Python Level-1 with Virtual Robotics

Learn Python with virtual robotics challenges and a pathway to Certification.

Mission 1 - Welcome

Take a tour of the CodeSpace Development Environment.

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.

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 *virtual* or *physical* device?

✓ Yes. This is simple!

✗ It looks too complicated.

✗ I don't think I can.

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

✓ How to move the camera

- ✓ How to open an objective
- ✗ How to run a half marathon
- ✗ How to control the weather

Mission 2 - Introducing CodeBot

Get to know your friendly neighborhood Virtual 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!

Locate 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***

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

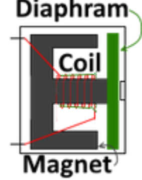
Solution:

N/A

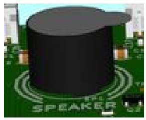
***Objective 3* - Speaker**

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

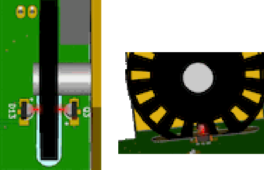
Solution:

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.

Goal:

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

Tools Found: Wheel Encoders

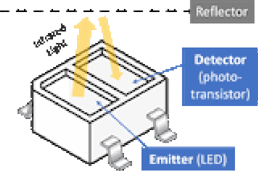
Solution:

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

Mission 3 **- Light the Way**

Write some Python code to light up those LEDs and get CodeBot flashing.

Objective 1 **- Hello, LED!**

Embedded Programming's "Hello World"

You may have heard about the concept of a "Hello World" program.

- That's traditionally the first program you write when you learn a new language.
- But what about an *embedded system,* like a robot that has no text-display?

Ya BLINK an LED ☺

Yes, a single LED may not seem exciting. But from such humble beginnings, *massive starships* are built...

So, here's your code:

Type this into the text editor (left side of screen).

- Go ahead and delete any *sample code* that's already there, and type in the following:

```
from botcore import *
leds.user_num(0, True)
```

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

To RUN this on your CodeBot, press the ▶ button!

It's at the top of your screen just above the **text editor**.

CodeTrek:

```
1  from botcore import *
```

This line gives your Python program access to CodeBot's hardware.

- You are `import`ing from the `botcore` library module.
- The "*" means import *everything!*

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

The `leds` symbol is from the **botcore** library you *imported* above.

- You are invoking a function `user_num(num, is_on)`
- with arguments: `num = 0` and `is_on = True`
- This will turn USER LED number `0` on!

```
3
```

Goals:

- Open the CodeTrek to learn about your code with the button
- RUN your code to LIGHT *User LED* number `0`

Solution:

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

Objective 2 **- LED Patterns**

LED Binary Patterns

There are 8 red LEDs labeled 0-7 just above the word "**BYTE**"

- "BYTE" means *8-bits*, which are "binary digits"
- And binary means *base-2*, so the digits are '0' (OFF) and '1' (ON)

You could use these "user" LEDs to display **any** number in binary!

- ...um, actually only 2^8 = 256 different numbers.

  Naturally, your Python code can control *ALL* the CodeBot LEDs.

Start by controlling the USER LEDs one bit at a time

Notice the LED numbering starts at **0**.

- They count up from *right to left.*
- The **CodeTrek** will show you the way!

  Sometimes the CodeTrek will be vital to learning more!

This button will open the *CodeTrek* directly from the instructions:

CodeTrek:

```
1  # Access CodeBot's built-in "botcore" library
```

This line is a **comment**.

- Comments are ignored by the computer - they're just for humans :-)
- In Python, comments begin with `#`

```
2  from botcore import *
```

This line gives your Python program access to CodeBot's hardware.

- You are `import`ing from the `botcore` library module.
- The "*" means import *everything!*

```
3
4  # To make the pattern 10011001 there are 4 LEDs to turn on:
5  #   0, 3, 4, 7
6
7  # Set User LED bit number 0 to True (ON)
8  leds.user_num(0, True)
```

Just like in **Objective 1**, this turns *user LED* `0` ON.

```
9
10 # Now do the same for the other three LEDs
11 leds.user_num(...)
12 leds.user_num(...)
13 leds.user_num(...)
```

Fill in the blanks for these 3 lines!

