of line sensor 0.

Tools Found: Print Function, Keyword and Positional Arguments, str, int

Solution:

```
from botcore import *
while True:
    # Read line sensor 0
    val = ls.read(0)

    # Display the sensor value to the Console
    print("Line Sensor 0 value = ", val)

```

***Objective 3* - Crossing the Line**

What's the threshold between *detected* and *NOT detected*?

The 🔧line sensors provide an 🔧analog value with **4096** shades of contrast!

- But your code must **decisively** detect a *boundary line*.
- The **line** can be *light* or *dark*, but it will have different *reflectivity* than the background.
- Your 'bot needs to know if it has hit the boundary **line**, `True` or `False` !

That's a 🔧boolean value you're looking for.

- You have used 🔧comparison operators to make `True` / `False` decisions.
- ...*usually* with 🔧control flow like the `if` statement below.

```
threshold = 2500
if val < threshold:
    # Detected a reflection!
```

But you can *also* assign the ***boolean* result** of a *comparison* to a 🔧variable, as in the example here:

```
threshold = 2500
is_detected = val < threshold
leds.ls_num(0, is_detected)
```

- `is_detected` is assigned the **boolean** result of the **comparison**.
- ...and the **LED** will turn **On** if `is_detected` is `True`.

Check the 'Trek!

Modify your code to **turn on** 🔧Line Sensor LEDs **0** when `val` is below a certain `threshold`.

- You should have a good idea what the value of `threshold` should be, based on your prior observations!

Run It!

Test it by passing your finger below **Line Sensor 0**.

- It's pretty cool to have a *real-time control loop!*
- Could it detect a white line against a dark background?

CodeTrek:

```
from botcore import *

threshold = # TODO: Set a threshold
```

Pick a number somewhere **inbetween** the *reflectivity* of your **line** and your **background**.

Confused?

Fear not!

Simply run the code from *last* objective and stick your finger under the line sensor.

My 'bot reads a value of 3859 when my finger is **off** the sensor,
and 229 when it's **on** the sensor.

So I would pick a threshold of 2000!

```
threshold = 2000
```

```

while True:
    val = ls.read(0)
    is_detected = val < threshold
```

is_detected will return True **if** val is **less than** threshold.

```
    leds.ls_num(0, is_detected)
```

Goals:

- Set the variable threshold to a number somewhere **inbetween** the *reflectivity* of your **line** and your **background**.
- Set the variable is_detected to return True **if** val is **less than** threshold.
- Call leds.ls_num(0, is_detected)

Tools Found: Line Sensors, Analog to Digital Conversion, bool, Comparison Operators, Branching, Variables, CodeBot LEDs

Solution:

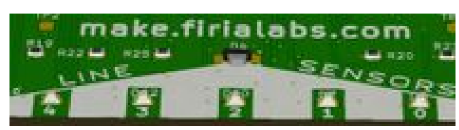
```
from botcore import *

threshold = 2000

while True:
    val = ls.read(0)
    is_detected = val < threshold #@2
    leds.ls_num(0, is_detected)
```

Objective 4 **- (Fun)ctions**

Take a look at the front edge of CodeBot - *sensors* on the bottom, *LEDs* on the top.

Those LEDs are positioned so you can use them to indicate Line Sensors **detection**.

- You have ***LS LED 0*** working already.
- The goal of this step is to invite the rest of the **Line Sensors** to your *LED party*!



You could achieve this without much thought by *copying and pasting* what you have, *BUT*...

Concept: *Don't Repeat Yourself (DRY)*

Here is ancient coding wisdom:

> ***Never write the same code twice.***

Okay, alright, a little *repetition* isn't awful, *but* if you find yourself typing the same code over and over, just think how much work it will be to **change** it (or fix a **bug** in it) in the future.

> Instead, let your *programming tools* (like functions) do the work!

Challenge Accepted? *Excellent!*

▷ Run It!

CodeTrek:

```
from botcore import *

def detect_line(n):
```

> *Aww yeah*, your first 🔧function!
>
> You can run this **any** time by calling it's name!

```
    # TODO: Read Line sensor value `n`

    is_detected = val < threshold
    # TODO: Set LS LED `n`
```

> You'll want to light up the 🔧LED associated with the 🔧line sensor you're reading.
>
> *But remember*, you're not reading sensor `0` anymore!
>
> - You'll need to supply the variable `n` as the LED index argument!
>
> ```
> leds.ls_num(n, is_detected)
> ```

```

threshold = 2000

while True:
    detect_line(0)
```

Goals:

- ***Define*** a function named `detect_line` and **call** it.
- *In* the function `detect_line`:
- Read the 🔧line sensor at index `n`.
- *In* the function `detect_line`:
- If the `threshold` is crossed, light the LS 🔧LED at index `n`.

Tools Found: Line Sensors, Functions, Locals and Globals, LED

Solution:

```
from botcore import *

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)

threshold = 2000

while True:
    detect_line(0)
```

Objective 5 **- Line Sensor Magic Lights!**

Your next step is to scan *all* the sensors.

- Are you thinking something like this?

```
detect_line(0)
detect_line(1)
detect_line(2)
detect_line(3)
detect_line(4)
```

Not so fast!

That's a **lot** of *repetition*. Not very ***DRY***!

- You know how to make a loop to count from **0 to 4**.
- And you could package it in a new function.
    - Then you'd scan **all** the *Line Sensors* with one line of code!

After all, *scanning the sensors* is just **one** of the things your 'bot will be doing in the ***Fence Patrol*** project!

Check the 'Trek!

Write another function `def scan_lines():` to:

- Loop through all **5** line sensors.
- Call `detect_line(n)` for each of them.

Click on the loop tool for a hint on making a loop to count from **0 to 4**.

Now your code will have **two** functions defined.

Run It!

All your **Line Sensor LEDs** should be tracking the **sensors** below!

- Can you see them track your finger as you pass it below the sensors?

CodeTrek:

```
from botcore import *

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)

def scan_lines():
```

`scan_lines()` calls `detect_line(n)` for *each* of the **5** line sensors.

```
    # Loop across all Line Sensors and 'detect'
    n = 0
    while n < 5:
```

Loop from `0` to `4`!

- You'll **increment** *(add **1** to)* `n` *each* repitition.
- The while loop `condition, n < 5`, will be met after **5** repititions!

```
        # TODO: Call detect_line
```

Each repitition of the loop, `n` will have a *different* value between `0` and `4`.

- *Therefore,* we need to supply `n` as the argument for `detect_line`!

`detect_line(n)`

```
        # TODO: Increment n
```

Add `1` to `n` *every* repitition.

- `n = n + 1`

```

threshold = 2000

while True:
    scan_lines()
```

Call your *new* function!

Goals:

- **Define** a function called `scan_lines()`.
- Use a while loop with the `condition n < 5`.
- **In** the loop `while n < 5`:
- Call `detect_line(n)`
- Increment `n` by assigning `n = n + 1`
- Call `scan_lines()`

Tools Found: Loops, Functions, Line Sensors, Keyword and Positional Arguments

Solution:

```
from botcore import *

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)

def scan_lines():
    n = 0
    while n < 5:
        detect_line(n)
        n = n + 1

threshold = 2000

while True:
    scan_lines()
```

Quiz 1 - Checkpoint

Question 1: Using **more reflective** objects, or moving them **nearer** to the sensor makes the `ls.read(0)` values:

- ✓ Decrease
- ✗ Increase
- ✗ Stay the same

Question 2: How could you make your program detect a **dark** line against a **light** background?

✓ Use ">" instead of "<" in the comparison.

✗ Set "threshold" to a higher value

✗ Use a different LED function.

Question 3: What does the *acronym* **DRY** stand for?

✓ Dont Repeat Yourself

✗ Design Reference Year

✗ Defensive Rushing Yards

Objective 6 - Defensive Driving

Your **sensors** are tuned up and ready!

- Now it's time to plan for adding 🔧motors into the fun.
- The goal of this step is to develop the **algorithm** for your ***Fence Patrol*** robot.

You will need a small area to run your 'bot, with a **boundary line** that contrasts with the surface.

- *Electrical Tape* works well for making **lines**.

Before you get moving, check out these ***pro-tips*** -

Caution: *Safe Driving!*

Note 1: Nice robots *wait* before moving. It's very bad manners for a 'bot to jump right off your desk the instant you **run** the code.

- Always include a 🔧loop waiting for a **button press** *(or other human-initiated action)* ***before*** moving.

Note 2: CodeBot *safety-stop*

- If your code stops due to an error, *or*
- You press **Stop** in *CodeSpace*,
- → **CodeBot will disable the motors automatically!**

Check the 'Trek!

Add code to **Wait for the user to press BTN-0**.

- Insert the code at the top of your file, just after the `import` section:
- Click on the 🔧Loops tool if you need a reminder of how to break out of a loop!
- You already have a lot of experience using 🔧conditions and 🔧CodeBot Buttons.

Run It!

Make sure your **Line Detect** code doesn't run ***until*** you press **BTN-0**.

CodeTrek:

```
from botcore import *

while True:
    # TODO: Break out if BTN-0 was pressed
```

I hope you *remember* this from last mission!

Use `buttons.was_pressed(0)` to *detect* BTN-0 being pressed.

Use `break` to exit the `while` loop!

```
if buttons.was_pressed(0):
    break
```

```

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)

def scan_lines():
    n = 0
    while n < 5:
        detect_line(n)
        n = n + 1

threshold = 2000

while True:
    scan_lines()


```

Goal:

- Add a `while` 🔧loop at the beginning of your program that `break`s when the user hits BTN- 0.

Tools Found: Motors, Loops, bool, Buttons

Solution:

```
from botcore import *

while True:
    if buttons.was_pressed(0):
        break

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)

def scan_lines():
    n = 0
    while n < 5:
        detect_line(n)
        n = n + 1

threshold = 2000

while True:
    scan_lines()


```

Objective 7 - Driving in Bounds

Now to get your *algorithm* on!

So here's the **Fence Patrol** *algorithm*:

1. Scan the line sensors.
2. If **any** sensor detects a line, **back up and turn**.
3. Else if **no** line detected, **drive forward**.
4. Repeat forever, from step 1.

Check the 'Trek!

You need to make a couple of changes to your code, so that when you *"Scan the line sensors"* you'll know if a line was ***HIT***.

1. Modify your `detect_line(n)` function so that it returns a value: *"Was a line detected or not?"*
2. Modify your `scan_lines()` function so that it too returns a value: *"Were **ANY** lines detected in this* scan*?"*
3. To show that it's working, add a `line_count` global variable that you update in your main loop each time `scan_lines()` returns `True`.

    Display `line_count` in binary on the **User LEDs**.

Run It!

Are you satisfied that `scan_lines()` is properly reporting that *CodeBot* has hit a line?

- Press **BTN-0** to start the action.
- Watch your **USER LEDS** to see the count.
- What happens when a line is detected ***continuously?***
- Something *interesting* happens when `line_count` reaches `256`!

CodeTrek:

```
from botcore import *

while True:
    if buttons.was_pressed(0):
        break

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)
    # TODO: Return is_detected
```

`detect_line(n)` needs to be **modified** to return a value that answers the question:

"Was a line detected or not?"

Simply **return** `is_detected`.

```
return is_detected
```

```

def scan_lines():
    # Loop across all Line Sensors and 'detect'.
    # Return True if ANY line is detected!
    got_line = False
```

`scan_lines()` needs to be modified to answer the question:

- *"Were **ANY** lines detected in this scan?"*

Use `got_line` to keep track of whether ***any*** `detect_line(n)` call returns `True`.

- got_line will be **returned** at the *end* of the 🔧function!

```
    n_sens =  0
    while n_sens < 5:
        # Use the return value of detect_line()
        if detect_line(n_sens):
            got_line = True
```

We got one!

- When a line is detected, update `got_line` to `True`!

```

        n_sens = n_sens + 1

    # Always return True or False.
    return got_line
```

Don't forget to `return got_line`!

```

threshold = 2000
line_count = 0
```

Initialize a *new* 🔧variable `line_count`.

- It's used to keep track of *how many times* `scan_lines()` returns `True`.

```

while True:
    hit = scan_lines()
```

Now, `scan_lines()` returns `True` if a line was detected on ***any*** of the **5** 🔧line sensors.

```

    if hit:
        # Update count and display on User LEDs
        line_count = line_count + 1
        leds.user(line_count)
```

When a line is `hit`, update `line_count` and *display* the ***new total*** on the 🔧user LEDs!

```



```

Goals:

- 🔧Return `is_detected` from the `detect_line(n)` 🔧function.
- **Return** `got_line` from the `scan_lines()` 🔧function.
- Call `leds.user(line_count)`.

Tools Found: Parameters, Arguments, and Returns, Locals and Globals, Binary Numbers, Functions, Variables, Line Sensors, CodeBot LEDs

Solution:

```
from botcore import *

```

```python
while True:
    if buttons.was_pressed(0):
        break

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)
    return is_detected

def scan_lines():
    # Loop across all Line Sensors and 'detect'.
    # Return True if ANY line is detected!
    got_line = False
    n_sens =  0
    while n_sens < 5:
        # Use the return value of detect_line()
        if detect_line(n_sens):
            got_line = True

        n_sens = n_sens + 1

    # Always return True or False.
    return got_line

threshold = 2000
line_count = 0

while True:
    hit = scan_lines()

    if hit:
        # Update count and display on User LEDs
        line_count = line_count + 1
        leds.user(line_count)


```

Objective 8 **- Flicker Begone!**

Bugs Ahead

Is it just me, or are your *User LEDs* flickering like crazy too?

- They are actually **counting up** in 🔧binary.
- After the count reaches 0b11111111 (255)... ***BOOM!***

See below for a suggested fix for this bug.

Check the 'Trek!

Change your code to fix the **ValueError** bug.

- When `line_count` reaches **256**, set it back to 0b00000000 (0).
- It will resume counting up from the zero!

Run It!

- Make sure you see the counter *wrap around to zero* and count up from there.
- In the next step you'll ***slow the count down.***

Be sure to verify that no matter which *Line Sensor* detects the line, you see *User LED*s counting up.

CodeTrek:

```python
from botcore import *

while True:
```

```
    if buttons.was_pressed(0):
        break

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)
    return is_detected

def scan_lines():
    # Loop across all Line Sensors and 'detect'.
    # Return True if ANY line is detected!
    got_line = False
    n_sens =  0
    while n_sens < 5:
        # Use the return value of detect_line()
        if detect_line(n_sens):
            got_line = True

        n_sens = n_sens + 1

    # Always return True or False.
    return got_line

threshold = 2000
line_count = 0

while True:
    hit = scan_lines()

    if hit:
        # Update count and display on User LEDs
        line_count = line_count + 1
        # TODO: When line_count reaches 256, set it to 0
```

To *prevent* another **error**, `if line_count is 256`, set it to `0`!

```
if line_count == 256:
    line_count = 0
```

```
        leds.user(line_count)


```

Goal:

- When `line_count == 256`, set it to `0`.

Tools Found: Binary Numbers

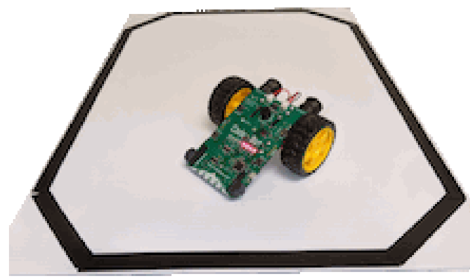
Solution:

```
from botcore import *

while True:
    if buttons.was_pressed(0):
        break

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)
    return is_detected

def scan_lines():
    # Loop across all Line Sensors and 'detect'.
    # Return True if ANY line is detected!
    got_line = False
    n_sens =  0
    while n_sens < 5:
```

```
        # Use the return value of detect_line()
        if detect_line(n_sens):
            got_line = True

        n_sens = n_sens + 1

    # Always return True or False.
    return got_line

threshold = 2000
line_count = 0

while True:
    hit = scan_lines()

    if hit:
        # Update count and display on User LEDs
        line_count = line_count + 1
        if line_count == 256:
            line_count = 0
        leds.user(line_count)

```

Objective 9 - Fence Patrol v1.0

Ready to Motor Up?



- You've designed your **algorithm**.
- You have a *"Wait for button-press"* safety feature.
- And your sensor code has been *fully tested*.

All systems are go!

> **Caution**
>
> Be sure your threshold calculation is set for your line type.
>
> - Dark line on light surface (as shown above): `is_detected = val > threshold`
> - Light line on dark surface: `is_detected = val < threshold`

Reviewing your algorithm:

1. Scan the line sensors.
2. If **any** sensor detects a line, **back up and turn**.
3. Else if **no** line detected, **drive forward**.
4. Repeat forever, from step 1.

Final Steps:

- Add **two new functions** to ***drive the motors***:
  - `go_forward()` and `back_turn()`.
- Call those functions from your main `while True:` loop.
  - Based on the return value of `scan_lines()` of course!

> **Check the 'Trek!**
>
> You already know how to run the motors. Take a look back at your ***Time and Motion*** mission code if you need a refresher!
>
> - Be sure to **enable** the motors before your main loop.
> - Remember to do `from time import sleep` so you can *delay* while backing up and turning.
> - **Use at most 50% motor power to begin with!**

> **Run It!**
>
> ***You need batteries loaded for this!***

- Place your 'bot inside the **boundary** and *press BTN-0*.
- Is CodeBot staying in-bounds?
- **Hey** - With the delay in `back_turn()` you can **read the line count on the User LEDs!**

Try Your Skills: *Customize your Code*

Any *mods* you'd like to try?

- Your initial *guess* at the **speeds** and **delays** in `back_turn()` could probably use some adjustment to make your bot cover more ground.
- Are you feeling the ***need for speed?***
  - How high can your `go_forward()` function set the motor **power** to? Can you go **100%** *full-throttle* and still *read sensors* fast enough to stay in-bounds?
  - **Note:** You'll need to increase the *braking power* also! ...Watch those wheels *spin-out* when you `back_turn()`!

Enjoy!

CodeTrek:

```
from botcore import *
from time import sleep

while True:
    if buttons.was_pressed(0):
        break

def go_forward():
    # Start driving forward (no delay/stop needed)
    # TODO: code to run both motors at same (+) speed

def back_turn():
    # Back up a bit and turn around
    # TODO: code the following steps -
    # 1. Run both motors at same (-) speed (reverse).
    # 2. Sleep for a bit, backing up.
    # 3. Run motors in opposite directions.
    # 4. Sleep for a bit, rotating.
```

Don't be afraid!

You've already **mastered** each of the 4 steps here in the **Time and Motion** mission!

- If you get stuck, review your old code!

```

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)
    return is_detected

def scan_lines():
    # Loop across all Line Sensors and 'detect'.
    # Return True if ANY line is detected!
    got_line = False
    n_sens =  0
    while n_sens < 5:
        # Use the return value of detect_line()
        if detect_line(n_sens):
            got_line = True

        n_sens = n_sens + 1

    # Always return True or False.
    return got_line

threshold = 2000
```

```
line_count = 0

# Enable the motors
motors.enable(True)

while True:
    hit = scan_lines()

    if hit:
        back_turn()
```

`if scan_lines` **hits** a line, call your *new* function `back_turn`!

```

        # Update count and display on User LEDs
        line_count = line_count + 1
        if line_count == 256:
            line_count = 0
        leds.user(line_count)
    else:
        go_forward()
```

`if scan_lines` **does not** find a line, call your *new* `go_forward` function!

Goals:

- *Define* `go_forward` and `back_turn` functions.
- When `scan_lines` **hits**, call `back_turn()`.
- When `scan_lines` does *NOT* **hit**, call `go_forward()`.

Tools Found: Motors, Line Sensors, Functions

Solution:

```
from botcore import *
from time import sleep

while True:
    if buttons.was_pressed(0):
        break

def go_forward():
    # Start driving forward (no delay/stop needed)
    motors.run(LEFT, 50)
    motors.run(RIGHT, 50)

def back_turn():
    # Back up a bit and turn around
    motors.run(LEFT, -50)
    motors.run(RIGHT, -50)
    sleep(0.1)
    motors.run(LEFT, -50)
    motors.run(RIGHT, 50)
    sleep(0.1)

def detect_line(n):
    val = ls.read(n)
    is_detected = val < threshold
    leds.ls_num(n, is_detected)
    return is_detected

def scan_lines():
    # Loop across all Line Sensors and 'detect'.
    # Return True if ANY line is detected!
```

```
    got_line = False
    n_sens =  0
    while n_sens < 5:
        # Use the return value of detect_line()
        if detect_line(n_sens):
            got_line = True

        n_sens = n_sens + 1

    # Always return True or False.
    return got_line

threshold = 2000
line_count = 0

# Enable the motors
motors.enable(True)

while True:
    hit = scan_lines()

    if hit:
        back_turn() #@1

        # Update count and display on User LEDs
        line_count = line_count + 1
        if line_count == 256:
            line_count = 0
        leds.user(line_count)
    else:
        go_forward()  #@3

```

Mission 5 Complete

This project has covered a lot of ground.

- Analog sensors - you've mastered a *classic* non-contact sensor, used in many **Industrial** and **Commercial** products!
- Using **threshold** comparison operations to make decisions with sensor *data*.
- Working with the *Debug Console*, including a powerful way to experiment with sensor values using Python's `print()` statement.
- Building *safety features* into your products, so they don't surprise the user (or worse!) on startup.
- ***AND*** you have now built a truly **autonomous** robot.
    - Your 'bot makes *decisions* and takes *action* based on *perception*.
    - It's sensing its environment, *not* just running a pre-programmed sequence or being remote-controlled!

Code like this impacts your life *every day!*

- Automatic Guided Vehicles (**AGV**s) use this kind of code to *zoom* around warehouse distribution centers, getting packages to you!
- Robots are used to clean up environmental waste, explore underground mines, and discover shipwrecks in the deepest oceans.
- *Maybe **you** will invent the next amazing application of this technology!*

Try Your Skills

***Mission 6* - Line Follower**

Follow the Road!

Self-driving cars, autonomous flying drones, and other computing systems that *navigate on their own* have some basic principles in common.

Whether you are writing code for a vehicle with a high-powered *vision processing* system or for CodeBot's efficient *low-power sensors*, you'll face many of the same challenges to achieve the objective:

- Based on **sensor inputs**, what **actions** should you take to *stay on the path*?

In this project you will build and refine a *Line Following Robot*.

Line Followers are a staple of *Robotics competitions*, used in challenges such as:

- **Races** - which robot can make the best time navigating a curvy track?
- **Mazes** - A maze of lines is laid out on a table or floor.
  - Your 'bot has *one* run through to "learn" the maze, and a *second* run to solve it at *high speed!*

But even *basic* line followers aren't just for competitions!

- Robots that zip through warehouse *distribution centers* often follow **lines** as they pick and pack items you order when shopping online!

Project Goals:

- Create a basic line follower using 2 edge sensors.
- Improve the design with a center-line sensor to keep it straight.
- Use all 5 line sensors for *proportional* steering control.
- Adapt to your environment with *Line Calibration* code.

***Objective 1* - Speedy Sensing!**

Your **line follower** 'bot will need to continuously check for the *presence* of a line beneath **all 5** sensors.

- You already know how to read the line sensors with `ls.read(n)`.
- And you can compare against a **threshold** value to get a 🔧bool status: *is_detected*.

  That's what your `detect_line(n)` function did in the last project!

But now you need to write code that runs the following sequence:

1. Detect lines on *all 5 sensors* → store the *is_detected* value for each line.
2. Decide how to steer based on the 🔧bool values you stored.
3. Repeat!

In **step 1** of the above *algorithm* you need to ***store*** the values, since **step 2** will need them for 🔧comparison and 🔧control flow. Checking the sensors ***just once per loop*** is key if you want your 'bot to be ***Fast!***

> *Reading* the line sensors **takes time**, and if your *control loop* is too slow then your 'bot will have to *slow down* to stay on track.

Here's some code that could work: *(don't type this in)*:

```
# Read all the sensors one time
line0 = detect_line(0)
line1 = detect_line(1)
line2 = detect_line(2)
line3 = detect_line(3)
line4 = detect_line(4)

# Use line values to steer the bot
# TODO: ...
```

The code above has a *lot* of **repetition** right?

It sure would be nice if there was a better way to deal with **lists** of things...

Concept: *list*

A list is a sequence of items you can access with an **index**.

Click on the list tool to learn more details.

Example:

```
# Create a list of musical frequencies
song = [392, 440, 349, 175, 262]
spkr.pitch(song[0])  # Play the first pitch in the list
```

Remember:

- Use **square brackets** `[ ]` to create a list
- Access items by their **index** (an integer in brackets).
- The first **index** is ***zero***, not *one*!

Make a fast function

- With lists in your toolbox, you can make a function that checks the line sensors and returns *is_detected* for **all** of them.

Create a New File!

Debug

Step through your code with the debugger, and make sure it works as expected.

Is your **line** being detected?

- Notice that the list is displayed by **index** 0,1,2,3,4.
  - But the line sensors are numbered 4,3,2,1,0.
- Don't worry: `detected[0]` is **line sensor 0**
  - The picture above shows how they *line up*.

CodeTrek:

```
from botcore import *

def check_lines(thresh):
    # Create a list for 5 sensors,
    # initialize to False: "not detected".
    detected = [False, False, False, False, False]
```

Your first list!

Each **item** in the list represents the *threshold* to *sensor reading* **comparison** for each line sensor.

```

    n_sens = 0
    while n_sens < 5:
```

Look familiar?

This section of the code is *just like* your `scan_lines()` function from last mission!

If you've forgotten, *go back* and check it out!

```
        val = ls.read(n_sens)
```

```
        # Compare sensor reading to threshold
        if val < thresh:
            # Line detected!
            # TODO: Set this indexed item in list to True.
```

When a **line** is *detected*, update the `detected` list!

Each **item** in the `detected` list corresponds with a line sensor.

- If sensor `0` is `True`, item `0` needs to be `True` as well!

To update a *specific* value in a list, **access** the item by it's *index!*

```
detected[n_sens] = True
```

```
        n_sens = n_sens + 1

    # Return the list
    return detected

# Test: call the function
vals = check_lines(2500)
```

Goals:

- **Define** a function named `check_lines()`.
- Create a list named `detected`.
- Update an **item** in the `detected` list by **indexing** it.
- **DEBUG** your program and use the **STEP IN** button.

Tools Found: bool, Comparison Operators, Branching, list, int, Functions, Line Sensors

Solution:

```
from botcore import *

def check_lines(thresh):
    # Create a list for 5 sensors,
    # initialize to False: "not detected".
    detected = [False, False, False, False, False] #@1

    n_sens = 0
    while n_sens < 5: #@2
        val = ls.read(n_sens)
        # Compare sensor reading to threshold
        if val < thresh:
            # Line detected!
            detected[n_sens] = True
        n_sens = n_sens + 1

    # Return the list
    return detected

# Test: call the function
vals = check_lines(2500)
```

Objective 2 - Using the REPL

Run the code from the previous step!

It doesn't do much right now...

- The program ends quickly, since it only calls `check_lines(2500)` one time.
- But wait, *there's more to the story...*

Although your program has *ended*, ***CodeBot can still accept commands!***

You have used the **Debug Console** to `print()` values. Now you'll be using its most powerful feature:

- The **REPL** *(Read Evaluate Print Loop)* lets you *interact* with CodeBot's *Python environment*.

Try this!

- Click the ☰ button at the lower-right to open the *console* panel.
- Click in the **Debug Console** panel, where you see the **>>>** prompts.

If you press **Enter** on your keyboard you'll see a new **>>>** prompt. **You can type any valid *Python* statement here.** *It's a great way to test out snippets of code!*

Concept: *CB2 vs CB3 REPL Difference*

The **REPL** works a little differently on the CodeBot model CB2 vs the CodeBot model CB3.

- On the CB2 your program state remains after your code finishes.
  - You can still call any local functions or access any local variables from your program.
- On the CB3 your program state is cleared when your code finishes.
  - You must `import` modules to use them, like `from botcore import *`
  - You can also `import` your program using the module name `main`.

Give it a try!

Try Your Skills: *Experiment with the REPL*

Hint:

- Run into an **ImportError**?

Fear not!

Usually this can be fixed by re-running the code from the *previous* objective.

Good luck!

Goal:

- Run the `check_lines()` function in the **REPL**.
- **CodeBot CB3** users need to call `from main import check_lines` *first!*

Tools Found: import, Parameters, Arguments, and Returns, Functions, CodeBot LEDs

Solution:

N/A

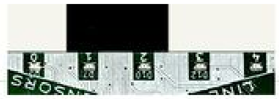
Quiz 1 - Check My Lines?

You're packing a lot of Python knowledge!

- Hopefully ***you*** can help clear a few things up about this program :-)

Here's what happened when I ran my `check_lines()` code in the debugger:

> *Note:* My 'bot is positioned on a *black* line against a *white* background as shown.



LOCAL VARIABLES	index: 0 1 2 3 4	
detected	[True, False, False, True, True]	<list>
n_sens	4	<int>
thresh	2500	<int>
val	1764	<int>

- An obvious problem above is that the `detected` values are ***inverted***.
  - *Each value is the **opposite** of what it should be for my **black line**.*

Please answer the following questions based on the program ***at the point shown in the LOCAL VARIABLES panel above***:

Question 1: What is the value of `detected[0]`?

✓ `True`

✗ `False`

✗ `0`

✗ `1`

Question 2: What is the value of `detected[n_sens]`?

✓ `True`

✗ `False`

✗ `0`

✗ `1`

***Question 3:* Assume you continued *stepping* through this program...**

What value would `n_sens` have when the program reaches the `return` statement ***after the loop***?

✓ `5`

✗ `4`

✗ `0`

✗ `[False, True, True, False, False]`

Question 4: Only sensors **1** and **2** are **on** the black line.

- The 🔧bool values are *inverted*!
- You need to **modify the code** so that:
  - `detected[1]` and `detected[2]` are `True`.
  - ...the rest are `False`

With this *modification* the **detected** *list* would be:

```
[False, True, True, False, False]
```

Which of the following is a way you could ***modify the code*** to achieve that?

✓ Change comparison to `if val > thresh:`

✗ Pass-in a lower value for `thresh`, like `check_lines(150)`

❌ Change comparison to `while n_sens > 5:`

***Objective 3* - Magic Lights Redux**

Check the 'Trek!

Run It!

Take your new ***Magic Lights*** for a spin!

- Your 🔧loop is *really* simple now.
- That's good! Now you can work with the ***whole group*** of sensors together.

Now that you know how to deal with sensor values in 🔧lists, you're ready to **unlock** some of CodeBot's more powerful ***APIs***.

- *And* you can keep your *algorithms* nice and tidy, like the 🔧loop above!

CodeTrek:

```
from botcore import *

def check_lines(thresh):
    # Create a list for 5 sensors,
    # initialize to False: "not detected".
    detected = [False, False, False, False, False]

    n_sens = 0
    while n_sens < 5:
        val = ls.read(n_sens)
        # Compare sensor reading to threshold
        if val < thresh:
            # Line detected!
            # Set this indexed item in list to True.
            detected[n_sens] = True
        n_sens = n_sens + 1

    # Return the list
    return detected


while True:
    # TODO: Check the line sensors.
```

Simply call your function `check_lines()` and assign the output to a 🔧variable!

We'll use the **value** of the variable to light the 🔧LEDs in the *next* line!

- Don't forget to supply `check_lines()` with a `threshold` argument (in my case I'm using `2500`)!

```
vals = check_lines(threshold)
```

```
    # TODO: Set LS LEDs with detected vals.
```

Here's where the *magic* happens!

Use the 🔧variable assigned on the **line above** as an argument to `leds.ls()`.

```
leds.ls(vals)
```

Goals:

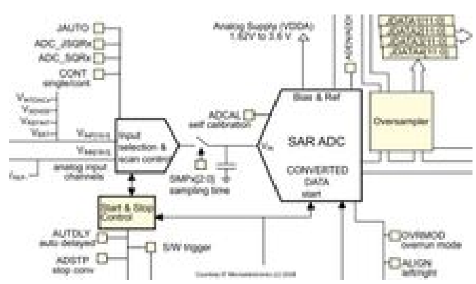
- Assign the **output** of `check_lines(threshold)` to the 🔧variable `vals`.
- Use the **variable** `vals` as the **input** to `leds.ls(vals)`.

Tools Found: CodeBot LEDs, list, bool, Binary Numbers, Loops, Variables, LED

Solution:

```
from botcore import *

def check_lines(thresh):
    # Create a list for 5 sensors,
    # initialize to False: "not detected".
    detected = [False, False, False, False, False]

    n_sens = 0
    while n_sens < 5:
        val = ls.read(n_sens)
        # Compare sensor reading to threshold
        if val < thresh:
            # Line detected!
            # Set this indexed item in list to True.
            detected[n_sens] = True
        n_sens = n_sens + 1

    # Return the list
    return detected


while True:
    vals = check_lines(2500)
    leds.ls(vals)


```

Objective 4 - Down to the Metal!



Oh, *just one more thing* before you start writing *actual* **Line Follower** code.

- Seriously, I'm not just *stalling* here... ***You're gonna want this!***
- The diagram to the right shows a *portion* of the 🔧ADC **hardware** inside CodeBot's 🔧CPU.
- That hardware can be configured to **scan** multiple inputs ***fast!***
    - ...***Way*** *faster* than your `check_lines()` code can possibly run.

Taking `check_lines()` to the ***hardware level***

- Can you write **Python** code to directly access all that ***sophisticated hardware ADC power?***
    - Yes! *...but that's for a more advanced course.*
- Writing code that controls **hardware** *directly* is called **"programming down to the metal"**. You are doing quite a lot of that already!
- Fortunately the **botcore** library provides some pre-coded functions for the 🔧Line Sensors that take advantage of all that 🔧ADC hardware.

Concept: *ls.check()*

This **botcore** 🔧Line Sensors function is very similar to the `check_lines()` function you have just developed.

Usage Example:

```
from botcore import *

thresh = 2500  # Set threshold
is_reflective = False  # Set for black line

# Check line sensors and return bools
vals = ls.check(thresh, is_reflective)

leds.ls(vals)
```

As you can see from the code above, `ls.check()` can easily replace your `check_lines()` function.

How is it different?

- It has a second parameter `is_reflective` that controls whether **"detected"** means the **sensor is** `> thresh` or `< thresh`.
- It 🔧returns a 🔧tuple rather than a 🔧list.
    - A **tuple** is basically a *read-only* form of **list**.
- It's ***screaming fast*** since it uses the ADC hardware *channel scanning* feature.
- Due to the *sampling method* used, the ADC value will be different than `ls.read()`.
    - That means you need to ***calculate a new*** `thresh` ***value!***
- You can get a 🔧tuple of the **raw ADC values**:
    - Use *zero* for the threshold value, `ls.check(0)`. *(no 2nd argument needed)*

Once more to the REPL

A quick way to find the right **threshold** value for your *Line Follower* is to enter `ls.check(0)` on the REPL.

- After you see the range of **raw values** for your *line/ground* areas, you can test it out with `ls.check(thresh, is_reflective)`.

Try Your Skills: *Debug Console*

Open the **Debug Console** and experiment to find the `thresh` value that works well with `ls.check()`.

Notes:

- If code is *already running* you will need to press the ■ **Stop** button first!
- The running code *already* did `from botcore import *` so you have access to `ls.check()` from the REPL.
    - If you *did not* have code already loaded, you'd need to type `from botcore import *` on the REPL to bring it in!

Pro-Tip: Press ↑ *Up-Arrow* to recall a prior command. *Save some typing!*

Create a New File!

Use the **File** → **New File** menu to create a new file called ***LineFollow1***.

Check the 'Trek!

- Yes, this is *finally* the start of something that *moves!*
- Your first step is to code the **most magical version yet** of ***Magic Lights***.

Now that you have `ls.check(thresh, is_reflective)` *and* `leds.ls(vals)` this should be a pretty short program!

> *The algorithm is exactly the same as your previous step*

Run It!

Surprise!

- Okay, maybe it *appears* the same to the *casual observer*...
- But now you're rolling with ***the metal!***

CodeTrek:

```
from botcore import *

thresh = 2500  # Set threshold
is_reflective = False  # Set for black line

while True:
    # TODO: Check the line sensors
```

Use the *new* `ls.check()` 🔧function to **set** the 🔧variable `vals` this time!

```
vals = ls.check(thresh, is_reflective)
```

```
    leds.ls(vals)


```

Goal:

- Assign the **output** of `ls.check(thresh, is_refective)` to the 🔧variable `vals`.

Tools Found: Analog to Digital Conversion, CPU and Peripherals, Line Sensors, Parameters, Arguments, and Returns, tuple, list, Variables, Functions

Solution:

```
from botcore import *

thresh = 2500  # Set threshold
is_reflective = False  # Set for black line

while True:
    vals = ls.check(thresh, is_reflective)
    leds.ls(vals)


```

Objective 5 - Between the Edges

Your first **line following** algorithm will be simple:

- Use only **two** sensors: ***LEFT*** (0) and ***RIGHT*** (4).
- Start your bot *"straddling"* the line.
- Move forward until a **LEFT or RIGHT** sensor is hit.
- Turn to get back on track!
    - Hit **LEFT** sensor: *turn **LEFT***
    - Hit **RIGHT** sensor: *turn **RIGHT***

Your already have a loop checking the 🔧line sensors. Just a *few more lines of code* to control the 🔧motors and you'll have your first *line follower!*

Check the 'Trek!

Modify your code to do the following:

Algorithm:

1. Wait for *BTN-0* press before enabling motors.
2. Check 🔧line sensors with `vals = ls.check(thresh, is_reflective)`.
3. Show sensor **vals** on *Line Sensor LEDs*.
4. If **left** edge is hit, turn *LEFT*.
5. If **right** edge is hit, turn *RIGHT*.
6. Otherwise go straight ahead!
7. Repeat from step 2.

Be sure to make your **SPEED** easy to adjust. *(use a 🔧variable)*

Run It!

Try this code on a few different ***line courses***. Start with a ***low* speed** before you turn up the *power*.

Experiment with Line Types:

- At a low speed, try different **turn angles** and **curves**.
- How does it handle a **gap** in the line?
- What about a **crossroad**?

Experiment with your Code:

- Try increasing the **SPEED**.
- Change the ***Turn*** code.
  - If your course has *gentle curves* then one wheel can just be a little slower than the other.
  - But if you have *tight turns* then one wheel may need to spin **backwards**. *(negative 'speed' number)*

Not too bad!

- Your 'bot can navigate some pretty *twisty* roads...
- Even with a very **simple** *algorithm* and only **two** sensors!

But it has limitations too. *Imagine this is a "package delivery robot" inside a factory...*

- There are sharp turns in some places, and this 'bot gets lost!
- It's too slow - why can't it go much faster when the path is straight?

Concept: *Take Note*

Test your **Line Follower** and make some notes:

- What situations does this algorithm handle nicely?
- What situations make it **fail**?
- ***Why*** does it *fail* in those cases?

CodeTrek:

```
from botcore import *

# Wait for BTN-0 before moving
while True:
    if buttons.was_pressed(0):
        break
```

Waiting for *BTN-0* press is good practice when your 'bot is about to **move!**

It prevents any *unexpected* cable pulls!

```

motors.enable(True)

# Set speed in one place so you can change it easily
SPEED = 30

while True:
    # Check line sensors: set for black line
    vals = ls.check(2500, False)

    # Show results on LS LEDs
    leds.ls(vals)

    # Keep the line between the edges
    if vals[0]:
        # Left edge. Turn left.
        motors.run(LEFT, 0)
        motors.run(RIGHT, SPEED)
```

If line sensor `0` detects the `threshold` was crossed,
turn **LEFT** *into* the line!

```
    elif vals[4]:
        # Right edge. Turn right.
        motors.run(LEFT, SPEED)
        motors.run(RIGHT, 0)
```

If line sensor `4` detects the `threshold` was crossed, turn **RIGHT** *into* the line!

```
    else:
        # TODO: Go straight
```

If the `threshold` hasn't been crossed on **either** of the *outer* 🔧line sensors, the line is still *under* your CodeBot! Keep going forward!

```
motors.run(LEFT, SPEED)
motors.run(RIGHT, SPEED)
```

Goals:

- Wait for *BTN-0* press ***before*** enabling motors.
- Using an ***if-elif-else*** 🔧condition:
- If the **LEFT** sensor isn't triggered
- and if the **RIGHT** sensor isn't triggered
- then drive **forward**.

Tools Found: Line Sensors, Motors, Variables, bool

Solution:

```
from botcore import *

# Wait for BTN-0 before moving
while True:
    if buttons.was_pressed(0):
        break

motors.enable(True)

# Set speed in one place so you can change it easily
SPEED = 30

while True:
    # Check line sensors: set for black line
    vals = ls.check(2500, False)

    # Show results on LS LEDs
    leds.ls(vals)

    # Keep the line between the edges
    if vals[0]:
        # Left edge. Turn left.
        motors.run(LEFT, 0)
        motors.run(RIGHT, SPEED)
    elif vals[4]:
        # Right edge. Turn right.
        motors.run(LEFT, SPEED)
        motors.run(RIGHT, 0)
    else:
        # Go straight.
        motors.run(LEFT, SPEED)
        motors.run(RIGHT, SPEED)

```

Objective 6 - Increased Reliability and Speed

You can make your **Line Follower** *faster* and *better* at tracking lines with some small changes to your **code**.

- First thing to focus on is where your *algorithm* is ***failing***...
- It seems like the 'bot sometimes *fails to detect the line*, but that's **not** the case!
- If you ***observe closely*** just before it *loses the path*:
  - The **LEDs** show that your bot ***does*** detect the line.

- So it *must* at least **start to turn**...

What if your 'bot *overshoots* the line?

- If it *departs* the line, it's back to ***"Go Straight"*** code!
- That means even the *slightest* ***overshoot*** sends CodeBot *off on a tangent.*

How would you fix this?

- One possibility is to try to ***turn harder***.
    - But it takes *time* for the motors to change CodeBot's direction.
    - If the 'bot is going fast and the turn is sharp, the motors may not be able to overcome its *momentum* quickly enough to prevent **overshooting** the line.
- A better fix is to **keep turning** when the line is lost!
    - That means you *only "Go Straight* if you're sure you are *centered* on the line.
    - **Use the *middle three line sensors* to confirm you are on the line.**

Check the 'Trek!

Modify your code so that the `else` becomes an `elif` and you only "Go Straight" if *one or more* of them is `True`.

- Test for `elif vals[1] or vals[2] or vals[3]:`.

Check out the 🔧Logical Operators for background on `or`.

Run It!

Test your ***Line Follower!***

- Check out the cases where it *failed* before.
- Try increasing the **SPEED**.

Tune Your Turns

Based on the layout of your **line course** you can modify your code to change the amount your 'bot turns when it sees the edge of the line.

- Remember you can also make *one wheel go* ***backwards*** if you have *tight* turns.

  This code can really *move out!*

CodeTrek:

```
from botcore import *

# Wait for BTN-0 before moving
while True:
    if buttons.was_pressed(0):
        break

motors.enable(True)

# Set speed in one place so you can change it easily
SPEED = 30

while True:
    # Check line sensors: set for black line
    vals = ls.check(2500, False)

    # Show results on LS LEDs
    leds.ls(vals)

    # Keep the line between the edges
    if vals[0]:
        # Left edge. Turn left.
        motors.run(LEFT, 0)
        motors.run(RIGHT, SPEED)
```

```
    elif vals[4]:
        # Right edge. Turn right.
        motors.run(LEFT, SPEED)
        motors.run(RIGHT, 0)
    else: # TODO: Update this 'else' to an 'elif'
```

If neither `val[0]` or `val[1]` has crossed the ***treshold***,
go *straight* **only** if the line is beneath line sensors `1`, `2`, and `3`!

Replace `else:` with:

```
    elif vals[1] or vals[2] or vals[3]:
```

```
        # Go straight.
        motors.run(LEFT, SPEED)
        motors.run(RIGHT, SPEED)

```

Goal:

- Modify your code so that the `else` becomes an `elif` and you only "Go Straight" if **ANY** of the **middle** *three* line sensors are `True`.

  The *middle three* sensors are `vals[1]`, `vals[2]`, and `vals[3]`.

Tools Found: Logical Operators, Line Sensors

Solution:

```
from botcore import *

# Wait for BTN-0 before moving
while True:
    if buttons.was_pressed(0):
        break

motors.enable(True)

# Set speed in one place so you can change it easily
SPEED = 30

while True:
    # Check line sensors: set for black line
    vals = ls.check(2500, False)

    # Show results on LS LEDs
    leds.ls(vals)

    # Keep the line between the edges
    if vals[0]:
        # Left edge. Turn left.
        motors.run(LEFT, 0)
        motors.run(RIGHT, SPEED)
    elif vals[4]:
        # Right edge. Turn right.
        motors.run(LEFT, SPEED)
        motors.run(RIGHT, 0)
    elif vals[1] or vals[2] or vals[3]:
        # Go straight.
        motors.run(LEFT, SPEED)
        motors.run(RIGHT, SPEED)

```

Objective 7 - Proportional Control

Use the line sensors for *smarter* turning control so your 'bot can go *full speed ahead!*

A weakness of your *Line Following* code is that it *always* uses the *same turning force*.

- If you have some *not too curvy* sections, you'd rather it turn **gently.**
- But if you have *sharp bends*, you need it to turn **hard!**

Instead of always using the same *turning force*, can you make the turn *proportional* to how far off-center CodeBot is?

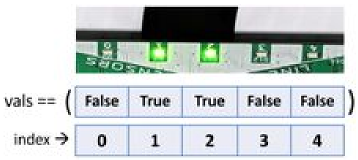
With **5** sensors you can detect much more than just a *Left* or *Right* ***edge***.

- How many **"steps"** *off-center* can you detect?
- ...it depends on the **width** of your line!

Try Your Skills: *Collect Data*

Run your last program, and **make notes** of the 🔧Line Sensor LEDs as you pass your 'bot *Left* and *Right* across the line.

- Find **every detected position** by *slowly* moving your 'bot ***Left*** and ***Right***.



Lift the wheels slightly off the surface while you do this test. Or 🔧comment-out the `# motors.enable()` so you can observe the LEDs without motors running.

Example: ***My Collected Data***

Using standard 3/4" black electrical tape on a white surface, I got the following:

Your data may be different - run your own experiments and find out!

Line Pos	LEDs (vals)
Far Left	vals == (**True**, False, False, False, False)
	vals == (**True**, **True**, False, False, False)
	vals == (False, **True**, False, False, False)
	vals == (False, **True**, **True**, False, False)
Center	vals == (False, False, **True**, False, False)
	vals == (False, False, **True**, **True**, False)
	vals == (False, False, False, **True**, False)
	vals == (False, False, False, **True**, **True**)
Far Right	vals == (False, False, False, False, **True**)

As the table above shows, I can detect **4 steps** of *off-center* in both *Left* and *Right* directions.

Check the 'Trek!

There are **two** significant changes to your code:

1. In your `if statement:`, instead of comparing *indexed values* like: `vals[0] or vals[1]`, **compare** 🔧tuples with code like: `if vals == (0,1,1,0,0):`
   - **Yes!** A 🔧comparison on the *whole sequence* all at once!
   - ***Note:*** *You can use **0** and **1** rather than **False** and **True** to save typing!*
2. Instead of calling `motors.run()` for LEFT and RIGHT motors at every different line position, ***define a function*** `drive(left, right)` to do it in one line of code **plus** add a *SPEED_LIMIT*.

Use the code in the CodeTrek as a guide, but ***make it your own!***

Run It!

Take your new version for a *test drive!*

- How well does it navigate gently turning sections?
- Can it handle the sharp turns?

You will probably want to do some more ***adjustments*** to make this version work in all your test scenarios!

CodeTrek:

```
from botcore import *

# Wait for BTN-0 before moving
while True:
    if buttons.was_pressed(0):
        break

motors.enable(True)

# Speed limit. Max speed = 1.0, 50% is 0.5, etc.
SPEED_LIMIT = 0.8

def drive(left, right):
    # Apply -100 to 100 power to motors, enforcing SPEED_LIMIT.
    motors.run(LEFT, left * SPEED_LIMIT)
    motors.run(RIGHT, right * SPEED_LIMIT)
```

Your *new* `drive()` function takes the **LEFT** and **RIGHT** *motor speed* as arguments.

- Implementing a `SPEED_LIMIT` variable will allow you to *alter* the **MAX** speed *easily* during testing.

```

while True:
    vals = ls.check(2500, False)
    leds.ls(vals)

    # Drive based on sensor readings.
    if vals == (1,0,0,0,0):  # Far Left
        drive(-20, 50)
```

`if` **ONLY** the **LEFT** line sensor is triggered, *then* turn **LEFT**!

```
    elif vals == (1,1,0,0,0):
        drive(0, 60)
    elif vals == (0,1,0,0,0):
        drive(40, 80)
    elif vals == (0,1,1,0,0):
        drive(80, 100)
    elif vals == (0,0,1,0,0): # Center
        drive(100, 100)

    elif vals == (0,0,1,1,0):
        drive(100, 80)
    elif vals == (0,0,0,1,0):
        drive(80, 40)
    elif vals == (0,0,0,1,1):
        drive(60, 0)
    elif # TODO: If the right sensor only is True
```

Use a tuple here! If **ONLY** the **LEFT** sensor is triggered, your tuple will look like this:

```
(0,0,0,0,1)
```

```
        drive(50, -20)
```

Goals:

- Using a tuple in the `if` statement, drive **RIGHT** when *only the rightmost* line sensor is triggered.
- *Define* a new function named `drive()` that drives **BOTH** the **LEFT** and **RIGHT** motors.

Tools Found: Line Sensors, CodeBot LEDs, Comments, tuple, Comparison Operators, Functions, Keyword and Positional Arguments, Variables

Solution:

```
from botcore import *

# Wait for BTN-0 before moving
while True:
    if buttons.was_pressed(0):
        break

motors.enable(True)

# Speed Limit. Max speed = 1.0, 50% is 0.5, etc.
SPEED_LIMIT = 0.8

def drive(left, right):
    # Apply -100 to 100 power to motors, enforcing SPEED_LIMIT.
    motors.run(LEFT, left * SPEED_LIMIT)
    motors.run(RIGHT, right * SPEED_LIMIT)

while True:
    vals = ls.check(2500, False)
    leds.ls(vals)

    # Drive based on sensor readings.
    if vals == (1,0,0,0,0):  # Far Left
        drive(-20, 50)
    elif vals == (1,1,0,0,0):
        drive(0, 60)
    elif vals == (0,1,0,0,0):
        drive(40, 80)
    elif vals == (0,1,1,0,0):
        drive(80, 100)
    elif vals == (0,0,1,0,0): # Center
        drive(100, 100)

    elif vals == (0,0,1,1,0):
        drive(100, 80)
    elif vals == (0,0,0,1,0):
        drive(80, 40)
    elif vals == (0,0,0,1,1):
        drive(60, 0)
    elif vals == (0,0,0,0,1):
        drive(50, -20)
```

Objective 8 - Read the Line

*Your **Line Follower** rocks!* But there are ***so many improvements*** you can make!

As a final step in this project, **make your 'bot *adapt* to its environment.**

Hard Coded Values

Your program uses **"hard-coded"** values for `threshold` and `is_reflective`. *You have to modify the program **each time** you change them.*

- What if you take your 'bot to a different ***environment?***
- The line and background may be *darker* or *lighter*.
- You may encounter a *reflective* line, or a *non-reflective* one.

Instead of having to *modify the code* every time, **write code** to sense the line ***automatically*** at the start of the program ***when BTN-1 is pressed.***

- The user must *first* place CodeBot with the **middle** sensor **(LS 2)** ***ON*** the line, and the **outer** sensors (**LS 0** and **LS 4**) ***OFF*** the line.
- With the 'bot in position, they press **BTN-1**, and the code *auto-calibrates*.
- Give the user a **confirmation tone** using the speaker:

- Proper line was found! → *happy 2-tone beep: "low-high"*
- Invalid detection → *sad boooop tone*

Auto-Calibration Algorithm

At the start of the program when **BTN-1** is pressed:

1. Read the **line sensor** analog values with `ls.check(0)` for integers rather than bools.

```
sensors = ls.check(0)  # Get analog values.
line = sensors[2]      # Middle sensor on the line.
ground = sensors[0]    # Just use one outer sensor.
```

2. Verify that the `line` and `ground` values are *far apart*. *(ex: `gap > 500`)*
   - If they're too *close together*, **play *invalid* beep!** *(ex: single low tone)*
   - Otherwise **play *happy* beep!** *(ex: 2-tone "low-high" chirp)*
3. If `line < ground` then the line is **reflective**. Otherwise it's **not**. Set `is_reflective` based on this.
4. Calculate and store a `thresh` value. Half-way between `line` and `ground` is a good start.

Implementation Note 1: *storing new settings*

- To keep your code **readable** you will need to put the above *algorithm* in its own function.
- Define global variables `thresh` and `is_reflective` to replace the *hard-coded* values in your code.

Concept: *globals*

You have seen the terms **Global** and **Local** when you use the ***debugger*** to inspect variables while stepping through your code.

Check out the Locals and Globals tool for more information!

When you ***assign to a variable inside a function***, Python *assumes* it's a **local** variable.

- You must use the `global` *statement* to specify you want the **global** variable instead.

Concept: *math built-ins*

CodeTrek:

```

from botcore import *
from time import sleep

# Default environment settings
thresh = 2500
is_reflective = False

def calibrate():
    # Auto-detect the line with 'bot centered on it.
    # Sets 'thresh' and 'is_reflective' globals on success.
    global thresh, is_reflective
```

When you ***assign to a variable inside a function***, Python *assumes* it's a **local** variable.

- You must use the `global` *statement* to specify you want the **global** variable instead.

The `thresh` and `is_reflected` variables are assigned *above!*

```

    # Check the line (analog values)
    sensors = ls.check(0)

    # Use center and one edge sensor
    line = sensors[2]
    ground = sensors[0]

    # Calculate gap from 'ground' to 'line'
```

```
    gap = line - ground
```

The gap 🔧variable represents the *difference* between the **line** and the **ground.**

```

    if abs(gap) < 500:
        # Too close - Invalid line
        # TODO: Play boooop sound (200Hz for 0.2 sec?)
```

Remember using the **speaker** in the ***Animatronics*** mission?

There was a specific *sequence* of code you wrote, the algorithm looked like:

1. Play a sound
2. Wait for a duration
3. Turn off the sound

In *Python*, it looks like *this!*

```
spkr.pitch(200)
sleep(0.2)
spkr.off()
```

```
    else:
        # Good line!
        # Lower numbers mean more reflective.
        is_reflective = line < ground
        # Calculate "half-way" from ground.
        thresh = ground + (gap / 2)
        # Round to an integer.
        thresh = round(thresh)
        # TODO: Play happy chirp (500Hz, 1000Hz for 0.1 sec each?)
```

Similar to the previous **step**, except now we're playing ***2*** sounds!

```
spkr.pitch(500)
sleep(0.1)
spkr.pitch(1000)
sleep(0.1)
spkr.off()
```

```

# Wait for BTN-0 before moving, and Calibrate if BTN-1 pressed.
while True:
    if buttons.was_pressed(0):
        break
    elif buttons.was_pressed(1):
        calibrate()
```

Aww yeah, *another* 🔧condition!

If the user pressed *BTN-1* **BEFORE** *BTN-0*, run the ***calibration*** 🔧function!

```
        buttons.was_pressed(1)  # Debounce!

motors.enable(True)

# Speed limit. Max speed = 1.0, 50% is 0.5, etc.
SPEED_LIMIT = 0.8

def drive(left, right):
    # Apply -100 to 100 power to motors, enforcing SPEED_LIMIT.
    motors.run(LEFT, left * SPEED_LIMIT)
    motors.run(RIGHT, right * SPEED_LIMIT)

while True:
    vals = ls.check(thresh, is_reflective)
    leds.ls(vals)


    # Drive based on sensor readings.
```

```
    if vals == (1,0,0,0,0):
        drive(-20, 50)
    elif vals == (1,1,0,0,0):
        drive(0, 60)
    elif vals == (0,1,0,0,0):
        drive(40, 80)
    elif vals == (0,1,1,0,0):
        drive(80, 100)
    elif vals == (0,0,1,0,0):
        drive(100, 100)
    elif vals == (0,0,1,1,0):
        drive(100, 80)
    elif vals == (0,0,0,1,0):
        drive(80, 40)
    elif vals == (0,0,0,1,1):
        drive(60, 0)
    elif vals == (0,0,0,0,1):
        drive(50, -20)

```

Goals:

- *Define* a function named `calibrate()`.
- *In* the `calibrate()` function:
- Use the `global` statement to specify you want the **global** variables `thresh` and `is_reflective`.
- *In* the `calibrate()` function:
- Play a *'booooooooop'* (200Hz) sound if `abs(gap) < 500`.
- `else`, play a *'happy chirp'* (500Hz then 1000Hz) sound

Tools Found: Speaker, Analog to Digital Conversion, int, bool, Functions, Locals and Globals, Variables, Math Operators, float

Solution:

```

from botcore import *
from time import sleep

# Default environment settings
thresh = 2500
is_reflective = False

def calibrate():
    # Auto-detect the line with 'bot centered on it.
    # Sets 'thresh' and 'is_reflective' globals on success.
    global thresh, is_reflective

    # Check the line (analog values)
    sensors = ls.check(0)

    # Use center and one edge sensor
    line = sensors[2]
    ground = sensors[0]

    # Calculate gap from 'ground' to 'line'
    gap = line - ground

    print(gap)
    if abs(gap) < 500:
        # Too close - Invalid line
        spkr.pitch(200)
        sleep(0.2)
        spkr.off()
    else:
        # Good Line!
        # Lower numbers mean more reflective.
        is_reflective = line < ground
        # Calculate "half-way" from ground.
        thresh = ground + (gap / 2)
```

```
        # Round to an integer.
        thresh = round(thresh)
        spkr.pitch(500)
        sleep(0.1)
        spkr.pitch(1000)
        sleep(0.1)
        spkr.off()

# Wait for BTN-0 before moving, and Calibrate if BTN-1 pressed.
while True:
    if buttons.was_pressed(0):
        break
    elif buttons.was_pressed(1):
        calibrate()
        buttons.was_pressed(1)  # Debounce!

motors.enable(True)

# Speed Limit. Max speed = 1.0, 50% is 0.5, etc.
SPEED_LIMIT = 0.8

def drive(left, right):
    # Apply -100 to 100 power to motors, enforcing SPEED_LIMIT.
    motors.run(LEFT, left * SPEED_LIMIT)
    motors.run(RIGHT, right * SPEED_LIMIT)

while True:
    vals = ls.check(thresh, is_reflective)
    leds.ls(vals)


    # Drive based on sensor readings.
    if vals == (1,0,0,0,0):
        drive(-20, 50)
    elif vals == (1,1,0,0,0):
        drive(0, 60)
    elif vals == (0,1,0,0,0):
        drive(40, 80)
    elif vals == (0,1,1,0,0):
        drive(80, 100)
    elif vals == (0,0,1,0,0):
        drive(100, 100)
    elif vals == (0,0,1,1,0):
        drive(100, 80)
    elif vals == (0,0,0,1,0):
        drive(80, 40)
    elif vals == (0,0,0,1,1):
        drive(60, 0)
    elif vals == (0,0,0,0,1):
        drive(50, -20)

```

Quiz 2 **- Curves Ahead**

Slow down a bit!