- Replace the (...) with the right code to turn ON the other three LEDs.

```
14
```

Goals:

- Run through the CodeTrek using the button in the instructions of the objective panel.
- Display the following pattern with the CodeBot user LEDs:

1	0	0	1	1	0	0	1

Tools Found: Binary Numbers, CodeBot LEDs

Solution:

```
1  # Access CodeBot's built-in "botcore" library
2  from botcore import *
3
4  # To make the pattern 10011001 there are 4 LEDs to turn on:
5  #   0, 3, 4, 7
6
7  # Set User LED bit number 0 to True (ON)
8  leds.user_num(0, True)
9
10 # Now do the same for the other three LEDs
11 leds.user_num(3, True)
12 leds.user_num(4, True)
13 leds.user_num(7, True)
14
```

Objective 3 - Basic Binary

Basic Binary

Wait a minute. Is all this binary stuff just *"Gratuitous Educational Content?"*

Absolutely not! I wouldn't do that to you!

- The 🔧CodeBot LEDs really are controlled by a binary 🔧shift register.
- AND the *fastest* way for your Python code to set the LEDs is *directly in binary*.

Instead of the `leds.user_num(num, isOn)` function, you can do the following:

```
# Set User LEDs with a binary pattern
leds.user(0b01100110)
```

The code above uses Python's literal binary notation.

- Prefixing a number with `0b` tells Python it's *binary.*
- With no prefix naturally it's gonna assume a *decimal* number (base-10).

Sometimes there will be a single important concept to learn about.

- Click this button to see an important concept in the CodeTrek:

Go ahead and test out this new 🔧API *(**A**pplication **P**rogramming **I**nterface).*

- Display some different *binary* numbers.
- You can also give `leds.user(val)` decimal values.

CodeTrek:

```
1  from botcore import *
2
3  # Light ALL the User LEDs
4  leds.user(0b11110111)
```

This line of code uses the binary API to control the user LEDs.

- A prefix of `0b` followed by 1's and 0's is a *binary* value.
- There are 8 LEDs, so you need 8 binary 1's.
- This looks *really* close, but one of the digits is wrong. Fix it and run the code!

```
5
```

Goals:

- Open the CodeTrek using the button in the instructions to learn a critical concept.
- Complete this Objective by turning on ALL the User "BYTE" LEDS

Tools Found: Binary Numbers, CodeBot LEDs, Binary Shift Register, API

Solution:

```
1  from botcore import *
2  leds.user(0b11111111)
```

***Quiz 1* - Enlightenment**

Question 1: What is an **API**?

✓ It stands for "Application Programming Interface", describing how code interfaces with *other* code.

✗ The "Automatic Peripheral Interface". This is how keyboards and other devices automatically interface with cyberspace.

✗ Pronounced like "happy" without the *H*, it's the feeling you get when you fix a bug in your code.

Question 2: Which of the following are valid 🔧binary literals in Python?

✓ `0b0101`

✗ `0b0002`

✗ `bin0101`

✓ `0b11`

Question 3: Assuming all **LEDs** are off to begin with, what is the equivalent 🔧binary form of the following?

```
leds.user_num(3, True)
```

✓ `leds.user(0b00001000)`

✗ `leds.user(3)`

✗ `leds.user(0b00010000)`

✗ `leds.user(0b00000100)`

Objective 4 **- All the LEDs**

All the LEDs

So far you've programmed 8 LEDs.

- There are 9 more to go!

This code demonstrates the botcore API for the remaining 🔧CodeBot LEDs:

```
# Line sensor 0-4
leds.ls_num(0, True)  # Or binary: leds.ls(val)

# Proximity sensor 0-1
leds.prox_num(0, True)  # Or binary: leds.prox(val)

# USB
leds.usb(True)

# Power
leds.pwr(True)
```

Notice when there are multiple LEDs we always start numbering at 0.

- Just like with the *User* LEDs, there's a matching 🔧binary API for the others also.
- You'll soon learn about Python 🔧lists which also have *indexes* starting at 0.

Write code to Light ALL the LEDs!