With all the 🔧lists and 🔧tuples and interesting 🔧operators going on in your code, I'm getting *dizzy!*

- Take a moment to review what you've learned.

Please answer the following questions based on this tuple:

```
speeds = (-32, 73, 88, 95)
```

Question 1: What is the value of `speeds[1]`?

✓ 73

- ✗ `-32`
- ✗ `88`
- ✗ `95`

Question 2: Can you change the top speed in this 🔧tuple to 100?

- ✓ No, tuples are immutable.
- ✗ `speeds[3] = 100`
- ✗ `speeds[100] = True`
- ✗ No, tuple values can't exceed `100`

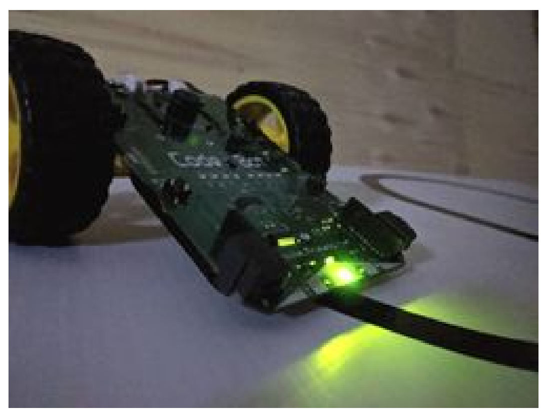
Question 3: What is `abs(speeds[0])`?

- ✓ `32`
- ✗ `73`
- ✗ `88`
- ✗ `-32`
- ✗ `0`

Mission 6 Complete

You now have the tools...

- From here you can go on to build *world class* Line-Following robotic software.
- There's always a "fine *line*" between work and play with Line Followers!
- The coding skills you've learned apply to **industrial** and **commercial** *robotics applications*.
- But they're also ***very useful*** for *school* and *club robotics competitions*.



Best of all, *you* know what makes this thing tick!

- There's no "hidden magic" going on here.
  - Even though I'll admit it *seems* magical :-)
- You're writing **Python** code *down to the metal*.

  *This is as **real** as it gets!*

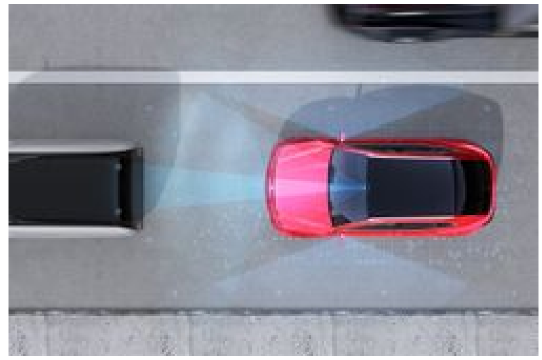
Try Your Skills

Suggested Re-mix Ideas:

- Experiment to see if there is a **better threshold** point than *half-way* between `line` and `ground`.
  - Expect that your 'bot will *bounce* slightly as it navigates a line course.
  - Can you reliably avoid erroneous sensor readings?
- Selectable driving setup:
  - Use BTN-0 to select from up to 8 different setups.
  - When ready, position 'bot and press BTN-1 to *Calibrate and Start!*
- Robot Racing Circuit:
  - Set up a *race track loop* course with some tight turns as well as gentle curves and straight sections.
  - Test your 'bot on the track, and increase its speed until it *fails.*
  - Examine ***why*** it fails, and *improve your code*.
  - Use a stopwatch to check your lap-time, and compete with your friends!

Mission 7 - Hot Pursuit

Can CodeBot *see* objects in its path?



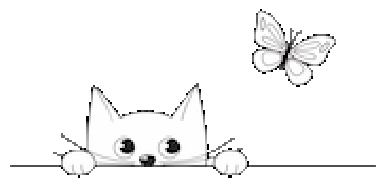
- Not exactly, since it doesn't have a *camera*.
- But it *does* have an ***Infrared Proximity Sensor System!***
- Your 'bot uses reflected IR light to detect obstacles.

In this project you'll go in-depth with the 🔧proximity sensors and write code to **detect**, **pursue**, and **avoid** objects.

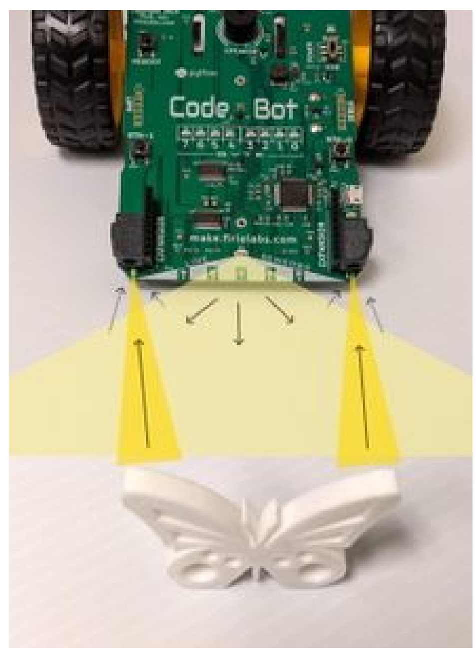
> These sensors add an *awesome new dimension* to CodeBot's capabilities!

Project Goals:

- Use the basic 🔧proximity sensors `detect()` API to make a ***presence detector***.
- Experiment with light and dark *ground-surfaces* to find the best **emitter power** and **detection threshold** levels for each environment.
- Use the `range()` API to make an interactive display of object *reflectivity*.
- Write ***calibration*** functions so CodeBot can *adapt to its environment!*
- Bring in the 🔧motors for a **Face Off** challenge.
- Code a *"Curious Puppy Bot"* that will chase a ball around.



Objective 1 - Presence Detector



The 🔧proximity sensors on CodeBot's front corners detect *infrared* (IR) light that bounces back from objects in its path.

- Each sensor is covered by a *visor* so most of the incoming light comes from *straight ahead*.
- The *visor* shades the sensor from IR in sunlight and overhead lights.
- ...It also blocks some reflections from the ground and objects to the sides.

The source of the *infrared* light is the **LED emitter** behind Line Sensor LED #2.

- Like a bright "headlight", it lights up objects in front of your 'bot.

Notice the picture on the right.

- The *LED emitter* emits light, and the *sensors* detect it.
- Most of the light *reflected* back into the sensors is from the *butterfly*.
- But there is also some light bouncing off the *ground* into the sensors...

Introducing the `prox.detect()` API.

The first function you'll use with the 🔧proximity sensors is `prox.detect()`. Calling this function *pulses* the emitter and *detects* reflected IR light.

- This function returns a 🔧tuple of *two* 🔧bool values: `(left, right)`
- The values are `True` if a reflection is detected, `False` if not.

```
vals = prox.detect()
left_detected = vals[0]
right_detected = vals[1]
```

> **Note:** The `botcore` library defines constants `LEFT = 0` and `RIGHT = 1`. You've used these with the 🔧motors, but they're also handy for the 🔧proximity sensors.

Create a New File!

Use the **File** → **New File** menu to create a new file called ***HotPursuit***.

Check the 'Trek!

Write a program that uses `prox.detect()` to *detect the* ***presence*** of an object.

- Did you know there are 🔧LEDs just in front of each **Proximity Sensor**?

Run It!

Place your 'bot so its 🔧proximity sensors are **pointed into *open space*.**

- Looking out from the *edge* of your desk is a good position for this.

The PROX LEDs should be **Off** when there is nothing in front of the 'bot.

- Use a reflective object (like a piece of white paper) to test the sensors.
- How far away can they sense?

Those **prox** LEDs are a little harder to see, since they're partly shielded by the *visors*.

- Look closely and you should see ***amber*** LEDs lighting up in front of each sensor when it detects reflection!

  You may want to modify the code so that *both* the **prox** *and* **USER** LED arrays (`leds.user(p)`) show the sensor detect status...

Experiment with this code:

- It works well on the edge of a desk.
- But what about if the 'bot is placed in the middle of a white surface, like a piece of notebook paper?

Try Your Skills: *Test your code*

Try running this code while placing CodeBot on *white* and *black* surfaces.

- Reflection from the ground could be a problem, right?
- Don't worry... your 🔧proximity sensors can overcome this issue with a little more coding.

CodeTrek:

```
from botcore import *

while True:
    # Check proximity sensors
    p = prox.detect()

    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

Goals:

- **Assign** the *output* of `prox.detect()` to the 🔧variable `p`.
- Use the variable `p` as an **argument** to `leds.prox()`.

Tools Found: Proximity Sensors, tuple, bool, Motors, LED, Variables

Solution:

```
from botcore import *

while True:
    # Check proximity sensors
    p = prox.detect()

    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

Objective 2 - Power and Sensitivity

The `prox.detect()` function is nice!

...but it has *limitations.*

- It is ***too sensitive*** when the 'bot is on a **white** surface.
- But on a **black** surface or *open space* you need *extra* sensitivity!

Be the Bot

Imagine you are in a *completely dark* room.

- You're wearing glasses that are very foggy. You can sense light from dark to bright, but that's it.
- You have a flashlight with variable power.

Think about how you'd navigate around the room, and you'll get a *sense* for how to **code** the 🔧proximity sensors!

CodeBot provides **both** the *"glasses"* and *"flashlight"* controls:

- A ***detection* sensitivity** from 0% - 100% controls *how much light* is needed for a `True` detection.

Detection Threshold	Sensitivity Level	Detect == `True`
0%	Minimum	Very bright light
100%	Maximum	Very dim light

- An ***emitter* power level** setting from **1** *(low power)* to **8** *(high power)* controls the brightness of CodeBot's *IR "flashlight"*.

More Control with `detect(power, threshold)`

It's time you get to know the two *optional* parameters for the `prox.detect()` function.

Check the 'Trek!

Modify your code to add the *optional* arguments for `prox.detect()`.

- Note: the default values with no arguments would be `prox.detect(1, 100)`.
  - That's **minimum** *"flashlight" power* and **maximum** *detection sensitivity*.

Run It!

Experiment with ***different values*** for `power` and `thresh`.

- If you ***decrease*** the `thresh` value, the 'bot works well even on a **white** surface!
- Can you find the **ideal** value for a given surface?
  - *Almost* sensitive enough to be blinded by *ground-reflection*... but not quite!
- How about your *"Open-Air"* test?
  - Photon torpedos at full power, and sensors at maximum!

CodeTrek:

```
from botcore import *

power = 1    # Minimum power "flashlight"
```

The the **minimum** *"flashlight"* power.

```
thresh = 75  # Detect at medium-high sensitivity.
```

A **medium-high** *detection sensitivity.*

- **max** is `100`!

```

while True:
    # Check proximity sensors
    # TODO: Use prox.detect() with 2 arguments
```

Use `power` and `thresh` as arguments to `prox.detect()`!

```

    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

Goals:

- **Assign** a variable named `power`.
- **Assign** a variable named `thresh`.
- Use variables `power` and `threshold` as arguments to `prox.detect()`

Tools Found: Proximity Sensors, Variables

Solution:

```
from botcore import *

power = 1     # Minimum power "flashlight" #@1
thresh = 75  # Detect at medium-high sensitivity. #@2

while True:
    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

Objective 3 - Range Scanning

The `prox.detect(power, thresh)` function lets you *adapt* to different environments.

But it takes a lot of *experimentation* to arrive at the ***ideal* combination** of `thresh` and `power` for a given surface.

- Fortunately the proximity sensors ***API*** has another function which makes it much easier.

Concept: *prox.range()*

The `prox.range()` function **scans *multiple*** *sensitivity levels* to find the ***lowest*** *detection threshold* where a reflection is detected.

Parameters *(all optional):*

```
prox.range(num_samples, power, range_low, range_high)
```

- `num_samples` : how many different sensitivity levels to try, 1-10. *(default = 4)*

- `power` : emitter power level for the scan, 1-8. *(default = 1)*
- `range_low` : lowest sensitivity range for scan, 0-100%. *(default = 0)*
- `range_high` : highest sensitivity range for scan, 0-100%. *(default = 100)*

Return value:

- 🔧tuple of `(left, right)` ***detection threshold*** values ***0-100%***.
- If ***no reflection was detected*** the return value will be `(-1, -1)`.

Try it on the REPL

Open the 🔧**Console** and test both **prox** API functions.

Notes:

- If code is *already running* you will need to press the ■ **Stop** button first!
- ***Before*** calling the new functions, type `from botcore import *` on the REPL to import the botcore library.

Test the ***prox*** functions with *no* 🔧arguments, to use the *default* parameter values.

- Type `prox.range()`, then press ↑ `ENTER` to repeat the command.
- Try **open air** proximity *(edge of desk)* while you test `prox.range()` and `prox.detect()`.

Check the 'Trek!

Modify your code to ***continuously*** call `prox.range()` and `print()` the result to the **debug console**.

- Use **10** for the `num_samples` argument.
- Higher `num_samples` is *slower* but gives *more accurate* results.

Remember, `print()` can take ***multiple*** 🔧***arguments***.

- It converts them to 🔧strings and *prints* them back-to-back to the **console**.

Run It!

Watch the *debug console* as you test with different objects and distances.

- It's *much* easier this way than typing individual commands, right?
- **Can you tell something about the *distance* to an object with `prox.range()`?**

CodeTrek:

```
from botcore import *

power = 1     # Minimum power "flashlight"
thresh = 75   # Detect at medium-high sensitivity.

while True:
    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)


    # Do a range scan
    sensed = # TODO: Do a range scan
```

`prox.range()` has **all** these *optional* parameters:

```
prox.range(num_samples, power, range_low, range_high)
```

For *this* example, set `num_samples` to `10`, and use your `power` 🔧variable as the **second** argument.

```
        sensed = prox.range(10, power)

    print("Range=", sensed)

```

Hint:

- *Getting an error?*
- 🔧Import **botcore** first by typing `from botcore import *` in the REPL!

Goals:

- **Execute** the function `prox.range()` in the 🔧REPL.
- **Assign** the *output* of `prox.range()` to the 🔧variable `sensed`.

Tools Found: Proximity Sensors, Functions, tuple, Print Function, Keyword and Positional Arguments, str, REPL, Variables

Solution:

```
from botcore import *

power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.

while True:
    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)


    # Do a range scan
    sensed = prox.range(10, power)
    print("Range=", sensed)

```

Objective 4 - Auto Calibration - part 1

You may have noticed some similarities between the 🔧Line Sensors and the 🔧Proximity Sensors.

- Both are based on *reflected infrared light*.
- And both require the 'bot to ***adapt*** **to its environment.**

Just like before, you've started by using **"hard-coded"** values.

- Surfaces with different *ground-reflection* require different values for `thresh` and `power`.

So once again, you need...

Auto-Calibration

Here's how it should work:

- Position the 'bot in a new *environment*, with no objects in front.
- When **BTN-1** is pressed, the 'bot scans to automatically find the ***ideal*** `power` and `thresh`.
- Those settings are saved until the next time **BTN-1** is pressed.

First step is to automate your `thresh` setting:

Check the 'Trek!

Modify your code to check for **BTN-1** inside the `while loop`:

- When pressed, grab `sensed = prox.range(10, power)`.
- Set your new `thresh` to **5% of the *maximum* threshold** *(`5`)* below the *minimum* `sensed[LEFT]` or `sensed[RIGHT]` value.
- ***Unless*** the value is `-1`, which means it detected **nothing!**
- If *both* sensors detect *nothing*, then `thresh` should be **100%**.

Run It!

Test out your new **Auto calibration** feature.

- Watch the **debug console** to see what `thresh` values it picks.
- Try some *different object distances* to make sure it's working.

CodeTrek:

```
from botcore import *

power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.


while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
```

The instructions state:

- *When BTN-1 is pressed, the 'bot scans to automatically find the ideal power and thresh.*
- *Those settings are saved until the next time BTN-1 is pressed.*

Therefore, we need to *continuously* check if *BTN-1* has been pressed!

- That's why your `if` statement should be indented *under* the while loop.

```
        # Assume we detect nothing
        det = 100  # max possible threshold
```

The variable `det` will represent the *lowest*
threshold reading between the **two** sensors.

You'll be *reassigning* the value ***IF*** one of the sensors reads a value.

- What if the sensors don't read any value (`-1`)?

  Then we'll need to use the **MAX** threshold!

So, set the *default* value of `det` to the **MAX** threshold, `100`!

```

        # Scan for minimum detection threshold
        sensed = prox.range(10, power)

        # Did Left detect something?
        if sensed[LEFT] > 0:
            # Save, it might be minimum...
            det = sensed[LEFT]
```

If the `LEFT` proximity sensor detected something, **assign** it to `det`.

- Since you haven't checked the *value* of the `RIGHT` sensor yet, this is your **LOWEST** known value!

```

        # Did Right detect something?
        if sensed[RIGHT] > 0:
            # Keep the minimum detected value.
            det = min(det, sensed[RIGHT])
```

If the `RIGHT` sensor *detects* a value, you'll need to do a *tiny* bit more work.

- You need to check whether it's **LOWER** than the value read by the `LEFT` sensor!
- *Fortunately,* if the `LEFT` sensor read a value, you *assigned* it to the variable `det`!

Simply compare the value of `det` to `sensed[RIGHT]` to find the **LOWEST** reading between the two sensors!

The function `min(det, sensed[RIGHT])` returns the **LOWEST** value!

```
        thresh = # TODO: Set to 5% below minimum ground-reflection detected.
```

The **MAXIMUM** threshold is `100`.

- 5% of `100` is `5`!

`thresh = det - 5`

```
        print("Sensed=", sensed, ", set thresh=", thresh)


    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

Goals:

- Check for *BTN-1* **press** inside the `while loop`.
- Use the `min` function to determine the **minimum** value *between* `sensed[RIGHT]` and `det`!
- ***After*** determining the *lowest sensor value* and assigning it to `det`:
- Set your new `thresh` to **5% of the *maximum* threshold** *(`5`)* below the *minimum* `sensed[LEFT]` or `sensed[RIGHT]` value.

Tools Found: Line Sensors, Proximity Sensors, Functions, Loops, Variables

Solution:

```
from botcore import *

power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.


while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        # Assume we detect nothing
        det = 100  # max possible threshold

        # Scan for minimum detection threshold
        sensed = prox.range(10, power)

        # Did Left detect something?
        if sensed[LEFT] > 0:
            # Save, it might be minimum...
            det = sensed[LEFT] #@1

        # Did Right detect something?
        if sensed[RIGHT] > 0:
            # Keep the minimum detected value.
            det = min(det, sensed[RIGHT]) #@2

        thresh = det - 5

        print("Sensed=", sensed, ", set thresh=", thresh)


    # Check proximity sensors
    p = prox.detect(power, thresh)

```

```
    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

***Objective 5* - Auto Calibration - part 2**

Notice an Error?

Be sure to **test** the following *requirement:*

"If *both* sensors detect *nothing*, then `thresh` should be **100%**."

Right now your program is not handling that case!

- That there's a *bug*, friend :-)

Check the 'Trek!

Modify your code to fix the bug!

- Initialize `det` to a large "invalid" value.
- Then where you set `thresh`, add an `if` check to decide whether to use `det - 5` or `100`% as the new value.

Run It!

Give it another try, and **test** the *no ground-reflection* case also.

- This works pretty well...
- But you *still* have to experiment to find the right `power` **level** to set.
  - You'll ***automate*** that in the next step of the project.

Next Steps

Some of the code above is *crying out* to be made into a function. *(Can you hear it?)*

- In the next step of the project you'll take care of that!

Oh, and there's still a bug or two left in this code... perhaps you can ignore it for now :-)

CodeTrek:

```
from botcore import *

power = 1     # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.

while True:
    # Calibrate when BTN-1 pressed

    if buttons.was_pressed(1):
        # Assume we detect nothing
        det = 101  # larger than max possible sensitivity
```

Assign `det` an **invalid** value!

- If `det` is *still invalid after* checking **both** sensors, you'll know both returned `-1`!

```

        # Scan for minimum detection threshold
        sensed = prox.range(10, power)

        # Did Left detect something?
        if sensed[LEFT] > 0:
            # Save, it might be minimum...
            det = sensed[LEFT]

```

```
        # Did Right detect something?
        if sensed[RIGHT] > 0:
            # Keep the minimum detected value.
            det = min(det, sensed[RIGHT])

        # Set thresh
        if det > 100:
            # Nothing was detected, so set to max!
            # TODO: set thresh to the maximum threshold value!
```

`det` is still **invalid!**

- *Therefore*, the sensors didn't detect anything!
- Set the `thresh` to the **maximum** value!

```
thresh = 100
```

```
        else:
            # Set to 5% below minimum ground-reflection detected.
            thresh = det - 5
```

`det` is no longer **invalid!**

- ***At least one*** of the sensors returned a value!
- Carry on *as per usual* and take away **5%!**

```


        print("Sensed=", sensed, ", set thresh=", thresh)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

Goals:

- *Initialize* the 🔧variable `det` to a large **invalid** (`>100`) value.
- If `det` is **invalid** *after* checking both sensors, set `thresh` to the **MAXIMUM** value.

Tools Found: Functions, Variables

Solution:

```
from botcore import *

power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.

while True:
    # Calibrate when BTN-1 pressed

    if buttons.was_pressed(1):
        # Assume we detect nothing
        det = 101  # Larger than max possible sensitivity

        # Scan for minimum detection threshold
        sensed = prox.range(10, power)

        # Did Left detect something?
        if sensed[LEFT] > 0:
            # Save, it might be minimum...
            det = sensed[LEFT]

        # Did Right detect something?
        if sensed[RIGHT] > 0:
            # Keep the minimum detected value.
            det = min(det, sensed[RIGHT])
```

```
            # Set thresh
            if det > 100:
                # Nothing was detected, so set to max!
                thresh = 100
            else:
                # Set to 5% below minimum ground-reflection detected.
                thresh = det - 5


            print("Sensed=", sensed, ", set thresh=", thresh)

        # Check proximity sensors
        p = prox.detect(power, thresh)

        # Show (left, right) on the PROX LEDs
        leds.prox(p)
```

Quiz 1 - Checkpoint

Question 1: Which of these is the **best** criticism of this 🔧comment?

```
# Set to 5% below minimum ground-reflection detected.
thresh = det - 5
```

✓ If someone later changes the code to a different % value, say 8%, the comment will be wrong.

✗ TLDR. You lost me at "minimum"...

✗ The % character is considered an offensive rune in Elvish.

✗ The word "bellow" is spelled with two l's, not just one.

Question 2: What does the 🔧function `prox.detect()` **return?**

✓ A **tuple** of *two* **boolean** values.

✗ A **tuple** of *two* **integers**.

✗ A **boolean**.

✗ *Nothing.*

Question 3: What does the **function** `prox.range()` return ***if no reflection was detected.***

✓ `(-1, -1)`

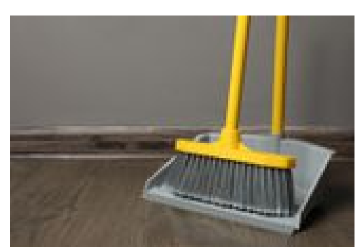
✗ `(False, False)`

✗ `(True, True)`

✗ `(False, True)`

Objective 6 - House Cleaning

Time for some *chores!*



Sometimes, being a **great** programmer requires you to *slow down* and **improve** your code!

- *Take some time* to **organize** by moving your **threshold calibration** code to its own 🔧function.

Check the 'Trek!

Modify your code by adding a function `def cal_thresh():`

- Cut and paste the code from beneath your `if buttons.was_pressed(1):` statement into the new function.
- You will need to declare `thresh` as a global inside the function.
- Add a `print()` statement in your main program to confirm the current `power` and `thresh` settings.
- Don't forget to **call** your new function when the button was pressed!

Run It!

Test your program and make sure it works just like before.

- Verify that your `print()` statement beneath the `if buttons` shows that you're *really* changing the global `thresh` value!

CodeTrek:

```
from botcore import *

def cal_thresh():
```

Define your *new* function named `cal_thresh`!

```
    # Calibrate detection threshold using prox.range()
    global thresh
```

Declare `thresh` as a global variable.

- This will allow you to **modify** the variable from *within* the function!

```

    # Assume we detect nothing
    det = 101  # larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)

power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.


while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        # TODO: Call your newly defined function
```

Call your new function!

```
        print("CAL: power=", power, ", thresh=", thresh)
```

Print the `power` and `thresh` to **verify** that your variables are being modified.

```


    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

Goals:

- **Define** a new function named `cal_thresh`.
- **Cut and paste** the code from beneath your `if buttons.was_pressed(1)` statement as the function's content.
- **Declare** `thresh` as a global variable *inside* the `cal_thresh` function.

Tools Found: Functions, Locals and Globals, Print Function, Variables

Solution:

```
from botcore import *

def cal_thresh():
    # Calibrate detection threshold using prox.range()
    global thresh

    # Assume we detect nothing
    det = 101  # larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)

power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.


while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_thresh()
        print("CAL: power=", power, ", thresh=", thresh)

```

```

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

Objective 7 **- POWER**

Now you're ready for `power`!

Your code can *calibrate* the **detection sensitivity threshold**, but can it figure out the ***ideal*** ***`power`*** ***level?***

- You will need a 🔧loop to cycle through each `power` level from **1** to **8**.
- ***STOP*** when either RIGHT *or* LEFT sensors detect reflection!

Check the 'Trek!

Modify your code by adding another 🔧function `def cal_power():`

- Declare `power` as a 🔧global inside this function.
- Loop through the `power` levels, and call `cal_thresh()` for each one.
- `break` out of the loop when reflection is detected.
- Call this new function instead of `cal_thresh()` in your `if buttons` block.

Run It!

Are you feeling the power?

This code is really nice.

- Watch the **debug console** as you *test different surfaces*.
- ***Be careful*** to **stay behind the sensors *as you press BTN-1***

CodeTrek:

```
from botcore import *

def cal_thresh():
    # Calibrate detection threshold using prox.range()
    global thresh

    # Assume we detect nothing
    det = 101  # Larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

```

```
    print("Sensed=", sensed, ", set thresh=", thresh)


def cal_power():
    # Calibrate power and detection threshold
    # TODO: Declare power as a global variable
```

Declare `power` as a global variable so you can *alter* it's value from within your *new* `cal_power` function!

- `global power`

```
    power = 0
    while power < 8:
        power = power + 1
        cal_thresh()
```

Calling `cal_thresh()` will *alter* the `thresh` variable!

- If you alter `power`, then call `cal_thresh()`, you'll be able to tell the **LOWEST** `power` level that detects a **reflection!**

```
        if thresh < 100:
            # Reflection detected.
            break
```

If a `thresh` value is ***detected***, you found an **ideal** `power` level, `break`!

```


power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.

while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_power()
```

Replace your `cal_thresh()` call with a `cal_power()` call.

- `cal_power` is *already* calling `cal_thresh`, no need to double up!

```
        print("CAL: power=", power, ", thresh=", thresh)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

Goals:

- **Define** a function named `cal_power`.
- **Declare** `power` as a global.
- Loop **through** the `power` levels.
- Call `cal_power()`.

Tools Found: Loops, Functions, Locals and Globals, Variables

Solution:

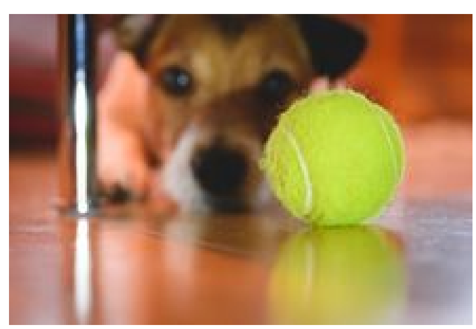
```
from botcore import *

def cal_thresh():
```

```
    # Calibrate detection threshold using prox.range()
    global thresh

    # Assume we detect nothing
    det = 101  # larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)


def cal_power():
    # Calibrate power and detection threshold
    global power
    power = 0
    while power < 8:
        power = power + 1
        cal_thresh()
        if thresh < 100:
            # Reflection detected.
            break


power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.

while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_power()
        print("CAL: power=", power, ", thresh=", thresh)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)
```

Objective 8 **- Face Off!**

Are you ready to get ***moving?***

- I *thought* so!



Get Your Motor(s) Running!

Your next mission is to write code so your 'bot uses its proximity sensors to detect and ***rotate*** to **face** an object moving in front of it.

- You just have to add a small amount of code to the end of your main `while` loop to make it happen!

Hint: After the `leds.prox(p)`, use an `if` statement with `p[LEFT]` and `p[RIGHT]` to control the direction of the motors.

Check the 'Trek!

Modify your code so your 'bot **Rotates to Face an Object**.

You have all the tools to do this step *on your own.*

- Check the Motors *tool* for a refresher if you need it.
- *Or take a look through your previous mission code!*
- Keep the *rotation speed* down around 50% to start with.

Run It!

How's it *rotating?*

- Don't forget to **Calibrate** using **BTN-1**.
- Does your 'bot **stop** when it's facing an object?
    - Or does it *oscillate* back and forth?
    - Like an *excited puppy?!?*
- How well does it ***track a moving target?***
    - Does it lose track? If so, *why?*
    - Can you *improve* it?

CodeTrek:

```
from botcore import *

def cal_thresh():
    # Calibrate detection threshold using prox.range()
    global thresh

    # Assume we detect nothing
    det = 101  # larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)

def cal_power():
    # Calibrate power and detection threshold
    global power
    power = 0
    while power < 8:
        power = power + 1
        cal_thresh()
        if thresh < 100:
            # Reflection detected.
            break


power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.
```

```
while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_power()
        print("CAL: power=", power, ", thresh=", thresh)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)


    # Use p[LEFT], p[RIGHT] to control the motors!
    # TODO: code this part
```

This part is up to **you!**

- `p[LEFT]` tells you if there's an object to your... `LEFT`!
- `p[RIGHT]` tells you the opposite!
- If **both** are `true`, the object is *infront!*

Give it your best shot!

Goals:

- Use an `if` condition chain as shown below:

```
if p[LEFT] and p[RIGHT]:
    # The object is straight ahead, stop moving!
elif p[LEFT]:
    # The object is to the left, turn left!
elif p[RIGHT]:
    # The object is to the right, turn right!
else:
    # The location of the object is unknown, stop moving!
```

- Within the `if` **condition** chain in the previous goal:
- Call `motors.run` in **both** directions ***at least*** **3 times** *(6 total calls).*

Tools Found: Proximity Sensors, Motors, bool

Solution:

```
from botcore import *

def cal_thresh():
    # Calibrate detection threshold using prox.range()
    global thresh

    # Assume we detect nothing
    det = 101  # larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

```

```
    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)

def cal_power():
    # Calibrate power and detection threshold
    global power
    power = 0
    while power < 8:
        power = power + 1
        cal_thresh()
        if thresh < 100:
            # Reflection detected.
            break


power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.

while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_power()
        print("CAL: power=", power, ", thresh=", thresh)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)


    # Use p[LEFT], p[RIGHT] to control the motors!
    motors.enable(True)
    if p[LEFT] and p[RIGHT]:
        # The object is straight ahead, do nothing!
        motors.run(LEFT, 0)
        motors.run(RIGHT, 0)
        motors.enable(False)
    elif p[LEFT]:
        # The object is to the left, turn left!
        motors.run(LEFT, -50)
        motors.run(RIGHT, 50)
    elif p[RIGHT]:
        # The object is to the right, turn right!
        motors.run(LEFT, 50)
        motors.run(RIGHT, -50)
    else:
        motors.run(LEFT, 0)
        motors.run(RIGHT, 0)
        motors.enable(False)
```

Objective 9 - Face Off Harder!

Another Handy Feature

While you're testing code like this it's nice to be able to ***toggle the motors ON/OFF***.

Check out the code *below!*

```
# Toggle a variable
go_motors = False

go_motors = not go_motors # (not False) == True

go_motors = not go_motors # (not True) == False

print("value=", go_motors) # "value= False"
```

- It uses the 🔧logical operator `not`.
- See how it *flips* the 🔧bool value from `False` to `True`?
- Then **next** time, it *flips it back* to `False`!

Use the `not` operator to *toggle* a 🔧variable for `motors.enable()`.

- You'll need code *inside your loop* checking for a *button press.*
- Use **BTN-0** for the "toggle" action.

Check the 'Trek!

Modify your code to add the ***motor toggle*** feature described above.

- Be sure to initialize the `go_motors` variable **above** your `while` loop.
- Maybe you should control a **USER LED** also, to *warn* the user when motors are *live*!

Run It!

Check it out!

Now you have **BTN-0** to ***enable/disable*** the motors, and **BTN-1** for ***calibration***.

CodeTrek:

```
from botcore import *

def cal_thresh():
    # Calibrate detection threshold using prox.range()
    global thresh

    # Assume we detect nothing
    det = 101  # Larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)

def cal_power():
    # Calibrate power and detection threshold
    global power
    power = 0
    while power < 8:
        power = power + 1
        cal_thresh()
        if thresh < 100:
            # Reflection detected.
            break


power = 1    # Minimum power "flashlight"
```

```
thresh = 75  # Detect at medium-high sensitivity.
go_motors = False # Motors are OFF initially
```

Initialize the variable `go_motors`, start with the motors turned **OFF**.

```

while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_power()
        print("CAL: power=", power, ", thresh=", thresh)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)

    # --- Motor control code from the previous objective ---
```

You ***don't*** need to copy this comment down!

- This is where your code from the *last* objective will be!

```

    if buttons.was_pressed(0):
        # TODO: Toggle go_motors
```

Assign the variable `go_motors` to the **inverse** of it's current value!

- `go_motors = not go_motors`

Now the value is flipped!

```
        motors.enable(go_motors)
```

Use the *newly flipped* `go_motors` variable as the argument to `motors.enable` to turn the motors **ON** or **OFF**.

```
        # Show "enabled" status LED
        leds.user_num(0, go_motors)
```

Display the *state* of `go_motors` on user LED `0`.

- Watch it update and *reflect* the state of the motors.

Goals:

- **Toggle** `motors.enable` by using the `not` operator on the `go_motors` variable.
- Call `leds.user_num(0, go_motors)`.

Tools Found: Logical Operators, bool, Variables, Motors, CodeBot LEDs

Solution:

```
from botcore import *

def cal_thresh():
    # Calibrate detection threshold using prox.range()
    global thresh

    # Assume we detect nothing
```

```
    det = 101  # larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)

def cal_power():
    # Calibrate power and detection threshold
    global power
    power = 0
    while power < 8:
        power = power + 1
        cal_thresh()
        if thresh < 100:
            # Reflection detected.
            break


power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.
go_motors = False # Motors are OFF initially

while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_power()
        print("CAL: power=", power, ", thresh=", thresh)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)

    # --- Motor control code from the previous objective ---

    if buttons.was_pressed(0):
        go_motors = not go_motors
        motors.enable(go_motors)
        # Show "enabled" status LED
        leds.user_num(0, go_motors)

```

Objective 10 **- Chase Mode**

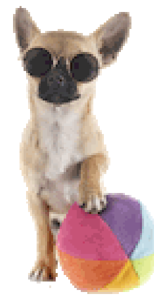
Your 'bot only needs a *tiny nudge* now to not just *track* an object, but to ***chase after it!***

A Simple Algorithm

1. If both **LEFT** `and` **RIGHT** sensors see an object, ***move forward.***
2. When only the **LEFT** sensor detects something, ***rotate left.***
3. When only the **RIGHT** sensor detects something, ***rotate right.***
4. If neither sensor detects something, ***stop moving.***

Even the simple algorithm above can empower your 'bot to follow a ball around!

- *Get **coding!***



Check the 'Trek!

Modify your code to **chase** an object in view!

Run It!

Test your code with a *variety* of ***objects!***

- What **sizes, shapes,** and **colors** work best?
- Try an *empty* **paper cup** turned *on its **side***.
    - Does your 'bot follow it around in a *circle?*

CodeTrek:

```
from botcore import *

def cal_thresh():
    # Calibrate detection threshold using prox.range()
    global thresh

    # Assume we detect nothing
    det = 101  # larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)

def cal_power():
    # Calibrate power and detection threshold
    global power
    power = 0
    while power < 8:
        power = power + 1
        cal_thresh()
        if thresh < 100:
            # Reflection detected.
            break

```

```

power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.
go_motors = False
SPEED = 50
```

Initialize the variable SPEED so you can *easily* adjust the speed by updating a single 🔧variable!

- Start with a SPEED of 50 but *feel free* to change it *as you see fit!*

```

while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_power()
        print("CAL: power=", power, ", thresh=", thresh)

    # BTN-0 toggles motors ON/OFF
    if buttons.was_pressed(0):
        go_motors = not go_motors
        motors.enable(go_motors)
        # Show "enabled" status LED
        leds.user_num(0, go_motors)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)


    # Chase mode!
```

The *difference* between **chasing** and **rotating** is *minimal!*

- Rather than **stopping** when both LEFT and RIGHT sensors are triggered, **go forward!**

```
    if p[LEFT] and p[RIGHT]:
        # I see you! Charge ahead!
        # TODO: drive forward!
```

Full steam ahead!

```
    elif p[RIGHT]:
        # Rotate right to face object.
        motors.run(LEFT, +SPEED)
        motors.run(RIGHT, -SPEED)
    elif p[LEFT]:
        # Rotate left to face object.
        motors.run(LEFT, -SPEED)
        motors.run(RIGHT, +SPEED)
    else:
        # Nothing in view. Stop moving.
        motors.run(LEFT, 0)
        motors.run(RIGHT, 0)
```

Goal:

- If **both** p[LEFT] *and* p[RIGHT] sensors are triggered, ***drive forward!***

Tools Found: Variables

Solution:

```
from botcore import *

def cal_thresh():
    # Calibrate detection threshold using prox.range()
```

```
    global thresh

    # Assume we detect nothing
    det = 101  # Larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)

def cal_power():
    # Calibrate power and detection threshold
    global power
    power = 0
    while power < 8:
        power = power + 1
        cal_thresh()
        if thresh < 100:
            # Reflection detected.
            break


power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.
go_motors = False
SPEED = 50

while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_power()
        print("CAL: power=", power, ", thresh=", thresh)

    # BTN-0 toggles motors ON/OFF
    if buttons.was_pressed(0):
        go_motors = not go_motors
        motors.enable(go_motors)
        # Show "enabled" status LED
        leds.user_num(0, go_motors)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)


    # Chase mode!
    if p[LEFT] and p[RIGHT]:
        # I see you! Charge ahead!
        motors.run(LEFT, +SPEED)
        motors.run(RIGHT, +SPEED)
    elif p[RIGHT]:
        # Rotate right to face object.
        motors.run(LEFT, +SPEED)
        motors.run(RIGHT, -SPEED)
    elif p[LEFT]:
        # Rotate left to face object.
```

```
        motors.run(LEFT, -SPEED)
        motors.run(RIGHT, +SPEED)
    else:
        # Nothing in view. Stop moving.
        motors.run(LEFT, 0)
        motors.run(RIGHT, 0)

```

Objective 11 **- Smarter Pursuit**

You may already have ideas about making an even *smarter Chase Bot!*

Here are some thoughts:

- What do you do when *nothing* is detected?
  - Is ***stopping*** the best answer?
  - Should you keep acting on the last *detection?*
  - Maybe run a search pattern...
- Does your 'bot really need to do ***full rotation code*** when the target it's chasing just swerved a little to one side?
  - *Rotation* stops forward progress. *It's slowing you down!*
- Your 'bot only checks *instantaneous* sensor readings, and *ignores the past.*
  - Losing sight of the target for an *instant* is no reason to give up!

Check the 'Trek!

Time to upgrade!

- Instead of always doing *"full rotation"*, make a new function `def drive(speed, turn_ratio)`
  - The `turn_ratio` parameter is the fraction of `speed` to use for turning.
  - So `0` will go straight, while `-0.2` would veer LEFT a little.
- Instead of acting on *instantaneous* sensor readings, ***add up*** **multiple samples**.
  - ***Ex:*** For every 100 readings, keep count of **LEFT**s versus **RIGHT**s.
  - Use that sample data to *calculate* the `turn_ratio`.

The code below replaces your `if...elif` logic with +/- calculation of a *"target position"*, and defines the new `drive()` function.

Run It!

Test this version out!

- It only takes about a *tenth of a second* for this code to *read 100 samples!*
- So it's *adjusting the motors* about **10 times per second**.
- Try setting the `SPEED` to **100** (max), and see how it performs.

You may want to put some `print()` statements of your own in this code, to ***explore*** what's going on with different sensor readings while it's running.

There are *many* ways to *improve* your ChaseBot!

- There is **no *perfect "Right Way"*** to implement this.
- I've just given you *some* ways you might begin thinking about it.
- ... and there's no doubt that ***YOU*** can make a much, much better version!

CodeTrek:

```
from botcore import *

def cal_thresh():
    # Calibrate detection threshold using prox.range()
    global thresh

    # Assume we detect nothing
    det = 101  # larger than max possible sensitivity
```

```

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)

def cal_power():
    # Calibrate power and detection threshold
    global power
    power = 0
    while power < 8:
        power = power + 1
        cal_thresh()
        if thresh < 100:
            # Reflection detected.
            break

def drive(speed, turn_ratio):
    # A fraction of the speed goes to turning.
    # speed: 0-100 ; turn_ratio: L=-1, R=+1, 0=straight
```

Your *new* 🔧function, `drive(speed, turn_ratio)`, will *gently* turn the 🔧motors using **math!**

- The 🔧parameter `speed` is self-explanatory!
- `turn_ratio` is *how hard* you should turn **left** or **right,**
    - `-1` means turn **left *hard!***
    - `-0.1` means turn **left** *gradually.*
    - `1` means turn **right *hard!***

```
    turn_spd = speed * turn_ratio
```

`turn_spd` represents... *the turn speed!*

- If the `turn_ratio` is `-1` *(which means **hard LEFT!**)*, and the `speed` is `50`, the `turn_spd` would be `-50`!
- You'll use this in *combination* with **forward speed** to determine each motor's speed!

```
    fwd_spd = speed - abs(turn_spd)
```

`fwd_spd` represents... *the forward speed!*

- The **harder** your turning **left** or **right**, the *slower* you want to move forward.
- The `abs(number)` returns the *absolute* value of a number.

```
abs(-50) == 50!
```

Therefore, if the `turn_spd` is `-50` (***hard left***), and the `speed` is `50`...

```
50 - abs(-50) == 0!
```

The faster the 'bot is turning, the slower forward it'll go!

```
    motors.run(LEFT, fwd_spd + turn_spd)
    motors.run(RIGHT, fwd_spd - turn_spd)
```

When turning `LEFT`:

- You want to **add** `fwd_spd` and `turn_spd` becuase `turn_spd` is **negative** when it represents a **left** turn!

When turning `RIGHT`:

- You want to **subtract** `fwd_spd` and `turn_spd` becuase `turn_spd` is **positive** when it represents a **right** turn!

```


power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.
go_motors = False
SPEED = 50

# Sensor reading sample data
n_sample = 0  # Count number of samples
target = 0    # Sensed target position.
              # +increment RIGHT, -decrement LEFT
```

In previous objectives, your 'bot was **jolting** back and forth when the sensor reading changed.

- You need a more *gradual* **response.**

So, rather than turning **every time** the reading changes, turn after `100` readings!

- The variable `n_sample` represents the **total** number of readings.
- The **variable** `target` represents the difference in the quantity of `LEFT` and `RIGHT` sensor hits.

```

while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_power()
        print("CAL: power=", power, ", thresh=", thresh)

    # BTN-0 toggles motors ON/OFF
    if buttons.was_pressed(0):
        go_motors = not go_motors
        motors.enable(go_motors)
        # Show "enabled" status LED
        leds.user_num(0, go_motors)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)


    # Update sample data.
    # Adjust sensed 'target' position to left(-) or right(+)
    n_sample = n_sample + 1
```

`n_sample` is a *tally* of the **number** of sensor readings.

- *Therefore,* every time we read the sensor, iterate `n_sample`!

```
    if p[LEFT]:
        target = target - 1
    if p[RIGHT]:
        target = target + 1
```

Here's how `target` will *keep track of* the **number** of `LEFT` and `RIGHT` sensor hits:

- If the `LEFT` sensor hits, **decrement** *(subtract `1`)* from `target`!

- If the `RIGHT` sensor hits, **increment** *(add `1`)* to `target`!

If *both* sensors hit, `target` will *end up unchanged!*

- Over `100` readings, `target` will tell us if one sensor hit *more* than the other!

```
    # Adjust motors each sample interval.
    if n_sample == 100:
        # Use 'target' value and 'n_sample' to calculate 'turn_ratio'.
        drive(SPEED, target / n_sample)
```

Time to put it all together!

- The `turn_ratio` 🔧parameter **expected** by the function `drive` can be *derived* by **dividing** `target` by `n_sample`.
- If *more* `LEFT` sensor hits are read, the value will be **negative.**
- If *more* `RIGHT` sensor hits are read, the value will be **positive.**
- The value will *always* fall **within** a range between `-1` and `1`!

```
        # Reset for next sample interval
        n_sample = 0
        target = 0

```

Goals:

- **Define** a function called `drive(speed, turn_ratio)`.
- **Call** `drive` every `100` samples.
- *When `n_sample` reaches `100`!*
- **Reset** `n_sample` and `target` *after* `100` samples.

Tools Found: Functions, Motors, Parameters, Arguments, and Returns, Variables, Iterable

Solution:

```
from botcore import *

def cal_thresh():
    # Calibrate detection threshold using prox.range()
    global thresh

    # Assume we detect nothing
    det = 101  # larger than max possible sensitivity

    # Scan for minimum detection threshold
    sensed = prox.range(10, power)

    # Did Left detect something?
    if sensed[LEFT] > 0:
        # Save, it might be minimum...
        det = sensed[LEFT]

    # Did Right detect something?
    if sensed[RIGHT] > 0:
        # Keep the minimum detected value.
        det = min(det, sensed[RIGHT])

    # Set thresh
    if det > 100:
        # Nothing was detected, so set to max!
        thresh = 100
    else:
        # Set to 5% below minimum ground-reflection detected.
        thresh = det - 5

    print("Sensed=", sensed, ", set thresh=", thresh)
```

```python

def cal_power():
    # Calibrate power and detection threshold
    global power
    power = 0
    while power < 8:
        power = power + 1
        cal_thresh()
        if thresh < 100:
            # Reflection detected.
            break

def drive(speed, turn_ratio):
    # A fraction of the speed goes to turning.
    # speed: 0-100 ; turn_ratio: L=-1, R=+1, 0=straight
    turn_spd = speed * turn_ratio
    fwd_spd = speed - abs(turn_spd)
    motors.run(LEFT, fwd_spd + turn_spd)
    motors.run(RIGHT, fwd_spd - turn_spd)


power = 1    # Minimum power "flashlight"
thresh = 75  # Detect at medium-high sensitivity.
go_motors = False
SPEED = 50

# Sensor reading sample data
n_sample = 0  # Count number of samples
target = 0    # Sensed target position. #@1
              # +increment RIGHT, -decrement LEFT

while True:
    # Calibrate when BTN-1 pressed
    if buttons.was_pressed(1):
        cal_power()
        print("CAL: power=", power, ", thresh=", thresh)

    # BTN-0 toggles motors ON/OFF
    if buttons.was_pressed(0):
        go_motors = not go_motors
        motors.enable(go_motors)
        # Show "enabled" status LED
        leds.user_num(0, go_motors)

    # Check proximity sensors
    p = prox.detect(power, thresh)

    # Show (left, right) on the PROX LEDs
    leds.prox(p)


    # Update sample data.
    # Adjust sensed 'target' position to left(-) or right(+)
    n_sample = n_sample + 1
    if p[LEFT]:
        target = target - 1
    if p[RIGHT]:
        target = target + 1

    # Adjust motors each sample interval.
    if n_sample == 100:
        # TODO: Use 'target' value and 'n_sample' to calculate 'turn_ratio'.
        drive(SPEED, target / n_sample)

        # Reset for next sample interval
        n_sample = 0
        target = 0


```

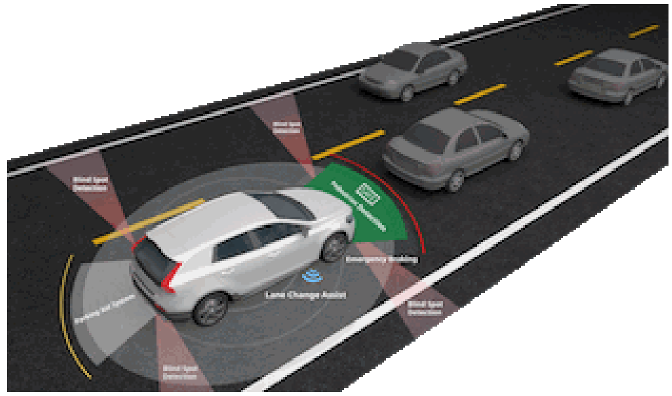
<u>Mission 7 Complete</u>

This project has given you an *in-depth* view of a type of technology that's used all around you!

Applications: Proximity Sensing

The kind of **code** you've written is inside stuff you might use every day, without even thinking about it!

- Touchless faucets, dispensers, and hand-dryers.
- Automatic doors.
- Vehicle navigation and safety systems.
- Factory automation systems of all kinds.



Try Your Skills

Suggested Re-mix Ideas:

- Create a ***"People Counter".***
  - Start with the simple *presence detector* code.
  - Count the number of *detect* events and `print()` it to the **debug console**.
- Code an ***obstacle avoiding robot***.
  - You can **chase** objects, but can you ***avoid*** them?
  - Turn *away* from detected objects and run!
- Code your ***own*** version of `prox.range()` using only the `prox.detect()` function.
  - Loop until you find the *minimum* sensitivity where `detected == True`
  - It won't likely be as *fast* as `prox.range()`, but it will be *yours!*

Mission 8 **- Navigation**

Plotting a Course for Your 'Bot!

- You're already using *sensors* for ***navigation.***
- 🔧Line Sensors can guide you precisely - as long as there's a guiding line on the surface.
- Following or avoiding objects with the 🔧Proximity Sensors is a form of *navigation* also.

But what if you just need to *move forward* a certain **distance** at a certain **speed?**

- ...or *rotate* for a specific **angle?**

Maybe you're thinking, "just use `motors.run()` and `sleep()` for those movements".

*But that approach isn't very **reliable** for navigation...*

When you move using the `motors.run()` function, you give a **% power** value to the 🔧motors. The actual **speed** the 'bot travels depends on a few factors:

- **Type of surface**. Wood, vinyl, tile, carpet - all have different textures and friction.
- **Battery** charge level. Fresh batteries give more power, and 100% basically means *"all the power the battery can give"*.
- **Slope** of the surface. Uphill, downhill, or flat.

In this mission you'll write code so your 'bot can do **Dead Reckoning**.

- Sounds like a *great* title for a Zombie movie!
- But seriously, the term "dead reckoning" means ***navigating*** by moving a specific *direction*, *distance*, and *speed* from a known location.
- Perfect for those times when you don't have other sensors to guide you!

Mission Goals:

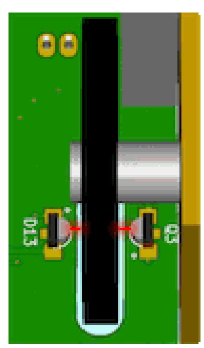
- Get to know CodeBot's 🔧Wheel Encoders.
- Write code to measure each wheel's *distance* traveled.
- Define a `drive()` function to move CodeBot an **exact distance**.
- Track distance over time, to measure the *speed* of your wheels.
- Calculate your 'bots **top-speed** in **"centimeters per second"**.
- Write **"Cruise Control"** code, to maintain a *set **speed*** over any terrain!
- Define a `rotate()` function that builds on your encoder code.

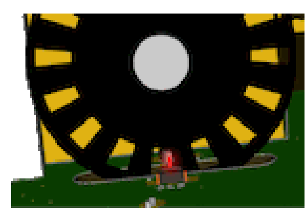
Navigate Ahead!

Objective 1 **- The Wheel Encoders**

Pick up your 'bot and turn it over!

Slowly rotate one of the wheels by hand, while observing the moving parts.





Take a close-up look at the 🔧wheel encoders.

- See how the **rotating disc** interrupts the *light beam* as it turns?
- The picture shows a *red* beam of light shooting across... but CodeBot uses an *infrared* LED, so it's invisible.

In this picture, **D13** is an 🔧LED (emitter) and **Q3** is a phototransistor (detector).

- As you might have guessed, your ***code*** has direct control of these parts!

💡 Concept: *Encoders API*

The `botcore` module provides a single function that:

- Activates the **emitter** LED.
- Reads the 🔧ADC value of the **detector**.
- Turns the emitter back off, and **returns** the 🔧integer value.

```
# Read the encoder's analog value.
val = enc.read(side)  # 'side' is LEFT or RIGHT
```

Just like the 🔧Line Sensors, the value returned is an 🔧integer between **0 and 4095**.

> The 🔧ADC has 12 🔧bits of *resolution* → 2^{12} = 4096 numbers.

With this sensor, **higher readings** indicate **more light** reaching the detector.

Lots **of Slots!**

As the *wheels* turn, **slots** in the *encoder disc* allow *pulses* of light to enter the *detector*.

- There are **20** slots in each of CodeBot's stock encoder discs.
- In a full rotation, the detector will see a ***"dark→light"*** transition 20 times.
- ... and it will see a ***"light→dark"*** transition 20 times too!

So with 20 slots, that's **40 "events"** your code can detect.

- Since a full rotation is *360*°, that's 360/40 = **9° measurement resolution**.

Create a New File!

Use the **File** → **New File** menu to create a new file called ***EncoderTest***.

Check the 'Trek!

Write code that calls `enc.read(LEFT)` repeatedly, and `print()`s the values to the 🔧**console**.

Run It!

Open the *Console.*

- Can you see the values change as you ***slowly*** rotate the *LEFT* wheel?
- What ***range*** of values do you observe?
  - Fully "open slot"? Fully "closed"?
- Moving less than **9°** is harder than you'd think!

CodeTrek:

```
from botcore import *
from time import sleep

while True:
    val = enc.read(LEFT)
    print(val)
    sleep(0.5)
```

Goals:

- **Assign** the value *returned* by `enc.read(LEFT)` to the 🔧variable `val`.
- 🔧Print the **variable** `val`.

Tools Found: Wheel Encoders, LED, Analog to Digital Conversion, int, Line Sensors, Binary Numbers, Print Function, Variables

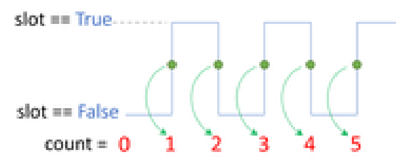
Solution:

```
from botcore import *
from time import sleep

while True:
```

```
    val = enc.read(LEFT)
    print(val)
    sleep(0.5)
```

Objective 2 **- Check Your Pulse!**

Slot: `True` *or* `False`?



You can sense the presence/absence of the *encoder disc* **slot**.

- Just a little **threshold** comparison code can make a bool out of that!

The picture to the right shows how your code would read `True` and `False` as the wheel turns.

- Increment `count` when your bool `slot` changes `False`→`True`
- ...and **also** when it changes back `True`→`False`!

Keeping up with *state*

This code will need to *remember* what **state** each *encoder disc* was last in.

- The sensor only tells you where it is *now*, not where it was *before*.
- But your *code* can **save state** in variables or lists.
- Then you can compare the *current state* versus *previous state*.

Ex: - using the *"Not equal to"* comparison operator `!=`

```
slot = sense_slot()      # Read current state
if enc_state != slot:    # Compare to previous state
    # Disc has moved...
```

Check the 'Trek!

Modify your code as follows:

- Create a new function called `sense_slot()`.
  - It should check your `enc.read()` value against a **threshold**, and *return* a bool.
- Define a variable called `enc_state`.
  - It will hold the last known *"slot"* ***state*** of the encoder disc.
- Define a variable called `enc_count`.
  - Increment this counter when `enc_state` changes.
- In your main `while` loop:
  - Sense the slot
  - If it's different than the last known state, *increment* and *print* the **count**!
- Remove the `sleep()` from your code!

Run It!

Give this code a ***spin!***

- Watch the *Debug Console* while it runs.
- You should see the `count` increasing as you turn the LEFT wheel.

Tune your `THRESH`!

You may need to *adjust* the value to make the count reliably change *only* when the wheel has moved.

Test your code!

Try turning the wheel *slowly* while observing the changing `count`.

- Hey, you've now coded an ***optical rotary encoder!***
- ...Like the ***Control Knob*** of a *massive sound system!*

What happens when you spin the wheel ***backwards?***

CodeTrek:

```python
from botcore import *

THRESH = # TODO: set a thresh value from your experiments.
```

Set a value somewhere *between* the high and low reading you *observed* in the last objective!

- *My* readings were between `300` and `3100`, so I'm going to pick `1000`!

```python
THRESH = 1000
```

Remember, a **high** reading means *more* light got through, indicating there was a ***slot!***

```python

def sense_slot():
```

`sense_slot()` returns `True` when a slot is **detected.**

- *In other words,* when the sensor reading is **high!**

```python
    val = enc.read(LEFT)
    # TODO: return True if val is greater than thresh
```

You can do this *in a single line!*

- Use the comparison 🔧operator `>`.
- `return val > THRESH`

```python

# Track encoder state: True if in slot.
enc_state = sense_slot()  # get initial position.
```

You **need** to know the position of the 🔧wheel encoder *on program run* in order to **accurately** track *how much* the disc has moved.

```python
enc_count = 0
```

`enc_count` *keeps track* of how many times `enc_state` **changes!**

```python

while True:
    slot = sense_slot()
    if enc_state != slot:
        # Disc has moved!
        enc_state = slot
        enc_count = enc_count + 1
```

When `enc_state != slot`, the **slot state** has *changed!*

- Save the *new* slot state to `enc_state`.
- 🔧Iterate `enc_count`!

```python
        print(enc_count)
```

Goals:

- **Assign** a value to the variable `THRESH`.
- **Initialize** the following variables:
- `enc_state`
- `end_count`
- **Define** a function called `sense_slot()`.
- In `sense_slot()`, **return** `True` if `val` is *greater than* `thresh`.

Tools Found: Comparison Operators, bool, Variables, list, Functions, Math Operators, Wheel Encoders, Iterable

Solution:

```
from botcore import *

THRESH = 1000   # Value from your experiments.

def sense_slot():
    val = enc.read(LEFT)
    return val > THRESH

# Track encoder state: True if in slot.
enc_state = sense_slot()  # get initial position.
enc_count = 0

while True:
    slot = sense_slot()
    if enc_state != slot:
        # Disc has moved!
        enc_state = slot
        enc_count = enc_count + 1
        print(enc_count)
```

Objective 3 - Sensing Both Wheels

With a little Python code you've brought an *encoder* to life.

- But just the ***LEFT*** wheel so far.
- That's easy to fix - but try to do so without *copying* a bunch of code.
    - Remember *"DRY" - Don't Repeat Yourself!*

Pro Tip:

Check the 'Trek!

Run It!

Give *both* wheels a *whirl!*

- You should see the `[LEFT, RIGHT]` `enc_count` values *streaming* by on the *Debug Console*.

Double the Fun!

But hey! **Not** ***double the code!***

- By using functions, parameters, and lists you kept your code just about as simple as the *one-wheel* version.

CodeTrek:

```
from botcore import *

THRESH = 1000   # Value from your experiments.

def sense_slot(side):
    val = enc.read(side)
```

Update your `sense_slot` function to take `side` as an argument.

- You'll be able to pass through `LEFT` or `RIGHT` and read their *respective* values.

```
    return val > THRESH

```

```
# Track encoder state (L,R). Store initial sens_slot() values.
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
```

Make `enc_state` *and* `end_count` lists!

- You'll be able to *easily* keep track of the **left** and **right** values by ***indexing*** the lists with the `LEFT` and `RIGHT` constants.

```

def check_enc(side):
    slot = sense_slot(side)
```

Onto your **new** function, `chec_enc(side)`!

- It has the parameter `side`, just like `sense_slot(side)`!
- It **replaces** your code to count disc movement from the *previous* objective.

```
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        # TODO: iterate the count of the side in enc_count
```

In order to iterate the **correct** value in the array, you'll need to ***index*** it by the `side`!

- `enc_count[side] = enc_count[side] + 1`

```
        return True

    # No movement
    return False


while True:
    left_moved = check_enc(LEFT)
    right_moved = check_enc(RIGHT)
    if left_moved or right_moved:
        print(enc_count)
```

If *either* wheel's **slot state** changes, print the `enc_count`.

Goals:

- Add the **parameter** `side` to your function `sense_slot()`.
- **Define** a new function called `check_enc(side)`.
- *On slot detection,* iterate the count of the `side` *in* `enc_count`.

Tools Found: list, Variables, Wheel Encoders, Parameters, Arguments, and Returns, Functions, Iterable, Keyword and Positional Arguments, Constants, Print Function

Solution:

```
from botcore import *

THRESH = 1000   # Value from your experiments.

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

# Track encoder state (L,R). Store initial sens_slot() values.
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
```

```python
enc_count = [0, 0]

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False


while True:
    left_moved = check_enc(LEFT)
    right_moved = check_enc(RIGHT)
    if left_moved or right_moved:
        print(enc_count)

```

Quiz 1 - Checkpoint

Question 1: What happens to the 🔧variable `count` when you spin the wheel *backwards?*

✓ `count` ***increases,*** same as forwards!

✗ `count` stays the same.

✗ `count` ***decreases.***

Question 2: How much does `count` change with a ***full*** 360° rotation?

✓ `40`

✗ `20`

✗ `30`

✗ `80`

✗ `360`

Question 3: How many ***slots*** are there in the CodeBot's *encoder disc?*

✓ `20`

✗ `40`

✗ `30`

✗ `2`

Objective 4 - Measuring Distance

Have you ever seen a *surveyor* walking along with a *Measuring Wheel?*

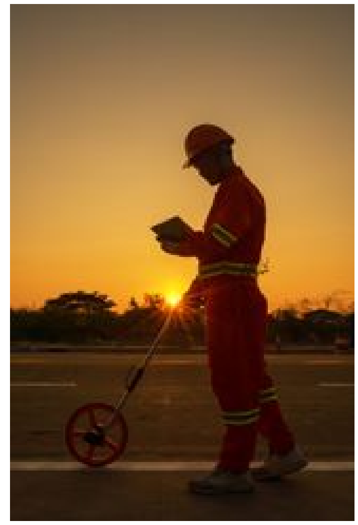
- They're measuring **distances** by tracking how far the wheel has turned.
- Can you measure *real distances* with your 🔧Wheel Encoders?

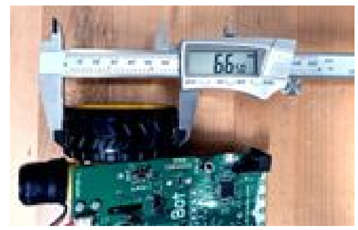
  Your mission for this step is to *measure distance traveled in* ***millimeters****!*

Around the Wheel in 40 Counts!

You know that 40 counts means the wheel has rotated 360°.

- So the distance the wheel moves across the ground is the same as the *circumference* of the wheel.





Grab your ruler!

CodeBot's standard wheels are **66.5**mm in **diameter**.

So the distance around the wheel is approximately:

$$circumference = 3.14 \times 66.5mm \approx 209mm$$

Concept: *Python's math Module*

You can't see it, but CodeBot's carrying around a *really fancy* ***scientific calculator!***

- Python provides a very rich set of **math** operations for your code to use when needed.
- And any scientific calculator worth its salt has a button for π!

Rather than defining your own 🔧constant to approximate *Pi*, you should use the one from the Python 🔧math module.

```
import math
WHEEL_DIA = 66.5
WHEEL_CIRC = (math.pi * WHEEL_DIA)
```

Do the Math

You know that the wheel moves **209 *mm* for every 40 *counts***.

- Write **"mm per count"** as a *fraction*.
- If you multiply **N counts** by that fraction, check out how the *units* of count *cancel out!*

$$D\ mm\ =\ N\ \cancel{counts} \cdot (\frac{209\ mm}{40\ \cancel{counts}})$$

*Armed with this knowledge, can you create a **Measuring Wheel** program?*

Check the 'Trek!

Modify your program to print the *distance* in *millimeters*.

- Define 🔧constants for:
  - the **wheel diameter** (66.5), *and*
  - ***counts per revolution*** (40).
- Use the *diameter* to calculate the **circumference** of your wheel.
- Define a 🔧function `counts_to_mm(counts)` that returns **mm** for a given number of **counts**.
- To make it more convenient, **reset** the count when **BTN-0** is pressed.

Run It!

Take your **Measuring *Wheels*** for a drive!

- Use a *ruler* or *tape measure* to verify your results.
- Hold the 'bot *steady* and click **BTN-0**...
- *Firmly and slowly* push it along the measured course.
  - Go at least *30 cm* and check your accuracy.

The wheel encoders are pretty *sensitive*, eh?

- Your code *counts* every change!
- Try *"jiggling"* a wheel and watch the *millimeters* clock by :-)
- But CodeBot's straight-line accuracy ***is*** pretty *impressive!*

Your Measuring Wheel is Rocking!

It's about time to get this thing *rolling...*

CodeTrek:

```
from botcore import *
import math
```

Import the math module to access the `math.pi` constant!

```

THRESH = 1000

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False


COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
```

Add `COUNTS_PER_REV` and `WHEEL_DIA` as constants!

```
WHEEL_CIRC = # TODO: calculate the wheel cirumference
```

Calculate the wheel **circumference** using the formula *in the instructions!*

- `WHEEL_CIRC = (math.pi * WHEEL_DIA)`

```

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV
```

`counts_to_mm` takes `count` as an argument and *returns* the distance in **mm!**

- You know a *full* wheel rotation will register `40` counts.
- You also know the wheel's **circumference!**

The rest is simple!

```

while True:
    left_moved = check_enc(LEFT)
    right_moved = check_enc(RIGHT)
    if left_moved or right_moved:
        print(enc_count)

        left_dist = counts_to_mm(enc_count[LEFT])
        right_dist = counts_to_mm(enc_count[RIGHT])
```

For *each* of the sides in `enc_count`, *translate* the **count** to **millimeters** using `counts_to_mm(count)`!

```
        print("Left Distance: ", left_dist, "mm")
        print("Right Distance: ", right_dist, "mm")
```

🔧Print the resulting ***translation!***

```

        # Reset the count if BTN-0 pressed.
        if buttons.was_pressed(0):
            enc_count = [0, 0]
```

Set `enc_count` to it's **default** on *BTN-0* press.

Goals:

- **Calculate** the **wheel circumference** and assign it to 🔧constant `WHEEL_CIRC`
- **Define** a new 🔧function called `counts_to_mm(counts)`.
- **Reset** the count when `BTN-0` is pressed.
- **Assign** the value of `counts_to_mm(enc_count[LEFT])` to the 🔧variable `left_dist` and then 🔧print it.

Tools Found: Wheel Encoders, Constants, Math Module, Functions, Variables, Print Function, import, Keyword and Positional Arguments

Solution:

```
from botcore import *
import math

THRESH = 1000

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

```

```

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

while True:
    left_moved = check_enc(LEFT)
    right_moved = check_enc(RIGHT)
    if left_moved or right_moved:
        print(enc_count)

        left_dist = counts_to_mm(enc_count[LEFT])
        right_dist = counts_to_mm(enc_count[RIGHT])
        print("Left Distance: ", left_dist, "mm")
        print("Right Distance: ", right_dist, "mm")

        # Reset the count if BTN-0 pressed.
        if buttons.was_pressed(0):
            enc_count = [0, 0]


```

Objective 5 - Driving Forward

***Oh wait!* I just realized... The Wheel Encoders are connected to Motors!!**

So the next step should be pretty obvious: Write a function to ***drive* a specified *distance***.

```
# Drive forward for 50 centimeters
drive(50)
```

So **simple**, so ***elegant!***

- And *now* you have the *Python* + *Robotics* ***skills*** to make it happen.

Check the 'Trek!

Modify your code to implement the *Simple and Elegant*™ function, `drive(cm)`.

- Define a function `mm_to_counts(mm)` that returns **counts** for a given distance in *millimeters*.
  - Looks a lot like a function you *already have!*
  - You'll use this to calculate the `count` based on requested `drive()` distance.
- Move your `while` loop into a new function called `drive(cm)` which moves the 'bot **cm** *centimeters* when called.
  - For now, `run()` both motors with a constant *power* of 50%.
  - Continuously call `check_enc()` for both sides.
  - `break` from loop and **stop** moving if `LEFT` ***or*** `RIGHT` wheels have gone >= `count`.
- In your *main program*:
  - Just **enable the motors** and call `drive()`!
  - *Oh yeah, don't forget to **wait for a button-press** before moving!*
- **Pro-Tip: *Organize*** your code in *sections* from top to bottom: *imports, constants, functions, global variables, main program.*
  - This is *optional*, but it will make your code much more *readable!*

Run It!

Can you `drive()` an *exact distance?*

- Try different distances in *centimeters.*
- Is your 'bot going precisely the **distance** you specify?
- Is it going in a *straight line?*

Test, Observe, Modify

You're probably already thinking of ways to improve this code!

- Does your `drive()` function tend to *overshoot* or *undershoot* the specified *distance*?
- Would it be *better* or *worse* to wait until both **LEFT *and* RIGHT** wheels are `>= count` before *stopping?*
- Experiment with the **power** value you set the motors to as well.

CodeTrek:

```
from botcore import *
import math

# -- Constants --
COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000
```

Move your constants to the *top* of the file.

- Good organization
  reduces mistakes!

```

# -- Functions --

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    # TODO: return translate mm to counts
```

`mm_to_counts(mm)` needs to return `counts` *from* **millimeters!**

- `return mm * COUNTS_PER_REV / WHEEL_CIRC`

```

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm):
    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
```

Translate `cm` to `mm` by multiplying by `10`!

- `mm_to_counts(mm)` *returns* the number of `counts` you need to drive forward.
- You'll know when to **stop** the motors when `counts` is *equal to* the number of ***slots detected!***

```

    # Start moving
    motors.run(LEFT, 50)
    motors.run(RIGHT, 50)

    # Keep going until 'count' reached
    while True:
```

Move the `while` loop into this *new* function!

```
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)
```

Continuously call `check_enc()` for ***both sides.***

```
        if left_moved or right_moved:
            print(enc_count)

            left_dist = counts_to_mm(enc_count[LEFT])
            right_dist = counts_to_mm(enc_count[RIGHT])
            print("Left Distance: ", left_dist, "mm")
            print("Right Distance: ", right_dist, "mm")

            # Are we there yet??
            if enc_count[LEFT] >= count or enc_count[RIGHT] >= count:
                break
```

If the `LEFT` *or* `RIGHT` wheel encoders have sensed `count` rotations *or more*, **stop** the motors!

```

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

# -- Global variables --
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]


# -- Main program --

# Wait for BTN-0. Good robot.
while True:
    if buttons.was_pressed(0):
        break
```

Wait for a button press ***before*** *moving!*

```

motors.enable(True)
```

Enable your motors *before* calling `drive(cm)`.

```

# Gonna take a centimeter journey...
# TODO: drive 50 cm!
```

Hold on *tight!*

- Start with `50` centimeters, but feel free to *experiment!*

  ```
  drive(50)
  ```

Goals:

- **Define** a function `mm_to_counts(mm)`.
- Translate **millimeters** to **counts** and `return` the value from `mm_to_counts(mm)`

- **Define** a function `drive(cm)`.
- **Call** `drive(50)`.

Tools Found: Wheel Encoders, Motors, Functions, Constants, Loops

Solution:

```
from botcore import *
import math

# -- Constants --
COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000 #@1

# -- Functions --

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm):
    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10) #@3

    # Start moving
    motors.run(LEFT, 50)
    motors.run(RIGHT, 50)

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)
        if left_moved or right_moved:
            print(enc_count)

            left_dist = counts_to_mm(enc_count[LEFT])
            right_dist = counts_to_mm(enc_count[RIGHT])
            print("Left Distance: ", left_dist, "mm")
            print("Right Distance: ", right_dist, "mm")

            # Are we there yet??
            if enc_count[LEFT] >= count or enc_count[RIGHT] >= count:
                break #@4

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

# -- Global variables --
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]


# -- Main program --
```

```

# Wait for BTN-0. Good robot.
while True:
    if buttons.was_pressed(0):
        break #@5

motors.enable(True)

# Gonna take a centimeter journey...
drive(50) #@6


```

Objective 6 **- Repeat the Journey**

Your code is good, but as it stands you have to **re-start** the code or reboot the 'bot each time you start a new journey!

- It would be nice to just press **BTN-0** to go again.
- That doesn't sound too hard - ***to the code!***

Check the 'Trek!

Modify your code to move the whole `# -- Main program --` section into an *outer* `while True:` loop.

- So rather than ending, your program just loops back to the **BTN-0** check again.

  (Remember, you can ***select*** *a block of code and hit* ***TAB*** *to indent it.)*

Run It!

Take this code for a *test drive!*

- Does it repeat as expected?

Caution: *Bug Alert*

Okay, maybe it wasn't ***that*** simple!

- You're going to need to debug this code.

Start by watching your `print()` output on the **Debug Console**.

- The first time you run, it works okay.
- *Observe* the **count** values on the *second* run.

CodeTrek:

```
from botcore import *
import math

# -- Constants --
COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

# -- Functions --

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV
```

```

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm):
    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)

    # Start moving
    motors.run(LEFT, motor_power[LEFT])
    motors.run(RIGHT, motor_power[RIGHT])

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)
        if left_moved or right_moved:
            print(enc_count)

            left_dist = counts_to_mm(enc_count[LEFT])
            right_dist = counts_to_mm(enc_count[RIGHT])
            print("Left Distance: ", left_dist, "mm")
            print("Right Distance: ", right_dist, "mm")

            # Are we there yet??
            if enc_count[LEFT] >= count or enc_count[RIGHT] >= count:
                break

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

# -- Global variables --
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]


# -- Main program --

# Outer loop - repeat forever.
while True:
```

Add *another* `while` loop *around* the ***main program!***

- It'll still *wait* for *BTN-0* to be pressed *before* running each time!

```

    # Wait for BTN-0. Good robot.
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Gonna take a centimeter journey...
    drive(50)

```

Hit ***TAB*** on your keyboard to *quickly* indent selected code!

Goal:

- Move the whole `# -- Main program --` under an *outer* `while True:` loop.

Tools Found: Reboot, Indentation, Debugging

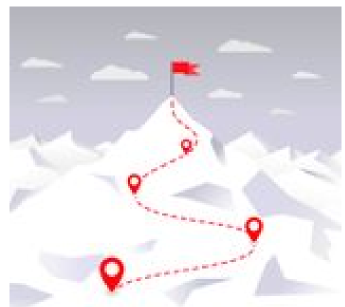
Solution:

```
from botcore import *
import math

# -- Constants --
COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

# -- Functions --

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm):
    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)

    # Start moving
    motors.run(LEFT, motor_power[LEFT])
    motors.run(RIGHT, motor_power[RIGHT])

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)
        if left_moved or right_moved:
            print(enc_count)

            left_dist = counts_to_mm(enc_count[LEFT])
            right_dist = counts_to_mm(enc_count[RIGHT])
            print("Left Distance: ", left_dist, "mm")
            print("Right Distance: ", right_dist, "mm")

            # Are we there yet??
            if enc_count[LEFT] >= count or enc_count[RIGHT] >= count:
                break

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)
```

```

# -- Global variables --
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]


# -- Main program --

# Outer loop - repeat forever.
while True:

    # Wait for BTN-0. Good robot.
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Gonna take a centimeter journey...
    drive(50)

```

***Objective 7* - Repeat the Journey 2**

No wonder it doesn't work. The 'bot thinks it's already *gone the distance!*



Okay, so on each re-start you need a way to **reset** the *starting line!*

- One option is to set `enc_count` back to `[0, 0]`.
  - That would work, but then you'd lose the "total distance traveled" - which might be nice to have at some point.
- Another option is to **save the starting count** each time `drive()` is called.
  - To check how far you've moved, just subtract `enc_count[xx] - start_count[xx]`.

Take the second option: Save the `start_count[]` at the beginning of the `drive()` function.

- To do that, you need to **copy** the contents of the 🔧list `enc_count`.

Concept

When copying a 🔧list you might be tempted to simply write:

```
# This will NOT make a new list!
start_count = enc_count
```

But that will only make a new 🔧variable `start_count` that ***references*** the exact same `enc_count` 🔧list you already had!

- In Python, normal *assignment* doesn't **copy** *objects* like 🔧lists.
- It only gets you a **reference** to the existing object.
  - That means if `enc_count[LEFT]` changes, so does `start_count[LEFT]`.
  - They're always equal, since they both refer to the same list!

To copy a 🔧list :

```
# Use the copy() method!
start_count = enc_count.copy()
```

Check the 'Trek!

Modify your code to save the *starting line*.

These changes are all inside the `drive()` function.

- At the beginning of the function, **copy** the `enc_count` to a new 🔧list called `start_count`.
- If `left_moved or right_moved` then...
  - Calculate the new count offsets from the starting line.
  - You might call them `count_left` and `count_right`.
- Use those new *count offsets* instead of `enc_count[LEFT]` and `enc_count[RIGHT]` when you print and check the distances.

Run It!

Time for another ***test drive.***

- Hopefully your results are better this time!
- Can you make multiple ***journeys*** just by pressing **BTN-0** again?

Test Your Machine!

Which of your motors is faster?

- They're usually **not** *exactly* the same!
- *Perhaps* you could write **code** to make them run the same speed, though...

Notice any other problems?

- You may see **BTN-0** trigger an occasional *double-trip!*
- Ah, it's your old friend, *contact bounce.*

You already know about **debouncing** buttons!

- Add a call to `buttons.was_pressed(0)` some time after the first click to clear any *extras* that occur.
- Rather than adding a `sleep()` delay, you can just discard any button presses that happen *during the journey.*

You can add the *debounce* **after** your call to `drive(xx)`, ***or*** just above your *button-check* loop:

```
while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
        break
```

You, my friend, are going to go *far!*

CodeTrek:

```
from botcore import *
import math

# -- Constants --
COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

# -- Functions --

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm):
```

```
    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)

    # Save the starting line.
    start_count = # TODO: copy enc_count
```

At the **beginning** of a `drive(cm)` call, create a copy of the `enc_count` list!

- `start_count = enc_count` doesn't make a ***new*** copy, it just references the *already existing* list!
- Make a **copy** using `enc_count.copy()`.

```
start_count = enc_count.copy()
```

```

    # Start moving
    motors.run(LEFT, 50)
    motors.run(RIGHT, 50)

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)
        if left_moved or right_moved:
            print(enc_count)


            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]
```

Calculate the *distance* from the position your 'bot was in when `drive(cm)` was *called* by subtracting the 'bot's *current* position from it's *starting* position!

```


            left_dist = counts_to_mm(count_left)
            right_dist = counts_to_mm(count_right)
```

Substitute `enc_count[LEFT]` and `enc_count[RIGHT]` with `count_left` and `count_right`!

```
            print("Left Distance: ", left_dist, "mm")
            print("Right Distance: ", right_dist, "mm")

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break
```

Don't forget to substitute them here too!

```

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

# -- Global variables --
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]


# -- Main program --

# Outer loop - repeat forever.
while True:

    # Wait for BTN-0. Good robot.
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)
```

```

    # Gonna take a centimeter journey...
    drive(50)


```

Goals:

- **Assign** the variable `start_count` as a **copy** of `enc_count` using the `list.copy()` function.
- **Assign** the variable `count_left` as a *subtraction* of `enc_count[LEFT]` and `start_count[LEFT]`.

Tools Found: list, Variables, Motors

Solution:

```
from botcore import *
import math

# -- Constants --
COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

# -- Functions --

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm):
    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # Start moving
    motors.run(LEFT, 50)
    motors.run(RIGHT, 50)

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)
        if left_moved or right_moved:
            print(enc_count)


            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]


            left_dist = counts_to_mm(count_left)
```

```
            right_dist = counts_to_mm(count_right)
            print("Left Distance: ", left_dist, "mm")
            print("Right Distance: ", right_dist, "mm")

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

# -- Global variables --
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]


# -- Main program --

# Outer loop - repeat forever.
while True:

    # Wait for BTN-0. Good robot.
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Gonna take a centimeter journey...
    drive(50)

```

Objective 8 **- Speed-o-Meter**

Get your 'bot Up to *Speed!*

Now that you can measure **distance**, the next step is to measure your ***speed.***

Why would you want to do that?

- To drive with **consistent speed**, regardless of *battery* level or *changing terrain*.
  - Stop using **constant % power** and start ***Sensing your Speed!***
- To drive in a **straight line**.
  - You'll need to make *both wheels* go exactly the ***same speed***.

Your first step is to write code to monitor and display the *speed* of each wheel.

What's CodeBot's *top speed?*

- ...would that be in: *Miles per Hour?, Kilometers per Hour?, Feet per Second?, Centimeters per Second?*
- Actually *all* of those are valid units of **speed**.

Replacing the word *"per"* with **division** shows you the equation for *speed*:

$$speed = \frac{distance}{time}$$

You've got the **distance** part covered with the code you just finished.

- Now you just have to keep track of **time** as your 'bot moves!

Just in *Time*

You've been using Python's 🔧time module to access the `sleep()` function. But it has *much more* to offer!

- Your `while` loop is calling `check_enc()` *very rapidly*, every time through the loop.
- Is there a way to quickly check ***how much time has elapsed*** also?
- Yes! Check out the `ticks_ms()` function in the 🔧time module.
- Use it to capture the current *time-tick count* in milliseconds.

Ex: - measure *milliseconds* between `t_start` and `t_stop`.

```
import time

t_start = time.ticks_ms()
# Do some stuff that takes time...
t_stop = time.ticks_ms()

t_diff = time.ticks_diff(t_stop, t_start)
print("That took ", t_diff, " milliseconds!")
```

Algorithm for Speed Sensing

1. Use `ticks_ms()` to check the elapsed time.
2. Use 🔧global variables `last_ms` and `last_count` to save millisecond and encoder counts.
3. Every 100ms interval, do the speed calculation:
4. Calculate distance based on current `enc_count` minus `last_count`.
5. Calculate $speed = \frac{distance}{100ms}$
6. Use a *global* 🔧list to store the current `cur_speed[LEFT]` and `cur_speed[RIGHT]`.
   - Keep the **speed** values in *"counts per second".*
   - ...you can convert *counts* to *distance* later.

Check the 'Trek!

You'll be defining *two* new 🔧functions

- `def update_speed(interval_ms):`
  - Checks the elapsed time interval and updates the global ***cur_speed*** list.
- `def print_speed_cps():`
  - Converts ***cur_speed*** from "counts per second" to "cm/s" and *prints* it.

Do this in *two* stages.

For ***Stage-1***, don't worry about *calculating the speed*. Just get the 🔧functions set up, and *organize* your code a bit.

1. Modify your `drive()` function by moving the 4 lines that *print the distance* to the **end**.
   - Print *"Total distance traveled."*
   - Remember you can *select* a block of lines and use 🔧editor shortcuts to *cut and paste* it where you want it.
   - You can use *SHIFT + TAB* to *unindent* a block of code too!
2. Add a call to `update_speed(100)` in your `drive()` function.
   - Just *before* the `if left_moved or right_moved:` inside your `while True:` loop.
3. Define a new 🔧function `print_speed_cps()`.
   - In *Stage-2* this will print *"centimeters per second..."*
   - For now just print the current `ticks_ms()` count.
   - This will be called every *100ms.*
4. Define a new 🔧function `update_speed(interval_ms)`.
   - No need to calculate speed yet.
   - For now, this function's job is just to call `print_speed_cps()` every **100ms**.
   - Check elapsed time: `t_ms = ticks_diff(ticks_ms(), last_ms)`.
   - Don't forget to declare the `last_ms` as 🔧global in your function.
     - ...and to set it to the current `ticks_ms()` value at the start of the next interval!

Run It!

Watch the **Debug Console** as you run this.

- You should see the **ms ticks count** printed every ***100ms***.
- If things are working as planned, those *counts* should be in steps of 100!
  - `ticks_ms()` uses *very accurate* hardware-based timers. But your *code* can 🔧branch based on `check_enc()`, so you may see the **ticks** count skip a few *milliseconds* occasionally.

CodeTrek:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff
```

🔧Import `ticks_ms` *and* `ticks_diff` from the 🔧time module!

```

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm):
    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # Start moving
    motors.run(LEFT, 50)
    motors.run(RIGHT, 50)

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        update_speed(100)
```

Add a call to `update_speed()`.

- You'll **define** this *new* 🔧function *shortly!*

```

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
```

```
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")
#@1

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms
```

Define `update_speed(interval_ms)`.

- Declare `last_ms` as a 🔧global, you'll be *altering* it in this 🔧function!

```

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
```

`t_ms` is a measurement of the *amount* of time that's elapsed since `last_ms`.

- `tick_diff(ticks1, ticks2)` returns the **difference** between *two* ticks!

```
    if t_ms >= interval_ms:
        last_ms = ticks_ms()
        print_speed_cps()
```

If *more* time has elapsed than the `interval_ms`:

- Update `last_ms` to **now**.
- Call your 🔧print function!

```

def print_speed_cps():
    ticks = # TODO: assign the current ticks count
```

For now, `print_speed_cps()` just needs to print the *current* ticks.

- `ticks = ticks_ms()`

```
    print("ticks = ", ticks)

# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
```

Initialize a new 🔧global variable, `last_ms`!

- You'll use this to **keep track** of the last time your *speed* was reported.

```

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Go forth!
    drive(30)
```

Goals:

- **Define** a new function `print_speed_cps()`
- **Define** a new function `update_speed(interval_ms)`

- **Call** `update_speed(100)` in your `drive(cm)` function.
- **Assign** the 🔧variable `t_ms` as the value *returned by* `ticks_diff(ticks_ms(), last_ms)`

Tools Found: Time Module, Locals and Globals, list, Functions, Editor Shortcuts, Branching, Variables, Print Function, import, Timing

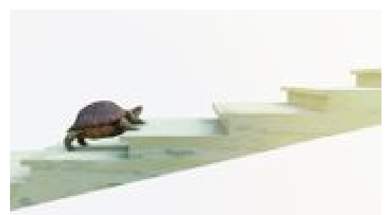
Solution:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm):
    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # Start moving
    motors.run(LEFT, 50)
    motors.run(RIGHT, 50)

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
```

```
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        last_ms = ticks_ms()
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    print("ticks = ", ticks_ms())

# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Go forth!
    drive(30)
```

Objective 9 **- Speed-o-Meter 2**

Ready for the next *Design Iteration?*



Professional code is usually developed using an *iterative process* like this.

- **Iterate** just means to do something *repeatedly.*
- You're taking small steps that *build* to the *whole solution*.

Check the 'Trek!

Stage-2 - *Speed Indeed!*

This stage completes your *Speedometer!*

1. Define new global variables: `last_count = [0, 0]` and `cur_speed = [0, 0]`.
2. Write the code for your `update_speed()` function so at *every interval* it:
   - Calculates distance traveled = `enc_count[xx] - last_count[xx]` for each wheel.
   - Resets `last_count = enc_count.copy()`.
   - Calculates speed in *"counts per second"* for each wheel.
     - You'll need to convert *milliseconds* to *seconds*:
       - $t_{sec} = t_{ms} \cdot \frac{1sec}{1000ms}$
   - Updates the global `cur_speed[]`.
3. Write the code for your `print_speed_cps()` function.
   - Use the global `cur_speed[]` which is in *"counts per second"*.
   - Your `counts_to_mm()` function will convert that to a *real* distance!
   - Then remember to convert *mm* to *cm*, and print *"cm/s"*.
   - Print it!

▷ Run It!

Give this a try, and ***check your speed!***

- Watch the 🔧**console**.
- Does the **speed** shown match your expectations?
  - Divide the *"Total distance traveled"* by the time it takes for a *run* to get an approximate *speed* for comparison.
- Test this with different **motor *power*** values.
- Try putting a little "friction" on one wheel, while the other is free.
  - Make sure you see the *slower* wheel print *lower* speeds!

CodeTrek:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm):
    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # Start moving
    motors.run(LEFT, 50)
    motors.run(RIGHT, 50)

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

```

```
    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")


def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count
```

Add `last_count` as a global.

```

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
```

Calculate the distance travelled *since the last interval*.

- `last_count` gets updated *every interval*, comparing it against the *current count* (`env_count[side]`) will tell you the distance travelled in the *current interval!*

```
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = # TODO: make a copy of enc_count
```

Make a **copy** of `enc_count`!

- You've done this in *previous* objectives, if you're confused go back and **review!**

```
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
```

Calculate the current **counts per second** by dividing the distance travelled by the interval!

- *Reminder,* $speed = \frac{distance}{time}$

```
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
```

Translate counts per second into **centimeters per second** by using your `counts_to_mm(counts)` function, then *dividing* by `10` to get **cm!**

```
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")
```

***Print** it!*

```python

# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
```

Instantiate two *new* 🔧variables:

- `last_count` will be used *similarly* to `start_count`!
- `cur_speed` will be used to store the... *current speed!*

```python


while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Go forth!
    drive(30)
```

Goals:

- **Define** new **global** 🔧variables `last_count` and `cur_speed` as `[0, 0]`.
- **Assign** the variable `last_count` as a **copy** of `enc_count` using the `list.copy` 🔧function.
- **Assign** the variable `cps_left` as the **cm per second** using `counts_to_mm(cur_speed[LEFT]) / 10`

Tools Found: Locals and Globals, Print Function, Variables, Functions

Solution:

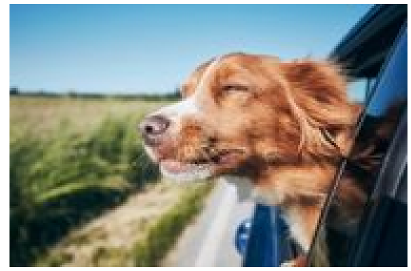
```python
from botcore import *
import math
from time import ticks_ms, ticks_diff

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm):
    # Convert centimeters to counts.
```

```
    count = mm_to_counts(cm * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # Start moving
    motors.run(LEFT, 50)
    motors.run(RIGHT, 50)

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")


def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")


# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"


while True:
    # Wait for BTN-0. Good robot.
```

```python
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Go forth!
    drive(30)
```

Objective 10 **- Cruise Control**

This step is going to be a *breeze!*

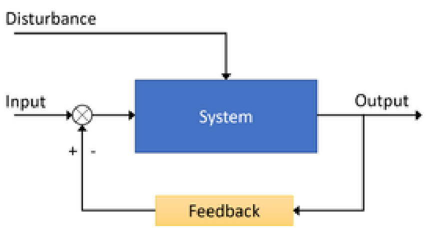


Wouldn't it be nice to tell your 'bot the **speed** you want to go and have it *automatically* maintain that speed, like the *cruise control* in a car?

- Up to this point changing **speed** has meant *experimenting* with **% power** settings for the 🔧motors.
- But now, with your new *speedometer* capability, you can *automate* that!

Process Control System

Your *cruise control* code will use a ***fundamental engineering concept*** that powers a lot of the modern technology you depend on every day.



Closed-Loop Control System

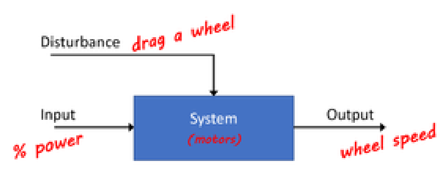
Closed Loop Control automates control of a *System* by sensing the *Output* state and comparing it to the desired state (*Input*). A *Feedback* loop continuously adjusts the *System* to keep the *error* (difference between *Input* and *Output*) close to *zero*.

- **Input** → Desired Speed
- **System** → 🔧Motors
- **Output** → Actual Speed
- **Feedback** → 🔧Wheel Encoders
- **Disturbance** → Friction, terrain, etc.

Your Code is *"Open Loop!"*

Right now you're *sensing* the **speed**, but your code is **not** using it to adjust the **power**.

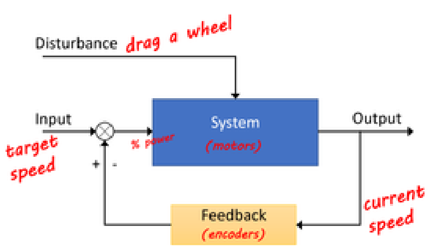
- Any *"Disturbance"* that happens will affect the *Output* (speed).
- And your *Input* in *raw* ***% power*** is only *loosely* related to the *Output* ***speed***.



Open-Loop Control

CodeBot Cruise Control

This is the *control system* you'll be coding.



Closed-Loop Control

- You're already *sensing* the `cur_speed`.
- For *Input* how about: `drive(distance, speed)` ?
- Your *Feedback* loop will calculate the error:

$$err = (Input - Output) \cdot F_{pwr}$$

- *Output* and *Input* are **speeds**.
- F_{pwr} is a 🔧constant *you choose* to set how *strong* the feedback is.

Feedback with *Code*

Your ***feedback loop*** needs to measure the **error** between *Input* and *Output*, and *feed* it back to the *System*.

- **Input** → `target_speed`.
- **Output** → `cur_speed[xx]`.
- **System** → `power[xx]` to the 🔧motors.

  Ex: Code to apply *feedback* for LEFT side.

```
# Calculate: err = (Input - Output) * Fpwr
err = ( target_speed - cur_speed[LEFT] ) * FEEDBACK_PWR

# Apply feedback to System (adjust motor power)
power[LEFT] = power[LEFT] + err
motors.run(LEFT, power[LEFT])
```

Consider how the code above works when your 'bot is going **slower** than the desired `target_speed`:

- `target_speed > cur_speed[LEFT]` so `err` will be ***positive***.
- Which means `power` to the motors will ***increase***!

Ready to *Code* this?

Relax! You can implement this with just a few more lines of Python code!

Check the 'Trek!

Modify your code as follows:

1. Define new global variables: `target_speed = 0` and `power = [0, 0]`.
2. Add a second parameter `speed` to your `drive()` function.
   - This will be in units of *"centimeters per second"* (cm/s).
3. In `drive()` convert `speed` to *"counts per second"*.
   - Save this value in global `target_speed` to compare with `cur_speed[xx]` in your *feedback loop*.
4. Remove the `# Start moving` code from `drive()`.
   - Control of the motors will be handled in a new function.
5. Define a new function `def control_speed(side):` to implement your **feedback**.
   - Calculate `err = (target_speed - cur_speed[side]) * FEEDBACK_PWR`
   - Start with `FEEDBACK_PWR = 0.1` to translate *speed error* to *power correction*.
   - Adjust `power[side]` based on `err`, and set `motors.run(side, power[side])`.
   - Keep `power[side]` in range of ±100%.
6. Call the `control_speed()` for *LEFT* and *RIGHT* sides from your `update_speed()` function.
   - Just before the call to `print_speed_cps()` is a good place to *invoke* it.

Run It!

Okay, give your ***Cruise Control*** a go!

- Try longer distance runs.
- Test it with *slow* and *fast* ***speeds!***
  - *Find your robot's **limits!***

Caution: *Quirky Code Note*

My code has a *quirk*. Okay, you might even call it a ***bug!***

- After the first run, my code *remembers* the `power` it was using before!
- That can make it *lurch* forward, instead of *ramping up the speed smoothly* like the first time.

You can ignore this bug for now - you'll fix it in the next step!

Testing a *Disturbance* to your *Control System*

Add a line of code to show the motor **% power** level while the 'bot is running.

Ex: - add just before calling `print_speed_cps()`

```
...
control_speed(LEFT)
control_speed(RIGHT)
print("Power: ", power)
print_speed_cps()
```

Set your code for a *long, slow cruise* like: `drive(100, 10)`.

- Hold your 'bot and watch the **Debug Console** when it runs.
- If you add *friction* to one wheel, do you see the **motor power** increase to correct the *error?*
- See your code *work hard* to achieve the `target_speed` ?
- When you *remove* the friction, does the **power** back off?

CodeTrek:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000
FEEDBACK_PWR = 0.1  # Impact of speed error on motor power.
```

Initialize constant `FEEDBACK_PWR`.

- You'll use it to control the **rate** that *speed error* is translated to *power correction*.

```

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm, speed):
```

Add `speed` to your `drive` function.

- `speed` is just your `target_speed` but in **centimeters per second!**

```
    global target_speed

    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
    target_speed = mm_to_counts(speed * 10)
```

Calculate and set the `global` variable `target_speed`.

- Convert `speed` to **millimeters** by multiplying by `10`.

```

    # Save the starting line.
    start_count = enc_count.copy()

    # [ Removed motors.run() "start moving" code ]
```

Remove your *"start moving"* code, you'll control the motors *later* with your *new* function, `control_speed`.

```

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
        control_speed(LEFT)
        control_speed(RIGHT)
```

Call your *new* 🔧function `control_speed(side)` for *each* side from your `update_speed(interval_ms)` function.

- Your *feedback loop* will be updated every `interval_ms`.

```
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")

def control_speed(side):
    # Set motor power to reach target speed.
    err =    # TODO: Calculate err = (Input - Output) * Fpwr
```

`err` is the value you'll *feed back* into your system!

- `target_speed` is the **input** and `cur_speed[side]` is the **output!**
- `err = (target_speed - cur_speed[side]) * FEEDBACK_PWR`

```
    pwr = power[side] + err
```

```python

    if pwr > 100:
        pwr = 100
    elif pwr < -100:
        pwr = -100
```

If the *result* of adding `err` to `power[side]` is **outside** the acceptable range, set it as the *closest* **in-range** value.

```python

    power[side] = pwr
    motors.run(side, pwr)
```

Run the motor at the *newly* calculated speed!

```python



# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
target_speed = 0   # Desired speed in "counts per second"
power = [0, 0]
```

Instantiate variables `target_speed` and `power`.

- `target_speed` is your *feedback loop's* **input**, it'll be in **counts per second**.
- `power` is the value you'll supply to the motors.

```python

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Drive (dist=cm, speed=cm/s)
    drive(100, 50)
#@10
```

Goals:

- **Assign** new global variables:
- `target_speed` as `0`
- `power` as `[0, 0]`
- **Define** a new function `control_speed(side)`.
- Calculate the **error** between *Input* and *Output* and **assign** the value to `err`.

Tools Found: Motors, Wheel Encoders, Constants, Locals and Globals, Functions, Variables

Solution:

```python
from botcore import *
import math
from time import ticks_ms, ticks_diff

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
```

```
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm, speed):
    global target_speed

    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
    target_speed = mm_to_counts(speed * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # [ Removed motors.run() "start moving" code ]

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
```

```
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
        control_speed(LEFT)
        control_speed(RIGHT)
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")


FEEDBACK_PWR = 0.1  # Impact of speed error on motor power.

def control_speed(side):
    # Set motor power to reach target speed.
    err = (target_speed - cur_speed[side]) * FEEDBACK_PWR
    pwr = power[side] + err

    if pwr > 100:
        pwr = 100
    elif pwr < -100:
        pwr = -100

    power[side] = pwr
    motors.run(side, pwr)



# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
target_speed = 0   # Desired speed in "counts per second"
power = [0, 0]

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Drive (dist=cm, speed=cm/s)
    drive(100, 50)
```

Objective 11 **- Slow Starts, Breaks, and Brakes!**

Check the 'Trek!

Run It!

When you run this code, you'll probably notice that it did ***not*** fix the problem!

- After the first run, the motors still *remember* the last `power` they were at!?

How can that be? You're setting `power = [0, 0]` right at the top of `drive()`!

Debugging

Debug

CodeTrek:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000
FEEDBACK_PWR = 0.1  # Impact of speed error on motor power.

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm, speed):
    global target_speed

    # TODO: Reset motors power to zero
```

Just like in the instructions!

- Reset the `power` variable to it's **default**.
- `power = [0, 0]`

```

    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
    target_speed = mm_to_counts(speed * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # [ Removed motors.run() "start moving" code ]

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break
```

```

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
        control_speed(LEFT)
        control_speed(RIGHT)
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")

def control_speed(side):
    # Set motor power to reach target speed.
    err = (target_speed - cur_speed[side]) * FEEDBACK_PWR
    pwr = power[side] + err

    if pwr > 100:
        pwr = 100
    elif pwr < -100:
        pwr = -100

    power[side] = pwr
    motors.run(side, pwr)



# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
target_speed = 0   # Desired speed in "counts per second"
power = [0, 0]

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Drive (dist=cm, speed=cm/s)
    drive(100, 50)
```

Goals:

- At the beginning of the `drive` function, *reset* the `power` 🔧variable to it's **defualt** (`[0, 0]`).
- **Step Into** the program using the 🔧debugger.

Tools Found: Functions, Motors, Variables, Advanced Debugging

Solution:

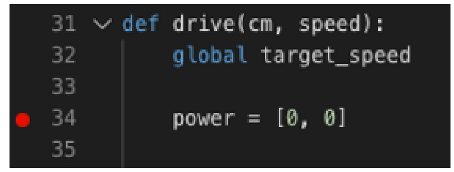
```
from botcore import *
import math
from time import ticks_ms, ticks_diff

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm, speed):
    global target_speed

    power = [0, 0]

    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
    target_speed = mm_to_counts(speed * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # [ Removed motors.run() "start moving" code ]

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Stop moving
    motors.run(LEFT, 0)
```

```
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
        control_speed(LEFT)
        control_speed(RIGHT)
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")


FEEDBACK_PWR = 0.1  # Impact of speed error on motor power.

def control_speed(side):
    # Set motor power to reach target speed.
    err = (target_speed - cur_speed[side]) * FEEDBACK_PWR
    pwr = power[side] + err

    if pwr > 100:
        pwr = 100
    elif pwr < -100:
        pwr = -100

    power[side] = pwr
    motors.run(side, pwr)



# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
target_speed = 0   # Desired speed in "counts per second"
power = [0, 0]

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Drive (dist=cm, speed=cm/s)
    drive(100, 50)
```

***Objective 12* - Breakpoints**

Introducing *Breakpoints*

A **breakpoint** is a *marker* you can place on any *executable* line of code.



- **Debug** the code, press CONTINUE, and it will **STOP** when it hits a *breakpoint!*
- Then you can *inspect your* *variables* and either:
  - STOP the program,
  - STEP into the *next* lines of code, or
  - CONTINUE running the code until it ends or hits the next *breakpoint,* or
  - STEP OVER and STEP OUT which you'll learn about in *a different mission!*

Notes:

- You can only set *breakpoints* when your CodeBot is ***connected*** and ***stopped***.
- Click in margin to the **left of the line number** where you want the *breakpoint.*
- To **remove** a breakpoint, just click on its *red dot* symbol.
- CodeBot supports up to **16** breakpoints at a time.

Debug: *with Breakpoints!*

Make sure your CodeBot is **connected** and **stopped**.

- Click to the left of the *line number* of the statement `power = [0, 0]` **in your `drive()` function.**
- Be sure you see the *red **dot*** marker (see picture above).

Now, click the the "Debug" button!

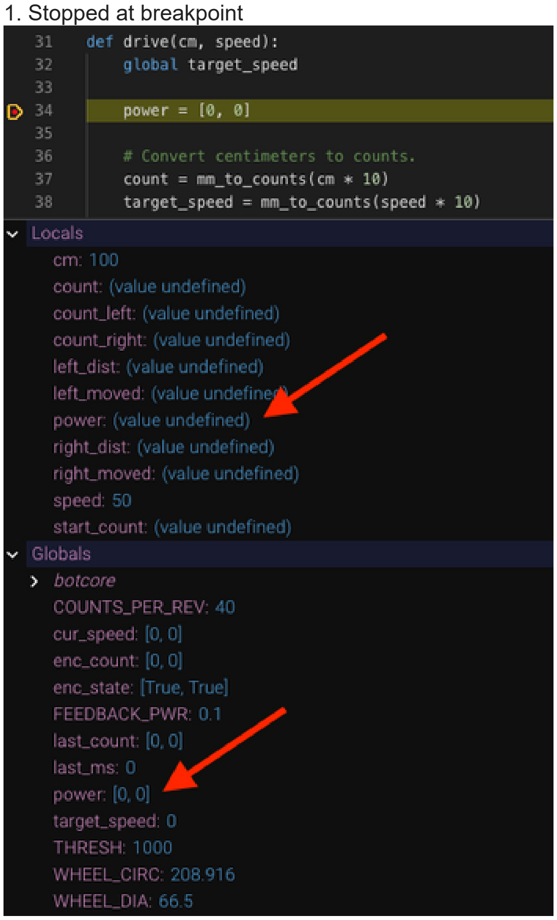
- When the yellow line appears, press the "Continue" button.
- On CodeBot, press **BTN-0** to start the first run.
- Your program should ***stop*** at the *breakpoint!*
  - This is the *first* run, so press to continue.
- After the first run, press **BTN-0** again.
- This time when your program ***stops*** at the *breakpoint*, start inspecting variables.

Stop, Step, and Inspect

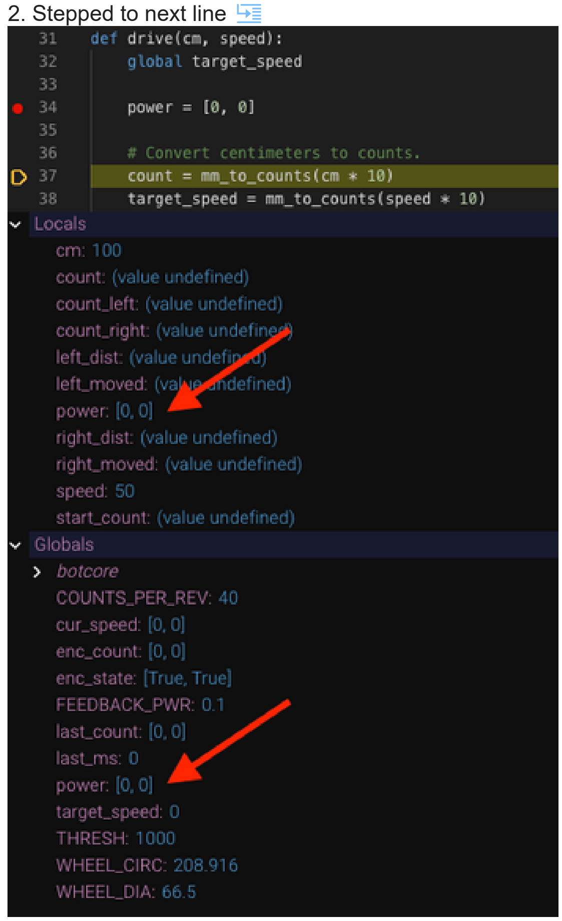
When your code hits the *breakpoint*, *open the* *console* to view your **Variables**.

- The pictures below show the ***variables at the breakpoint*** and after one *single step.*

1. Stopped at breakpoint



2. Stepped to next line



Ah! By mistake a *local* version of `power` was created!

So that explains why the global `power` was not *reset*.

- It was assigned inside a function which did **not** declare it as `global`.
- When you *assign* to a variable inside a function, Python assumes it is *local* unless you tell it otherwise.

Type in the Code

There's an *easy* fix for this problem: Add `power` to the `global` list at the top of `drive()`.

- Make the change and give it a try!

```
def drive(cm, speed):
    global target_speed, power
    ...
```

Run It!

Test that small fix, and make sure it works as planned.

Oh, and one more improvement!

Have you noticed that your 'bot sometimes *overshoots* the mark a bit when you're running at high speed on a measured path.

- You ***have*** been testing it on a *measured path*, haven't you??

If you want CodeBot to **"Stick the Landing"** then you'll need to *put on the **brakes!***

- Just *reverse the motors* for a short time, say **50ms**.

Check the 'Trek!

Find the code near the end of your `drive()` function that shuts down the motors.

- ***Before*** setting them to **0%**, add code to do a *quick reverse* of the motors.
- Use the current `power[]` settings, so the *braking* is proportional to your speed.
  - To *reverse* the current values, just *negate* them! *(flip the ± sign with −)*
- About **50 ms** of *braking* should be enough to fully stop the wheels.
- Use the `sleep_ms()` function from the time module to get a more precise delay than normal `sleep()`.

CodeTrek:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff, sleep_ms
```

Import `sleep_ms` from `time`!

- `sleep_ms` is the same as `sleep`, except it takes **milliseconds** as an argument *instead of* **seconds!**

```

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000
FEEDBACK_PWR = 0.1  # Impact of speed error on motor power.

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm, speed):
    global target_speed, # TODO: Add power as a global
```

Add the variable `power` as a global!

- `global target_speed, power`

```

    power = [0, 0]

    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
    target_speed = mm_to_counts(speed * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # [ Removed motors.run() "start moving" code ]

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
```

```
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Brake
    # TODO: reverse the motors to break
```

Run **both** 🔧motors in the reverse *briefly!*

- Each motor's current power is stored in the `power` `global`.

```
motors.run(LEFT, -power[LEFT])
motors.run(RIGHT, -power[RIGHT])
```

```
    sleep_ms(50)
```

Sleep for `50` **milliseconds!**

```

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
        control_speed(LEFT)
        control_speed(RIGHT)
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")

def control_speed(side):
    # Set motor power to reach target speed.
    err = (target_speed - cur_speed[side]) * FEEDBACK_PWR
    pwr = power[side] + err
```

```

    if pwr > 100:
        pwr = 100
    elif pwr < -100:
        pwr = -100

    power[side] = pwr
    motors.run(side, pwr)



# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
target_speed = 0   # Desired speed in "counts per second"
power = [0, 0]

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Drive (dist=cm, speed=cm/s)
    drive(100, 50)
```

Goals:

- **Debug** the code and use the continue button.
- **Add** `power` to the `global` list at the top of `drive()` *on the same line* as `target_speed`.
- Import `sleep_ms` from `time`.
- **Reverse** both motors by calling `motors.run` with the *inverse* of each `side` in `power`.

Tools Found: Variables, Print Function, Locals and Globals, Functions, Motors, Time Module, import

Solution:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff, sleep_ms

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True
```

```

    # No movement
    return False

def drive(cm, speed):
    global target_speed, power

    power = [0, 0]

    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
    target_speed = mm_to_counts(speed * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # [ Removed motors.run() "start moving" code ]

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Brake
    motors.run(LEFT, -power[LEFT])
    motors.run(RIGHT, -power[RIGHT])
    sleep_ms(50)

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
        control_speed(LEFT)
        control_speed(RIGHT)
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
```

```
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")


FEEDBACK_PWR = 0.1  # Impact of speed error on motor power.

def control_speed(side):
    # Set motor power to reach target speed.
    err = (target_speed - cur_speed[side]) * FEEDBACK_PWR
    pwr = power[side] + err

    if pwr > 100:
        pwr = 100
    elif pwr < -100:
        pwr = -100

    power[side] = pwr
    motors.run(side, pwr)



# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
target_speed = 0   # Desired speed in "counts per second"
power = [0, 0]

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Drive (dist=cm, speed=cm/s)
    drive(100, 50)
```

Objective 13 - Dead Reckoning!

You need just *one more* **Navigation** capability to *chart a course* with a distance and *direction* of your choosing!

Direction of Rotation: *Clockwise* or *Counter-clockwise?*

Making the wheels move a certain distance in a *straight line* is one thing.

- You might think **angular rotation** will be a lot tricker!
- Actually it's ***really easy!***
- Your `drive()` function is already doing most of the work.

How do you specify the **direction** of rotation?

- The table below shows **signs** you would use for **LEFT** and **RIGHT** motor power for *movement and rotation*.

Direction	LEFT	RIGHT
Forward	+	+
Backward	-	-
Rotate CW	+	-
Rotate CCW	-	+

Example:

```
# Rotate Clockwise
motors.run(LEFT, +50)
motors.run(RIGHT, -50)
```

Your `drive()` function only handles *forward* movement so far.

Your next step is to fix that!

Check the 'Trek!

Add the following to your code, for *selectable* **drive** *directions*.

- Use a *global variable* `direction[]` to hold the `LEFT` and `RIGHT` **signs** for *motor power*.
- In your `control_speed()` function, modify the *power* you set with `motors.run()`.
    - You'll need to *multiply* the `pwr` value by `+1` or `-1` to set the direction.
    - So `direction = [-1, -1]` will mean ***backward***.
- Set the global `direction` inside your `drive()` function.
    - When you call the `drive()` function, you can pass-in a new **direction**.
- The code below shows how you can define ***default parameter values*** in Python.
    - That means if you *don't* supply a new value for `dir` then the *default* value is used.
    - In this case, default to **forward** `[+1, +1]`

Run It!

Try driving **forward** and **backward**.

- Pretty simple change, right?
- Also try some **rotation!**

CodeTrek:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff, sleep_ms

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
TRACK_WIDTH = 114  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000
FEEDBACK_PWR = 0.1  # Impact of speed error on motor power.

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False


def drive(cm, speed, dir=[+1, +1]):
```

Update your `drive` function to accept `dir` as an argument.

- Setting the value of a parameter sets it's *default*.
- If you were to call `drive(30, 10)`, `dir` would be `[+1, +1]`.
- If you were to call `drive(30, 10, [-1, -1])`, `dir` would be `[-1, -1]`!

```
    global target_speed, power, direction
```

Add `direction` to your 🔧globals list.

```
    # Set the global direction.
    direction = dir
```

Set the *global* as the value `dir` supplied to `drive`.

- You'll be referencing it in *the next step!*

```


    # Reset the motor to zero power.
    power = [0, 0]

    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
    target_speed = mm_to_counts(speed * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Brake
    motors.run(LEFT, -power[LEFT])
    motors.run(RIGHT, -power[RIGHT])
    sleep_ms(50)

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
        control_speed(LEFT)
```

```
        control_speed(RIGHT)
        print("Power: ", power)
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")

def control_speed(side):
    # Set motor power to reach target speed.
    err = target_speed - cur_speed[side]
    pwr = power[side] + err * FEEDBACK_PWR

    if pwr > 100:
        pwr = 100
    elif pwr < -100:
        pwr = -100

    power[side] = pwr
    # TODO: Apply the direction to the line motors.run(side, pwr)
```

Update the `motors.run` call in your `control_speed()` function.

- *Multiply* `pwr` by `direction[side]` to *apply* the direction.
- `motors.run(side, pwr * direction[side])`

```

# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
target_speed = 0
power = [0, 0]

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Go forward - using default parameter [+1, +1]
    drive(30, 10)
```

Calling `drive()` *without* supplying a value to the parameter `dir` will use the parameter's default!

- In this case, it'll **default** to `[+1, +1]`.

```
    # Back up!
    drive(30, 10, [-1, -1])
```

Supplying a value to the parameter `dir` will cause the **default** to be *overridden!*

- *In this case*, `dir` will be `[-1, -1]`

Goals:

- **Define** a ***default parameter value*** in your `drive()` function for the *new* parameter `dir`.
- **Update** the `motors.run` call in your `control_speed()` function.
- Multiply `pwr` by `direction[side]`.

Tools Found: Motors, Locals and Globals, Functions, Keyword and Positional Arguments, Parameters, Arguments, and Returns

Solution:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff, sleep_ms

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
TRACK_WIDTH = 114  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False


def drive(cm, speed, dir=[+1, +1]):
    global target_speed, power, direction
    # Set the global direction.
    direction = dir


    # Reset the motor to zero power.
    power = [0, 0]

    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
    target_speed = mm_to_counts(speed * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Brake
    motors.run(LEFT, -power[LEFT])
```

```
    motors.run(RIGHT, -power[RIGHT])
    sleep_ms(50)

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
        control_speed(LEFT)
        control_speed(RIGHT)
        print("Power: ", power)
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")

FEEDBACK_PWR = 0.1  # Impact of speed error on motor power.

def control_speed(side):
    # Set motor power to reach target speed.
    err = target_speed - cur_speed[side]
    pwr = power[side] + err * FEEDBACK_PWR

    if pwr > 100:
        pwr = 100
    elif pwr < -100:
        pwr = -100

    power[side] = pwr
    motors.run(side, pwr * direction[side])


# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
target_speed = 0
power = [0, 0]

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

```

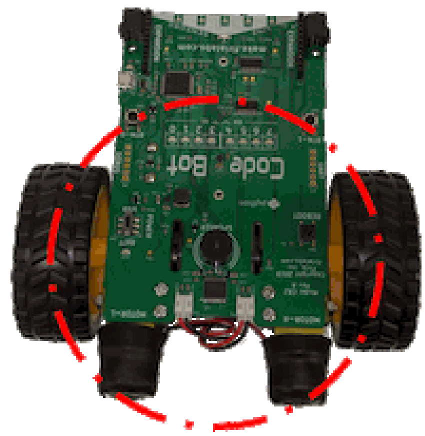
```
    # Go forward - using default parameter [+1, +1]
    drive(30, 10)
    # Back up!
    drive(30, 10, [-1, -1])
```

Objective 14 **- Dead Reckoning 2**

Rotation by a Specified Angle

When your 'bot rotates in place, the wheels trace a *circular path*.

- The **diameter** of the circle shown at right is called the **Wheel Track** width.
- So if the 'bot rotates through a **full 360°** circle,
  - ...the *wheels* travel its full **circumference!**



$$circumference = \pi \cdot track\ width$$

Ex: To rotate 180° *each wheel* would need to travel:

$$distance = circumference \cdot (\frac{180}{360})$$

For other *angles* substitute desired **angle** for **180°** in the above formula!

Now to add a 🔧function that makes **rotation** easy!

Change Your *Heading*

In *navigation*, your **heading** is the *direction* you are facing.

- If you turn **90°** to the right (clockwise), you've changed your *heading* by **+90°**.
- If you turn to the left (counter-clockwise) that's a ***negative*** *angle heading* change.

Check the 'Trek!

Define a new function `def rotate(angle, speed):`.

- See above for the formula to calculate **distance** based on **angle**.
  - Measure your **track width** and define a 🔧constant for it in *mm*. On my 'bot I measured `TRACK_WIDTH = 114`.
- A positive `angle` should rotate *clockwise*, and negative should rotate *counter-clockwise* like a navigational **heading**.

Run It!

Take your new **rotation** code for a *spin!*

- Test out different *speeds* and *angles*.
- The accuracy won't be perfect, but *it sure beats guessing!*

A New Level of Control

Your `drive()` and `rotate()` functions give you a much better way to provide exact movement instructions to CodeBot.

- Remember your *First Navigation Challenge* to **"drive in a square?"**
  - Try it now with your new code!

CodeTrek:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff, sleep_ms

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
TRACK_WIDTH = 114  # mm
```

Don't forget to set `TRACK_WIDTH` as a 🔧constant!

- Mine was *114!*

```
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000
FEEDBACK_PWR = 0.1  # Impact of speed error on motor power.

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm, speed, dir=[1, 1]):
    global target_speed, power, direction
    print("target dist=", cm, "cm")
    direction = dir

    # Reset the motor to zero power
    power = [0, 0]

    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
    target_speed = mm_to_counts(speed * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Brake
    motors.run(LEFT, -power[LEFT])
    motors.run(RIGHT, -power[RIGHT])
    sleep_ms(50)

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

```

```python
def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
        control_speed(LEFT)
        control_speed(RIGHT)
        print("Power: ", power)
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")

def control_speed(side):
    # Set motor power to reach target speed.
    err = target_speed - cur_speed[side]
    pwr = power[side] + err * FEEDBACK_PWR

    if pwr > 100:
        pwr = 100
    elif pwr < -100:
        pwr = -100

    power[side] = pwr
    motors.run(side, pwr * direction[side])


def rotate(angle, speed):
```

Define your *new* `rotate(angle, speed)` function.

- `angle` represents your **heading**.

```python
    # Determine direction of L,R wheels.
    if angle < 0:
        dir = [-1, +1]  # CCW heading
    else:
        dir = [+1, -1]  # CW heading
```

Determine motor direction based on the `angle`'s **sign**.

```python

    # Full 360 degree rotation in mm
    # is pi * diameter.
    circumference = math.pi * TRACK_WIDTH
```

Calculate the ***full*** `360` degree rotation of your 'bot in millimeters.

- `TRACK_WIDTH` is the **distance** between the *wheels!*

```python
    dist_mm = # TODO: Calculate the distance in mm!
```

Calculate the **distance** in **millimeters** to `drive()`!

$$distance = circumference \cdot (\frac{angle}{360})$$

- `dist_mm = abs(circumference * angle / 360 )`

```
    cm = dist_mm / 10
    drive(cm, speed, dir)
```

Put it all together and ***drive!***

```



# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
target_speed = 0
power = [0, 0]

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Turn to the right
    rotate(90, 10)
```

Start with `90` and `10`, *then* test out different ***speeds*** and ***angles!***

Goals:

- **Define** a *new* function `rotate(angle, speed)`.
- **Assign** the *wheel track width* of **your** 'bot to the constant variable `TRACK_WIDTH`.
- **Assign** the **distance** to drive in **mm** to the variable `dist_mm`.

Tools Found: Functions, Constants, Motors

Solution:

```
from botcore import *
import math
from time import ticks_ms, ticks_diff, sleep_ms

COUNTS_PER_REV = 40
WHEEL_DIA = 66.5  # mm
TRACK_WIDTH = 114  # mm
WHEEL_CIRC = (math.pi * WHEEL_DIA)
THRESH = 1000

def counts_to_mm(count):
    return count * WHEEL_CIRC / COUNTS_PER_REV

def mm_to_counts(mm):
    return mm * COUNTS_PER_REV / WHEEL_CIRC

def sense_slot(side):
    val = enc.read(side)
    return val > THRESH

def check_enc(side):
```

```
    slot = sense_slot(side)
    if enc_state[side] != slot:
        # Disc has moved!
        enc_state[side] = slot
        enc_count[side] = enc_count[side] + 1
        return True

    # No movement
    return False

def drive(cm, speed, dir=[1, 1]):
    global target_speed, power, direction
    print("target dist=", cm, "cm")
    direction = dir

    # Reset the motor to zero power
    power = [0, 0]

    # Convert centimeters to counts.
    count = mm_to_counts(cm * 10)
    target_speed = mm_to_counts(speed * 10)

    # Save the starting line.
    start_count = enc_count.copy()

    # Keep going until 'count' reached
    while True:
        left_moved = check_enc(LEFT)
        right_moved = check_enc(RIGHT)

        # Update speed every 100ms
        update_speed(100)

        if left_moved or right_moved:
            # print(enc_count)

            # Calculate distance from starting line
            count_left = enc_count[LEFT] - start_count[LEFT]
            count_right = enc_count[RIGHT] - start_count[RIGHT]

            # Are we there yet??
            if count_left >= count or count_right >= count:
                break

    # Brake
    motors.run(LEFT, -power[LEFT])
    motors.run(RIGHT, -power[RIGHT])
    sleep_ms(50)

    # Stop moving
    motors.run(LEFT, 0)
    motors.run(RIGHT, 0)

    # Print total distance traveled
    left_dist = counts_to_mm(count_left)
    right_dist = counts_to_mm(count_right)
    print("Left Distance: ", left_dist, "mm")
    print("Right Distance: ", right_dist, "mm")

def update_speed(interval_ms):
    # Update speed at given interval.
    global last_ms, last_count

    # Check if interval has elapsed.
    t_ms = ticks_diff(ticks_ms(), last_ms)
    if t_ms >= interval_ms:
        # Calculate distance traveled.
        d_left = enc_count[LEFT] - last_count[LEFT]
        d_right = enc_count[RIGHT] - last_count[RIGHT]
        # Save state for next time.
        last_ms = ticks_ms()
        last_count = enc_count.copy()
        # Calculate speed
        t_sec = t_ms / 1000  # convert to seconds
        cur_speed[LEFT] = d_left / t_sec
        cur_speed[RIGHT] = d_right / t_sec
```

```
        control_speed(LEFT)
        control_speed(RIGHT)
        print("Power: ", power)
        print_speed_cps()

def print_speed_cps():
    # Print current speed in cm per second.
    cps_left = counts_to_mm(cur_speed[LEFT]) / 10
    cps_right = counts_to_mm(cur_speed[RIGHT]) / 10
    print("Left: ", cps_left, "cm/s")
    print("Right: ", cps_right, "cm/s")

FEEDBACK_PWR = 0.1  # Impact of speed error on motor power.

def control_speed(side):
    # Set motor power to reach target speed.
    err = target_speed - cur_speed[side]
    pwr = power[side] + err * FEEDBACK_PWR

    if pwr > 100:
        pwr = 100
    elif pwr < -100:
        pwr = -100

    power[side] = pwr
    motors.run(side, pwr * direction[side])


def rotate(angle, speed):
    # Determine direction of L,R wheels.
    if angle < 0:
        dir = [-1, +1]  # CCW heading
    else:
        dir = [+1, -1]  # CW heading

    # Full 360 degree rotation in mm
    # is pi * diameter.
    circumference = math.pi * TRACK_WIDTH
    dist_mm = abs(circumference * angle / 360)
    cm = dist_mm / 10
    drive(cm, speed, dir)



# --- Main program ---
enc_state = [sense_slot(LEFT), sense_slot(RIGHT)]
enc_count = [0, 0]
last_ms = 0
last_count = [0, 0]
cur_speed = [0, 0]   # Current speed in "counts per second"
target_speed = 0
power = [0, 0]

while True:
    # Wait for BTN-0. Good robot.
    buttons.was_pressed(0)  # debounce
    while True:
        if buttons.was_pressed(0):
            break

    motors.enable(True)

    # Turn to the right
    rotate(90, 10)
```

Mission 8 Complete

This was a *challenging journey!*

The 🔧wheel encoders themselves weren't too difficult to understand, but the **code** you crafted to harness their true power was quite an *adventure.*

- Exploring the principles of ***rotary encoders.***
- Creating a ***measuring wheel*** that provides *true distance*.
- Building a ***speedometer*** and making your 'bot drive an exact *distance* and *speed* along a path.

- Mastering ***angular rotation*** for true *dead reckoning* capabilities!

And this *code* is not just for *Robots*...

- The ***Process Control Loop*** you implemented is *crucial* to modern appliances, vehicles, heating and air-conditioning, and many other technologies you rely on daily.
- From *factories* to *farms*, *phones* to *drones*, the ***computer science principles*** in this project are used by professional *embedded systems programmers* to craft the core capabilities that move the modern world.

Try Your Skills

Suggested Re-mix Ideas:

- **Experiment** with the 🔧constants your code depends on:
  - The 100ms *update interval*.
  - The `FEEDBACK_PWR` factor.
  - ***Keep notes*** on the effects as you **adjust** these 🔧constants.
  - Does the **"best"** setting depend on a particular journey's requirements?
- **Straighten Up** your path!
  - Your code uses `current_speed` in the *feedback* loop, but what about ***current distance?***
  - Add code to factor-in the **total distance** traveled in a run, to ensure *both wheels* go the same *distance*.
  - You're well on your way to *advanced* PID Control!

Mission 9 - All Systems Go!

In this project you'll get to know *CodeBot's* internal system sensors.

- Your 'bot can measure its own ***battery voltage*** and ***CPU temperature***.
- And it can sense its ***orientation*** as well as ***impacts*** and ***vibration*** with the CodeBot accelerometer.

Ah yes, CodeBot is *self-aware!*

Project Goals:

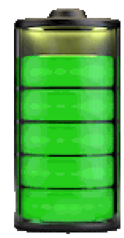
- Code a **battery tester** so you can tell how much fuel is left in your bot's tank.
- Use the *CPU temperature* to detect changes in the local "weather".
- Detect *orientation* with the CodeBot accelerometer and **rotate** toward the sky!
- Make a **motion alarm** "guard robot".

Robot, Know Thyself!

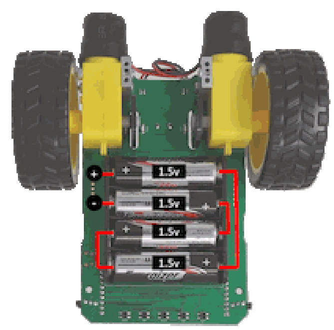
Objective 1 - Battery Check

When it comes to monitoring the state of your hardware, ***battery level*** is one of the most critical items to track.

- CodeBot's system sensors give your 'bot the tools to monitor its battery voltage.
- Understanding how the those sensors work is the first step to figuring out how much *"fuel"* is left in your bot's tank!



Typical *AA alkaline batteries* provide 1.5 Volts per cell when fresh.



- *CodeBot* has **4** batteries in *series*, wired as shown. Voltages in *series* **add**, so if you're using *Alkalines* the total voltage of a brand-new set of batteries would be:

$$V_{batt} = 1.5v \cdot 4 = 6.0v$$

As batteries use up their capacity, their voltage *decreases.*

- Due to battery chemistry, you'll see the full drop in voltage only when the batteries are under *load*, like when they're lighting up some LEDs or running the motors.

Concept: *System Power Monitoring*

The **botcore** library ***system*** object has two functions that allow CodeBot to check the status of its power supply:

```
# Measure power supply voltage (battery or USB)
v = system.pwr_volts()

# Am I powered by USB or Battery? (based on Power switch)
on_usb = system.pwr_is_usb()
```

The first function `pwr_volts()` returns the float power supply voltage, which might be coming from USB or from the onboard battery pack, depending on the position of CodeBot's ***power switch.***

- Note that even when you're connected to the USB port you can still set the switch to the BATT position so the 'bot draws its power from the onboard battery pack rather than USB.
- Computers and USB chargers typically supply **about** 5V (4.40V - 5.25V for USB 2.0). With the switch in the USB position you can confirm that!

The second function `pwr_is_usb()` returns int value `1` if the switch is currently in the *USB* position, and `0` if you're running on *batteries*.

Try it on the REPL

Open the **Console** and test **both** of these functions from the system sensors API.

- If code is *already running* you will need to press the ☐ **Stop** button first!
- To get started, type `from botcore import *` on the REPL.
- Now check to see if you're running from USB power:
  - Type `system.pwr_is_usb()`
  - The return value will be `1` if the power switch is set to USB, and `0` if not.

When you call `system.pwr_volts()` it returns the power supply voltage.

- Try it a few times. Remember, you can press *up-arrow* ↑ `ENTER` to repeat prior commands.
- The battery voltage should return a pretty consistent value, but if you *load* it down it will ***decrease***.
  - Turn on some LEDs to put a *load* on the batteries: **Enter** `leds.user(127)`
  - Now repeat the `system.pwr_volts()` command. *See a **lower** voltage?*
  - Turn the LEDs off with `leds.user(0)`, and try the voltage reading again. *Did the voltage **increase**?*

Now that you have a feel for how the **power monitor** features of the system sensors work, it's time to *write some code* to put that knowledge to good use!

Hint:

- *Having trouble in the REPL?*
- If code is *already running* you will need to press the ☐ **Stop** button first.
- *Before calling* either function, call `from botcore import *`.

Goals:

- Call `system.pwr_is_usb()` in the **REPL.**
- Call `system.pwr_volts()` in the **REPL.**

Tools Found: System Status Monitors, LED, Motors, import, Functions, float, int, Print Function, Parameters, Arguments, and Returns

Solution:

N/A

***Objective 2* - Battery Tester**

Create a New File!

Run It!

Take a look at the **console** when you run this code.

- How's ***your*** battery doing?
- Check out your test values:
  - Do the results for each *test voltage* match the table above?
  - Maybe you noticed *some* of the percentages are **not** very accurate!?
- You could adjust the `if` statements to improve it, or even add more *increments*.
  - But this *"table-driven"* approach is pretty limited.
  - There *must* be a better way...

CodeTrek:

```
from botcore import *

def vbatt_load():
```

Your *new* function `vbatt_load()` returns the power supply's voltage *under load.*

```
    # Read batt voltage under load
    leds.user(0b00001111)
```

Turn *on* 4 LEDs.

- This will *draw current* from your 'bot's power supply!

```
    v = # TODO: Get the power supply's voltage
```

Measure your *'bot's* power supply voltage!

- Use the `system` function, `pwr_volts()`!
- `v = system.pwr_volts()`

```
    leds.user(0)

    return v
```

`return` the measured voltage.

- *Don't forget* to turn **off** those LEDs!

```

def batt_table(v):
```

`batt_table(v)` takes a `voltage` reading and returns a float between `0.0` and `1.0`.

- This *corresponds* to capacity levesl from 0% to 100%.

```
    if v > 5.5:
        pct = 1.0
    elif v > 5.0:
        pct = 0.75
    elif v > 4.5:
        pct = 0.50
    elif v > 4.0:
        pct = 0.25
    else:
        pct = 0.0
```

The *table in the instructions* was used to create this `if` statement chain.

- It *associates* a **voltage** with a percentage **capacity!**

```

    print("batt_table: ", v, "->", pct)
    return pct
```

Print the change from the *input* (`v`) to the *output* (`pct`) and `return` the *output!*

```


# Main program
# Check my battery
vb = vbatt_load()
my_capacity = batt_table(vb)
```

Check out the current *capacity* of your *battery!*

```
print("My battery capacity: ", my_capacity)

# Try some test values
batt_table(3.9)
batt_table(4.2)
batt_table(4.8)
batt_table(5.2)
batt_table(5.8)
batt_table(6.2)
```

Test a *range* of **voltage** values to ensure your function is *functioning* correctly!

Goals:

- **Define** a functon named `vbatt_load()`.
- **Assign** the power supply's voltage to the variable `v`.
- **Define** a function named `batt_table(v)`.
- `return` a variable `pct` that represents the parameter `v` as a float between `0.0` and `1.0`.
- **Assign** `vb = batt_load()` and use `vb` as the argument for `batt_table(vb)`.

Tools Found: Parameters, Arguments, and Returns, CodeBot LEDs, System Status Monitors, float, Print Function, Variables, Functions, LED

Solution:

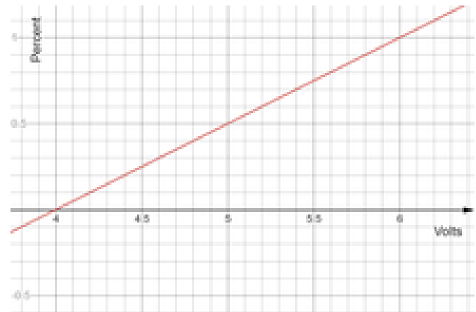
```
from botcore import *

def vbatt_load():
    # Read batt voltage under load
    leds.user(0b00001111)
    v = system.pwr_volts()
    leds.user(0)

    return v

def batt_table(v):
    if v > 5.5:
        pct = 1.0
    elif v > 5.0:
        pct = 0.75
    elif v > 4.5:
        pct = 0.50
    elif v > 4.0:
        pct = 0.25
    else:
        pct = 0.0

    print("batt_table: ", v, "->", pct)
    return pct


# Main program
# Check my battery
vb = vbatt_load()
my_capacity = batt_table(vb)
print("My battery capacity: ", my_capacity)

# Try some test values
batt_table(3.9)
batt_table(4.2)
batt_table(4.8)
batt_table(5.2)
batt_table(5.8)
```

```
batt_table(6.2)

```

Objective 3 - Equations To The Rescue!

Rather than using a **table**, maybe a little *math* can help here!

- You're approximating the battery voltage decay as a *straight line*.
- You may have seen the *equation* for a line: $y = mx + b$
- Plotting **Percent** on the Y-axis and **Volts** on the X-axis, the equation of this line is: $y = \frac{1}{2} \cdot x - 2$



Check the 'Trek!

Modify your code to use the *equation* above to calculate *battery capacity*.

- Add a new function `def batt_level(v):` that uses the equation: `percent = (volts / 2) - 2` to calculate and return the capacity between `0.0` and `1.0`
- Change your test code to use this new function rather than `batt_table(v)`

Run It!

Give the new version a try!

- Watch the **console** for the printout of test values.
- *Cool!* More accuracy *and* the `batt_level(v)` function actually has fewer lines of code!

CodeTrek:

```
from botcore import *

def vbatt_load():
    # Read batt voltage under load
    leds.user(0b00001111)
    v = system.pwr_volts()
    leds.user(0)

    return v


def batt_level(v):
```

`batt_level(v)` *replaces* `batt_table(v)` from the previous objective.

- It still `return`s a float value between `0.0` and `1.0`.
- Instead of relying on a **table**, it uses an **equation.**

```
    pct = # TODO: calculate the battery's capacity
```

Calculate the capacity float using the equation:

$$y = \frac{1}{2} \cdot x - 2$$

Where y is `pct` and x is `v`!

- `pct = (v / 2) - 2`

```

    if pct > 1:
```

```
        pct = 1
    elif pct < 0:
        pct = 0
```

Round off `pct` in case the **equation** `return`s a value *outside* of the range `0.0` to `1.0`.

```

    print("batt_level: ", v, "->", pct)
    return pct
```

Print the *translation* from `v` to `pct` and `return` `pct`!

```


# Main program
# Check my battery
vb = vbatt_load()
my_capacity = batt_level(vb)
```

Replace the `batt_table(vb)` call from *last objective* with `batt_level(vb)`.

```
print("My battery capacity: ", my_capacity)

# Try some test values
batt_level(3.9)
batt_level(4.2)
batt_level(4.8)
batt_level(5.2)
batt_level(5.8)
batt_level(6.2)
```

Test out some **voltage** values with your *new* `batt_level(v)` function!

Goals:

- **Define** the function `batt_level(v)`.
- **Assign** the variable `pct = (v / 2) - 2`
- Call `batt_level(vb)`.

Tools Found: Print Function, Functions, Variables, float

Solution:

```
from botcore import *

def vbatt_load():
    # Read batt voltage under load
    leds.user(0b00001111)
    v = system.pwr_volts()
    leds.user(0)

    return v


def batt_level(v):
    pct = (v / 2) - 2

    if pct > 1:
        pct = 1
    elif pct < 0:
        pct = 0

```

```
    print("batt_level: ", v, "->", pct)
    return pct


# Main program
# Check my battery
vb = vbatt_load()
my_capacity = batt_level(vb)
print("My battery capacity: ", my_capacity)

# Try some test values
batt_level(3.9)
batt_level(4.2)
batt_level(4.8)
batt_level(5.2)
batt_level(5.8)
batt_level(6.2)
```

Objective 4 **- Battery Indicator Light**

To polish off your battery tester you need a User Interface that doesn't require watching the *console!*

This step adds a useful function you could call when a program starts, so there would be an indication to the user of *battery health*. If the **Battery LED** stays lit, it's a warning to change batteries soon! ***One*** **blink** means the batteries are *full*. ***Two*** **blinks** means they're *used* but still healthy.

Check the 'Trek!

Define a new function called `def check_batt(pct):` that uses the percent value from `batt_level()` to activate a special ***battery LED***.

- This red LED is just above the *power switch*, controlled with `leds.pwr(is_on)`
- Blink the LED **once** if capacity is over 60%.
- Blink it **twice** if capacity is between 20% and 60%.
- Leave the LED lit continuously if capacity is under 20%.

Don't forget to use some test functions to be sure your `check_batt()` function works as expected for different capacity levels. Take a look at the example: remove the `#` comment from one test function at a time and run the program to check each level.

Run It!

Test this code, and make sure your **Battery Indicator** works as expected.

- This would be a good addition to the startup code for any program!

CodeTrek:

```
from botcore import *
from time import sleep
```

Don't forget to `import sleep`!

```

def vbatt_load():
    # Read batt voltage under load
    leds.user(0b00001111)
    v = system.pwr_volts()
    leds.user(0)

```

```
10     return v
11
12 def batt_level(v):
13     pct = (v / 2) - 2
14
15     if pct > 1:
16         pct = 1
17     elif pct < 0:
18         pct = 0
19
20     print("batt_level: ", v, "->", pct)
21     return pct
22
23
24 def check_batt(pct):
```

`check_batt(pct)` uses the float "percent value" returned from `batt_level()` to indicate **battery status** *through* the LEDs.

```
25     if pct < 0.2:
26         leds.pwr(True)
27         # TODO: return so the code below doesn't run.
```

If **battery capacity** is *lower* than `0.2`, light the LEDs and *leave them on!*

- Call `return`, that way the *remaining* code in the `check_batt(pct)` function will **only** run if battery capacity is *above* `0.2`.

```
28
29     if pct > 0.6:
30         blinks = 1
31     else:
32         blinks = 2
```

Use the variable `blinks` to store how many times the LEDs should ***blink*** depending on **battery capactiy.**

- **Above** `0.6`, `1` *blink.*
- **Below** `0.6`, `2` *blinks!*

```
33
34     while blinks > 0:
```

Iterate through `blinks`.

- You'll **decrement** `blinks` on line *40* so that it *eventually* reaches `0`!

```
35         leds.pwr(True)
36         sleep(0.5)
37         leds.pwr(False)
38         sleep(0.2)
```

Blink!

- Turn the LED on *briefly,* then turn it off!
- This *might* be run **twice** in a row, so make sure to add a small `sleep` *after* your LED is turned off.

```
39
40         # TODO: Decrement blinks
```

Decrement `blinks` each cycle to ensure it `breaks` out of this `while` loop!

- `blinks = blinks - 1`

```
41
```

```
# Main program
vb = vbatt_load()
my_capacity = batt_level(vb)
print("My battery capacity: ", my_capacity)
check_batt(my_capacity)
```

*Give your **battery indicator** a try!*

```


# Test Code:
#check_batt(0.1)
#check_batt(0.5)
#check_batt(0.8)
```

Goals:

- **Define** a new function called `check_batt(pct)`.
- If **battery capacity** is *under* `0.2`, turn **ON** the 🔧LEDs and `return`.
- **Decrement** `blinks` in your `while` 🔧loop.
- Call `check_batt(my_capacity)`

Tools Found: UI, Functions, LED, Comments, Loops, float, Variables, Iterable

Solution:

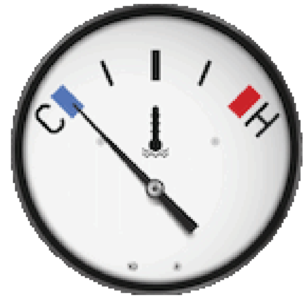
```
from botcore import *
from time import sleep

def vbatt_load():
    # Read batt voltage under load
    leds.user(0b00001111)
    v = system.pwr_volts()
    leds.user(0)

    return v

def batt_level(v):
    pct = (v / 2) - 2

    if pct > 1:
        pct = 1
    elif pct < 0:
        pct = 0

    print("batt_level: ", v, "->", pct)
    return pct


def check_batt(pct):
    if pct < 0.2:
        leds.pwr(True)
        return

    if pct > 0.6:
        blinks = 1
    else:
        blinks = 2

    while blinks > 0:
        leds.pwr(True)
        sleep(0.5)
        leds.pwr(False)
        sleep(0.2)

        blinks = blinks - 1

# Main program
```

```
vb = vbatt_load()
my_capacity = batt_level(vb)
print("My battery capacity: ", my_capacity)
check_batt(my_capacity)


# Test Code:
#check_batt(0.1)
#check_batt(0.5)
#check_batt(0.8)
```

Objective 5 - Sensing Temperature

Your bot's CPU has an *internal* **temperature sensor**.

- The system sensors API lets you read the temperature in *Celsius* or *Fahrenheit* using `system.temp_C()` and `system.temp_F()` respectively.



The CPU's reported temperature is influenced by:

- The *external* temperature of the *air* surrounding the 'bot, or in contact with the CPU.
- The level of activity in the processor.

NOTE: The CodeBot CB3 will generally report about 10°C warmer than the CB2 due to:

- Differences in the built-in temperature sensors.
- The metal shield covering the CB3's sensor and electronic parts.

Create a New File!

Use the **File** → **New File** menu to create a new file called ***TemperatureCheck***.

Check the 'Trek!

- Write a `while True:` loop that:
  1. Reads the temperature in °C. (use the `system.temp_C()` function)
  2. Prints the temperature to the *console*.
  3. Sleeps 200ms (use the `sleep_ms()` function from the time module)

Run It!

Watch the output on the *console* as your program runs.

- What's the normal *"ambient"* temperature?

If you have a **CodeBot CB2** you can gently touch the **CPU** with your finger to observe a change in *temperature*.

- Remove your finger and watch the temperature decrease slowly.
- Can you make it cool faster by blowing cool air across it?

CodeTrek:

```
from botcore import *
from time import sleep_ms
```

Don't forget to `import` `sleep_ms` from the `time` **module!**

```

while True:
    # TODO: code this part
```

This part is *up to you!*

1. **Read** the temperature.
2. **Print** the temperature.
3. **Sleep** for 200ms.

If you get stuck, check the *instructions!*

Goals:

- Read the tempterature in **°C** using `system.temp_C()`.
- Print the *temperature*.
- *Sleep* for `200`**ms** using `sleep_ms()`.

Tools Found: CPU and Peripherals, System Status Monitors, Loops, Print Function, Time Module

Solution:

```
from botcore import *
from time import sleep_ms

while True:
    # TODO: code this part
    t = system.temp_C()
    print(t)
    sleep_ms(200)
```

Objective 6 - Smoothing The Data

You might notice that the "raw" temperature readings jump around a bit. Some of these changes aren't due to variations in temperature, but instead to the accuracy limitations of the sensor itself. The data is *noisy!*

- Your next step will be to *smooth* the data out with a ***moving average*** algorithm.

Check the 'Trek!

Above your `while True:` loop, add a global variable `samples = []` initialized to an empty list.

- Inside your loop when you read the temperature, append it to the `samples` list.
  - `samples.append(temperature)`
- After you've collected **5** samples, average them and `print()` the result!
  - Define a function `def avg_list(nlist):` that takes a list of numbers and returns the **average**.
    - *"Average"* (also called *"mean"*) is the sum of the numbers divided by the count.
  - You can experiment with different *"smoothing widths"*, but **5** is a good start.
  - Empty the list with `samples.clear()` after you average it.

Run It!

How's the temperature feeling to ya?

- You will still see variations, but they should be smaller now.
- Averaging is a good way to smooth out noisy data!
  - This technique is useful for *lots* of things - not just temperature :-)

CodeTrek:

```
from botcore import *
from time import sleep_ms
```

```python

def avg_list(nlist):
```

avg_list(nlist) takes a list of numbers *(in your case, temperature readings)*, and returns the **average**.

```python
    count = len(nlist)
```

count represents the *length* of the list nlist.

- The len(list) function returns the length of the supplied list.
- If you've *followed* the code **exactly,** count will be 5!

```python
    sum = 0
```

sum represents the **total value** of all the numbers in nlist *added together!*

```python
    i = 0

    # Sum up all the numbers in nlist.
    while i < count:
```

Loop through each **index** in nlist.

- You'll be **iterating** i at the *end* of the **loop!**

```python
        sum = sum + nlist[i]
        i = i + 1
```

i represents the **index** or **position** in the list.

- Add the number at the *current **index*** to sum!
- **Iterate** i *after!*

```python

    # TODO: Calculate and return the average
```

The **average** is the **sum** *divided by* the **count**.

- return (sum / count)!

```python

samples = []
```

The global variable samples is used to *store* **temperature** readings.

```python

while True:
    t = system.temp_C()
    samples.append(t)
```

When the **temperature** is read, *add it* to your samples global.

```python
    if len(samples) == 5:
        average = avg_list(samples)
```

When samples has 5 **temperature** readings, use the list as an argument to your *new* function, avg_list(nlist)!

```python
        samples.clear()
        print("Average temp=", average)
```

```
#@10

    sleep_ms(200)
#@11
```

Goals:

- **Define** a 🔧function `avg_list(nlist)`.
- `return` the `sum` *divided* by the `count` from `avg_list(nlist)`.
- Empty the 🔧list `samples` using `samples.clear()`.

Tools Found: Locals and Globals, list, Functions, Parameters, Arguments, and Returns, Variables, Keyword and Positional Arguments, Loops, Print Function

Solution:

```
from botcore import *
from time import sleep_ms

def avg_list(nlist):
    count = len(nlist)
    sum = 0
    i = 0
    # Sum up all the numbers in nlist.
    while i < count:
        # Add this list item to 'sum', and store total in 'sum'.
        sum = sum + nlist[i]
        i = i + 1

    # Calculate and return the average
    return (sum / count)

samples = []

while True:
    t = system.temp_C()
    samples.append(t)
    if len(samples) == 5:
        average = avg_list(samples)
        samples.clear()
        print("Average temp=", average)

    sleep_ms(200)
```

Objective 7 - Temperature-Controlled Lights!

If you have a **CodeBot CB3** the code below will be difficult to **test** without applying a big change to the ambient temperature at the "can" of the 🔧CPU module.

- That can be achieved with a hair-dryer pointed at the module... but be careful not to heat it too hot to touch!
- Don't worry if you can't heat/cool it though. Try the code below to see how a temperature-controlled system works.

If you have a **CodeBot CB2** you'll have an easier time *"moving the needle"* by using your finger to heat the 🔧CPU.

First make note of the average **ambient temperature** your 'bot is reading.

This can be used as a ***"baseline"*** temperature.

- Your challenge is to make *CodeBot* respond with 🔧LEDs based on *temperature:*
  - **No LEDs** should be lit if the temperature is near the baseline.
    - You need a ***"deadband"*** of about 3°C around the baseline.
    - *Seriously!* In a *control system* that's what you call the range or *band* of *input values* where the *output* doesn't change.
  - Light the **red** 🔧User LEDs when the temperature rises *above* the **baseline** *plus deadband*.
  - Light the **green LS LEDs** when the temperature falls *below* the **baseline** *minus deadband*.

Check the 'Trek!

Define a new function called `def check_baseline(t)` which compares a temperature against the *baseline* value, and controls the LEDs per the above *algorithm*.

- Call this new function from your main loop, after each time you `print()` the temperature to the *console*.

Run It!

You may need to adjust your `BASELINE` a little higher based on where you want it.

- Try adjusting the `DEADBAND` constant also!

Hey, You've Made a *Thermostat!*

Think about it -

- If the temperature is too low, turn on the *heating element*;
- If the temperature is too high, turn on the *cooling fan!*
- Keep a *deadband* around the target temperature so you don't *constantly cycle* heater/fan, wearing the equipment out.

CodeTrek:

```
from botcore import *
from time import sleep_ms


BASELINE = # Your measured temperature value here!
```

Set the `BASELINE` variable to your **ambient temperature**.

- ***Confused?*** Just use the **average** temperature from *last objective!*

```
DEADBAND = 3.0
```

The `DEADBAND` represents an acceptable range of **temperature variability.**

```

def check_baseline(t):
```

`check_baseline(t)` takes a **temperature** reading `(t)` and controls the LEDs based on it's relationship to the `BASELINE`.

```
    # TODO: Turn Off LEDs
```

Start by *clearing* the LED state by turning **off** the LEDs. You'll be using *both* **ls** and **user** LEDs.

```
leds.ls(0b00000)
leds.user(0b00000000)
```

```

    # Check if t is more than DEADBAND away from BASELINE.
    if t > BASELINE + DEADBAND:
        # TODO: Light red User LEDs
```

When the **temperature** `t` rises *above* `BASELINE + DEADBAND`, turn on the ***red*** User LEDs.

- `leds.user(0b11111111)`

```
    # TODO: elif t < ???
```

When the **temperature** `t` falls *below* `BASELINE - DEADBAND`, turn on the ***green* LS LEDs.**

- `elif t < BASELINE - DEADBAND:`

```
        leds.ls(0b11111)


def avg_list(nlist):
    count = len(nlist)
    sum = 0
    i = 0
    # Sum up all the numbers in nlist.
    while i < count:
        # Add each item from list to sum
        sum = sum + nlist[i]
        i = i + 1

    return (sum / count)

samples = []

while True:
    t = system.temp_C()
    samples.append(t)
    if len(samples) == 5:
        average = avg_list(samples)
        samples.clear()
        print("Average temp=", average)
        check_baseline(average)
```

Call your *new* 🔧function *after* 🔧printing the **average temperature.**

```

    sleep_ms(200)
```

Goals:

- **Initialize** a `BASELINE` 🔧constant to your **ambient temperature.**
- **Define** a new function called `check_baseline(t)`.
- `if t > BASELINE + DEADBAND:`, turn **on** *all* 🔧User LEDs.

Tools Found: CPU and Peripherals, LED, CodeBot LEDs, Constants, Variables, Functions, Print Function

Solution:

```
from botcore import *
from time import sleep_ms

BASELINE = 38
DEADBAND = 3.0

def check_baseline(t):
    # Turn Off LEDs
    # TODO: code this part
    leds.ls(0b00000)
    leds.user(0b00000000)

    # Check if t is more than DEADBAND away from BASELINE.
    # TODO: fill-in code below
    if t > BASELINE + DEADBAND:
        # Light red User LEDs
        leds.user(0b11111111)
    elif t < BASELINE - DEADBAND:
        leds.ls(0b11111)


```

```python
def avg_list(nlist):
    count = len(nlist)
    sum = 0
    i = 0
    # Sum up all the numbers in nlist.
    while i < count:
        # Add each item from list to sum
        sum = sum + nlist[i]
        i = i + 1

    return (sum / count)

samples = []

while True:
    t = system.temp_C()
    samples.append(t)
    if len(samples) == 5:
        average = avg_list(samples)
        samples.clear()
        print("Average temp=", average)
        check_baseline(average)

    sleep_ms(200)
```

Objective 8 - Accelerometer

Your 'bot can detect **impacts** with other objects, changes in **motion**, and **orientation**.

- All thanks to the 🔧CodeBot Accelerometer, the tiny chip shown at right!
- CodeBot's accelerometer measures the force of acceleration in *3-directions:* **X**, **Y**, and **Z**.

Pulling some *g's!*

In the picture at right, if the circuit board is positioned flat (horizontal) and motionless on Earth, then it will have ***1g*** pulling down in the **-Z** direction.

- In *physics* the letter **g** means Earth's gravitational acceleration *(approximately 9.8m/s^2)*
- So in this *motionless* case you would expect the ***accelerometer*** to measure:
  - `X` = **0 g** (pointed toward the horizon, no significant gravitational acceleration)
  - `Y` = **0 g** (ditto, horizontal)
  - `Z` = **-1 g** (Earth's gravity pulling straight down, *opposite* to the **+Z** direction)

Concept: *Accelerometer Orientation API*

The 🔧CodeBot Accelerometer is a MEMS accelerometer.

- **MEMS** stands for *"Micro-Electro-Mechanical System"*
  - Inside this little chip are tiny silicon structures that really move!
  - ...and of course, electronic components to sense them.

The **botcore** 🔧library exposes the `accel` object, which provides access to the *chip's* many capabilities.

Some highlights of basic orientation functions:

```python
read()  # Read current axis values.
        # Returns a tuple (x, y, z) of ints.
        # 16-bit signed int range: -32767 to +32768
        # Default full-scale acceleration = ±2g

dump_axes()  # Print 3-axis values to debug console.
```

Now That You're *Oriented*

What value do you expect `accel.read()` to return for the "horizontal" case above?

- Seems like `(0.0, 0.0, -1.0)` would make sense, right?
- *But wait!* According to the API note, the `read()` function returns a 🔧tuple of 🔧integer, not 🔧float values.
- The values are 16-bit signed ints which max-out at 32,768 (2^{15})

- So the max full-scale value of +2g would be 32,768.
- That means our -1g would be (-32767 / 2) = **-16,383**

Create a New File!

Use the **File** → **New File** menu to create a new file called ***AccelTest***.

Check the 'Trek!

- Write some code to test the **Z = -16,383** theory.
- Make a `while True:` loop that constantly *prints* the 3-axis values.
  - Using `accel.dump_axes()` will print it to the *console* for you!
  - Use `sleep_ms(200)` to slow it down a bit.

Run It!

So... how level is your desk?

- You'll need to support the front of your 'bot to keep the circuit-board level.
  - When the **Z-axis** reads **about** -16383 you'll know it's level!
- Notice that the X-axis doesn't change much as you lift the front of your bot.
  - But what happens if you turn CodeBot on its side?
  - Try resting it on its *right wheel*, with **X-axis** pointing *skyward.*
- What value does **Z** have when the 'bot is *upside down?*

CodeTrek:

```
from botcore import *
from time import sleep_ms
```

Don't forget to import `sleep_ms` **from** `time`!

```

while True:
    # TODO: call accel.dump_axes()
```

`accel.dump_axes()` prints to the console *for you!*

- *Just call it!*

```
    # TODO: sleep for 200ms
```

Sleep for `200`ms using `sleep_ms`.

- `sleep_ms(200)`

Goals:

- **Call** `accel.dump_axes()`.
- **Call** `sleep_ms(200)`.
- Import `sleep_ms`.

Tools Found: Accelerometer, import, tuple, int, float, Print Function

Solution:

```
from botcore import *
from time import sleep_ms
```

```
while True:
    accel.dump_axes()
    sleep_ms(200)
```

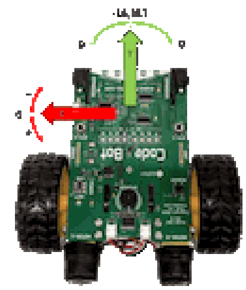
Objective 9 **- Reach for the Stars**

One way to put the 🔧CodeBot accelerometer to use is for **navigation**.

- You already know how to track your bot's **orientation** very precisely.
- With such knowledge you could create a *balancing vehicle* or a *quadcopter drone*...

  But for your first *accelerometer-driven-motor* project, keep it simple.

Rotate to Face *Skyward!*



Your challenge is to monitor the orientation, and control the 🔧motors to keep your nose pointed up at all times.

- Take a look at the image to the right, with the bot **oriented vertically.**
- You need to rotate the 'bot until **Y** is *negative* and **X** is near *zero.*
  - A **Y** value close to **-16,383** would be pointing *straight up!*
  - *But* **Y** is *negative* over a wide range.
- Maybe you could use just the **X-axis** to determine *rotation* direction!?
  - If **X** is *negative,* rotate *counter-clockwise;*
  - If **X** is *positive,* rotate *clockwise.*

Check the 'Trek!

Now add some *action* in the `while True:` loop of your **AccelTest** program!

- Implement the ***Face Skyward!*** challenge!
- Try using just the **X** value from the accelerometer to control the 🔧motors.

Run It!

Test this out on an *inclined surface!*

- A piece of sign-board works great, or even a large book will do.
- Does your 'bot *attempt* to face uphill, as you change the incline?

My 'bot looks pretty ***wobbly*** running this code! How about *yours?*

- You can see that it's *trying* to face uphill, but it keeps moving back and forth!

CodeTrek:

```
from botcore import *

motors.enable(True)
SPEED = 50
```

Set a `SPEED` 🔧constant and *enable* the 🔧motors!

```

while True:
    x, y, z = accel.read()
```

Yes, you *can* assign the 🔧tuple returned by `accel.read()` to a 🔧*variable list.*

```
    print((x,y,z))
```

What's up with the *double-parentheses* in the `print()` statement?

- It bundles `(x,y,z)` together as a tuple for `print()`

```

    if x < 0:
        # Rotate counter-clockwise at SPEED
        # TODO: code the motors.run() part
```

You've done this before!

- Run the `LEFT` motor **negative** and the `RIGHT` motor **positive.**

```
motors.run(LEFT, -SPEED)
motors.run(RIGHT, SPEED)
```

```
    else:
        # Rotate clockwise at SPEED
        # TODO: code the motors.run() part
```

Now run the `LEFT` motor **positive** and the `RIGHT` motor **negative!**

Goals:

- **Assign** `x, y, z` as `accel.read()` in *one line!*
- `if x < 0:`
- **rotate** your 'bot *counter-clockwise!*

`else:`

- **rotate** your 'bot *clockwise!*

Tools Found: Accelerometer, Motors, tuple, Variables, Constants

Solution:

```
from botcore import *

motors.enable(True)
SPEED = 30

while True:
    x, y, z = accel.read()
    print((x,y,z))

    if x < 0:
        motors.run(LEFT, -SPEED)
        motors.run(RIGHT, SPEED)
    else:
        motors.run(LEFT, SPEED)
        motors.run(RIGHT, -SPEED)
```

Objective 10 - Upgrade!

Okay, this code needs some work!

When you face a new coding challenge, often the best approach is: ***"The simplest thing that could possibly work."***

- Many times you'll find the simple solution works great!
- ...and when it *doesn't,* **you learn something!**

In this case you've learned that CodeBot will continuously *overshoot* the top position.

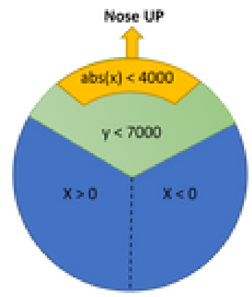
- It *"oscillates*" back and forth. *Not what you want!*
- So this **simplest solution** needs improvement.

Check the 'Trek!

Okay, time for some improvements to your code!

- There are a lot of ways to code this... Feel free to experiment on your own.
- To the right is a diagram that might help you think about how the **X** and **Y** values vary when the 'bot rotates on an incline.
- Since the CodeBot accelerometer's **Y-axis** is angled *downward* at rest, the value **7000** is an approximation of *"level"*. *(Feel free to adjust that value based on your measurements!)*
- The range of `abs(x) < 4000` is an approximate zone where you may want to *slow down* the rotation so you don't overshoot the top.

The code in the 'trek is just like the *simple* approach from the previous objective, but *adds* a ***slow down*** proportional to the **X** value near the top.



Run It!

This should allow your 'bot to *face uphill* more steadily.

- Notice when it centers on the *top*, the *motor adjustments* are very small!

Test this code thoroughly!

- Try it on a surface you can move around.
- Also try watching the *console* as you rotate the 'bot in your hand.

There ***so much more you can do*** with making the robot respond to its *orientation.*

Keep experimenting!

CodeTrek:

```
from botcore import *

motors.enable(True)
SPEED = 50

while True:
    x, y, z = accel.read()
    print((x,y,z))


    if y < 7000 and abs(x) < 4000:
```

Lets go through **both** of these statements *individually!*

`y < 7000`

- If `y` is *at* or *below* "level".
- To be simple, *if the nose isn't pointing up!*

`abs(x) < 4000`

- If `x` is between `-4000` and `4000`.
- Think of it as *"if the wheels have a significantly different tilt"*.

- Balance your 'bot on it's side on one of it's **wheels.** That's a *very different* tilt!

When this `if` statement triggers, your 'bot is *nearly* vertical! *Go slow!*

```
        # Near vertical: set speed proportional to X-axis
        rot_spd = SPEED * (x / 4000)
```

Almost there!

- The *closer* to **pointing up,** the *slower* `rot_spd` will be!

```
    elif x < 0:
        # TODO: Set rot_spd when x < 0
```

When this statement triggers your 'bot is *far* from **pointing up.**

- Rotate to *level out* the wheels!

```
rot_spd = -SPEED
```

```
    else:
        rot_spd = SPEED

    # Rotate
    motors.run(LEFT, rot_spd)
    motors.run(RIGHT, -rot_spd)
```

Use your `rot_spd` variable to ***power*** your motors!

Goals:

- Use `abs(x)` in an `if` statement.
- When your 'bot is *near vertical*, set `rot_spd` to `SPEED * (x / 4000)`.

Tools Found: Accelerometer, Print Function, Variables, Motors

Solution:

```
from botcore import *

motors.enable(True)
SPEED = 50

while True:
    x, y, z = accel.read()
    print((x,y,z))


    if y < 7000 and abs(x) < 4000:
        # Near vertical: set speed proportional to X-axis
        rot_spd = SPEED * (x / 4000)
    elif x < 0:
        rot_spd = -SPEED
    else:
        rot_spd = SPEED

    # Rotate
    motors.run(LEFT, rot_spd)
    motors.run(RIGHT, -rot_spd)

```

***Quiz 1* - Checkpoint**

Question 1: Approximately what value would the **Y-axis** have if you pointed CodeBot toward the sky?

(*🔧Proximity sensors pointed up)*

✓ `-16,383`

✗ `-1.0`

✗ `+16,384`

✗ `0`

Question 2: The **X-axis** value would be *zero* if your 'bot was facing straight up.

- In the code above, what would the speed `rot_spd` be in that case?

✓ `0`

✗ `-SPEED`

✗ `+SPEED`

✗ `50`

Question 3: What would the value of `y` be after the following assignment statement?

```
x, y, z = (30, 20, 40)
```

✓ `20`

✗ `30`

✗ `40`

✗ `(30, 20)`

✗ `(30, 20, 40)`

Question 4: The `accel.read()` function returns a 🔧tuple with **3** integers (x, y, z).

If you wrote: `vals = accel.read()`, which *axis* would `vals[1]` refer to?

✓ Y-axis

✗ X-axis

✗ Z-axis

Objective 11 **- Guard Bot**

Powerful and *Sensitive!*

Prepare to be amazed at the capabilities of the 🔧CodeBot accelerometer. *Ready to **pounce** at the slightest movement!*

- There are quite a few advanced capabilities *built into the hardware* of this device.
  - It even has the capability to detect motion based on programmable *thresholds* for any of its 3-axes.
  - Configuring all those built-in capabilities is left for a future project...
- Your **current challenge** is to combine your *accelerometer knowledge* with ***Python code*** to create a *sensitive motion detector!*

Algorithm

1. In a loop, continuously *sample* the accelerometer with `accel.read()`
2. Each time, compare the values from the previous `read()` to the current one.
3. If the ***difference between readings*** is greater than a configured *sensitivity threshold* `SENS`, sound the ALARM!
4. Wait a configured `DELAY` *ms* between samples, to allow time for motion to happen.

Create a New File!

Use the **File** → **New File** menu to create a new file called ***GuardBot***.

Check the 'Trek!

- Define a 🔧function `def alarm()` that will sound a short *"alert"* tone.
- Make 🔧constants for `SENS` and `DELAY` to control sensitivity.
- Inside your `while True:` loop:
  - Use `sleep_ms(DELAY)` between samples.
  - Save the *previous* sample to compare with the *current* one.
  - Just compare difference in **X-axis** values: `dx = now[0] - before[0]`
  - Use `abs()` to get the *magnitude* of the difference for comparison with `SENS`.

Run It!

Try some different values for `SENS` and `DELAY`.

- Pretty sensitive, right?

CodeTrek:

```
from botcore import *
from time import sleep_ms

def alarm():
```

`alarm()` sounds a *short* **alert** tone.

- You'll call it when your 'bot senses *movement*.

```
    # Alert - motion detected!
    print("Alarm!")

    # TODO: Make a short beep-boop with spkr.
    #      (keep under 200ms total)
```

Sound the alarm! This is as *simple* as the **following steps:**

1. Turn the speaker **on.**
2. Wait a bit (start with `200ms`).
3. Turn the speaker **off.**

I implemented it like this:

```
spkr.pitch(500)
sleep_ms(200)
spkr.off()
```

```

# Sensitivity Configuration
SENS = 50   # accelerometer value difference
DELAY = 100 # time between samples
```

`SENS`

- Represents the *sensitivity threshold*.
- Determines how much `x` needs to change to *trigger the alarm!*

`DELAY`

- Represents the *time between samples*.
- Determines how *frequently* we check if the alarm is triggered.

```

# Take first sample (x, y, z)
now = accel.read()
```

now represents *the most recent* sample!

- Go ahead a take one now to **initialize** it.

```

while True:
    # Delay for motion to happen
    sleep_ms(DELAY)
```

Apply the **DELAY** with `sleep_ms`.

```

    # Remember last sample
    before = now
```

`before` represents the *last* sample.

- You just called `sleep_ms(DELAY)`, `now` is old!
- **Assign** `now` to `before`, you'll *update* `now` on the next line.

```

    # Take a new sample
    now = accel.read()
```

Refresh `now`!

```

    # Calculate X-axis difference (movement)
    dx = now[0] - before[0]
```

Calculate the *difference* between `now` and `before`!

```

    # Compare magnitude of difference to threshold
    if abs(dx) > SENS:
        alarm()
```

If `x` is *outside* the range of `SENS` to `-SENS`, your 'bot moved!

- *Trigger the alarm!*

Goals:

- **Define** a 🔧function called `alarm()`.
- *Sound the alarm* by:
  1. Turning the speaker **on.**
  2. Waiting a bit (using `sleep_ms`).
  3. Turning the speaker **off.**
- Compare the *difference* in **X-axis** values by assigning `dx` as `now[0] - before[0]`.

Tools Found: Accelerometer, Functions, Constants

Solution:

```python
from botcore import *
from time import sleep_ms

def alarm():
    # Alert - motion detected!
    print("Alarm!")
    spkr.pitch(500)
    sleep_ms(200)
    spkr.off()

# Sensitivity Configuration
SENS = 50   # accelerometer value difference
DELAY = 100 # time between samples

# Take first sample (x, y, z)
now = accel.read()

while True:
    # Delay for motion to happen
    sleep_ms(DELAY)

    # Remember last sample
    before = now

    # Take a new sample
    now = accel.read()

    # Calculate X-axis difference (movement)
    dx = now[0] - before[0]

    # Compare magnitude of difference to threshold
    if abs(dx) > SENS:
        alarm()
```

Objective 12 - Guard Bot 2: Guard Harder

Detecting on All 3 Axes

Your program is ***great*** at *detecting movement* on the **X-axis.**

- What if someone were to *carefully* whisk your 'bot away while keeping the wheels **level?**
    - The alarm *wouldn't* sound!
- You can fix that gap by *detecting movement* on all **3 axes!**
    - **X**, **Y** *and* **Z**

Check the 'Trek!

Modify your code to detect motion in all 3 directions.

- Use the or operator to combine multiple comparisons!

Run It!

You may want to adjust `SENS` and `DELAY` again.

- Can you make it *impossible to move CodeBot* without detection?

CodeTrek:

```python
from botcore import *
from time import sleep_ms

def alarm(dx, dy, dz):
```

Update `alarm()` to accept `dx, dy, dz` as arguments.

```
    # Alert - motion detected!
    print("Alarm: ", (dx, dy, dz))
```

🔧Print the **axis values** that caused the *alarm to trigger!*

```

    spkr.pitch(500)
    sleep_ms(200)
    spkr.off()


# Sensitivity Configuration
SENS = 50   # accelerometer value difference
DELAY = 100 # time between samples

# Take first sample (x, y, z)
now = accel.read()

while True:
    # Delay for motion to happen
    sleep_ms(DELAY)

    # Remember last sample
    before = now

    # Take a new sample
    now = accel.read()


    # Calculate difference (motion)
    dx = now[0] - before[0]
    dy = now[1] - before[1]
    dz = now[2] - before[2]
```

Calculate the *difference* in motion for ***all 3 axes.***

```

    if # TODO: trigger if ANY of the 3 axes break the SENS thresh
        alarm(dx, dy, dz)
```

Last objective you used `abs(dx) > SENS` to determine if the 'bot had been moved.

- Use **all 3 axes** this time!
- `if abs(dx) > SENS or abs(dy) > SENS or abs(dz) > SENS:`

Goals:

- **Trigger** the `alarm` if ANY of the **3 axes** break the `SENS` threshold.
- **Update** `alarm()` to take `dx, dy, dz` as 🔧arguments.

Tools Found: Logical Operators, Keyword and Positional Arguments, Print Function

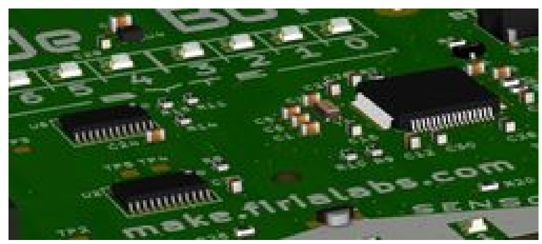
Solution:

```
from botcore import *
from time import sleep_ms

def alarm(dx, dy, dz):
    # Alert - motion detected!
    print("Alarm: ", (dx, dy, dz))

    spkr.pitch(500)
    sleep_ms(200)
    spkr.off()
```

```python

# Sensitivity Configuration
SENS = 50   # accelerometer value difference
DELAY = 100 # time between samples

# Take first sample (x, y, z)
now = accel.read()

while True:
    # Delay for motion to happen
    sleep_ms(DELAY)

    # Remember last sample
    before = now

    # Take a new sample
    now = accel.read()


    # Calculate difference (motion)
    dx = now[0] - before[0]
    dy = now[1] - before[1]
    dz = now[2] - before[2]

    # Compare differences to threshold
    if abs(dx) > SENS or abs(dy) > SENS or abs(dz) > SENS:
        alarm(dx, dy, dz)

```

Mission 9 Complete

You've examined *many* of your bot's *subsystems*



- **Battery** health will no longer be a mystery to you!
- And you can *sense* if things are starting to **heat up...** or get ***chilly.***
- Your code can behave differently based on **orientation**.
- And respond instantly to even the *slightest* ***movement!***
- ...and there's *so much more* to **explore!**

You already *use* this kind of code daily!

- Your phone tracks and displays its battery usage.
- Electronic thermostats control temperature in most buildings.
- Accelerometers are used in everything from smart-watches to game-controllers.
- ...now **you** are *in the game.* Code On!

Try Your Skills

Suggested Re-mix Ideas:

- **Battery Life Experiment:** Which brand/type of batteries last the longest?
  - *Set up a test!* Every 5 minutes `print()` the voltage to the *debug console*.
  - Also print `ticks_ms` from the time module to track *time* and detect *reboots*.
- **Fall Detector:** Use the CodeBot accelerometer to detect if your 'bot is falling.
  - Naturally your 'bot will want to *scream* if that's the case. *Make it so!*
  - *Clue:* When you're in free-fall (or outer space) it's nearly **zero-g** in all 3-axes!
- **Bump-Bot:** Move forward until you detect an impact. Then rotate a random amount and go again!
  - Okay, not very *graceful*... but *fun!*