CodeTrek:

```
1  from botcore import *
2
3  # User
4  leds.user(0b11111111)
5
6  # Line sensor 0-4
7
```

Add your code here

Get the rest of those LEDs shining!

```
8
9  # Proximity sensor 0-1
10
11  # USB
12
13  # Power
14
```

Goals:

- Light all the ***User*** LEDs
- Light all the ***Line Sensor*** (ls) LEDS
- Light the two ***Prox*** LEDs
- Light the ***USB*** LED
- Light the ***Power*** LED

Tools Found: CodeBot LEDs, Binary Numbers, list

Solution:

```
1  from botcore import *
2
3  # User
4  leds.user(0b11111111)
5
6  # Line sensor 0-4
7  leds.ls(0b11111)
8
9  # Proximity sensor 0-1
10  leds.prox(0b11)
11
12  # USB
13  leds.usb(True)
14
15  # Power
16  leds.pwr(True)
```

Objective 5 - Animation

Animation

You've mastered the basics of 🔧LEDs!

- But wait. ☹
- What about *blinking* LEDs?
  - ...or dazzling sequences of scintillating light??

Animation - a ***sequence*** of changes, at a controlled ***speed***.

- Python executes each line of code in sequence, from top to bottom.
  - So you already have the *sequence* covered. The computer is just too FAST!
  - All the lights appear to come on at the same time...

Sloooow it down with the `sleep(seconds)` function from the 🔧time module:

```
from time import sleep  # You just need this line of code once!

sleep(0.5)  # Delay the program for 0.5 seconds
```

Now, armed with this new function, let's see you **Animate those LEDs**

- I want to see *at least one* LED changing over time...
- **Blink at least twice** to complete this objective!

CodeTrek:

```
1  from botcore import *
2  from time import sleep
```

Import the `sleep()` function from the **time** module.

```
3
4  # Blink once
5  leds.user(0b11111111)
```

Light some LEDs

```
6  sleep(1.0)
```

Wait a second...

```
7  leds.user(0)
```

Change the LEDs!

```
8
9  # TODO...
```

Finish the code!

- You need to wait a bit, then turn some LEDs back ON.
- ... then wait, and turn them OFF again!

```
10
11  # Blink again
12  leds.user(0b11111111)
13
```

Goal:

- LED blink (on/off) with a time delay TWICE

Tools Found: LED, Time Module

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # Blink once
5  leds.user(0b11111111)
6  sleep(1.0)
7  leds.user(0)
8
9  sleep(1.0)
10
11  # Blink again
12  leds.user(0b11111111)
13  sleep(1.0)
14  leds.user(0)
```

Mission 4 **- Get Moving**

Get your motors running... Head out on the Virtual Highway!

***Objective 1* - Python Pirouette**

Rotate in Place

A nice test for the 🔧motors is to **rotate** your robot.

- CodeBot ***rotates*** when the wheels turn with *equal* power in *opposite directions.*
- The motors automatically *stop* when the program ends.
    - *(Soon your programs won't end... they may 🔧loop forever!)*
- For your first programs, just call `sleep()` to let the motors run briefly.
    - Refer to the 🔧time module for more information on that!

Use the CodeTrek to see how the the *motors* 🔧API works:

CodeTrek:

```
1  from botcore import *
2  from time import sleep
```

Get `sleep()` from Python's **time** module.

- In this case you're not *animating LEDs*...
- Instead, this is needed so you can run the motors for a specific time duration.

```
3
4  # Enable the motors
5  motors.enable(True)
```

You have to `enable()` the motors before they'll move.

```
6
7  # Apply 30% FORWARD power to the LEFT motor
8  motors.run(LEFT, 30)
9
10 # Apply 30% REVERSE power to the RIGHT motor
11 motors.run(RIGHT, -30)
```

Run the specified motor at given power level: **-100% to +100%**

```
12
13 # Sleep while motors run... they stop when program ends!
14 sleep(5.0)
```

If you don't *wait* here, the motors will stop immediately when your program ends.

- Pretty much before they've had a chance to start moving!

Hints:

- One wheel needs to have a (+) positive speed, and the other wheel should have *exactly the same speed* but (-) negative direction.
- It doesn't matter which wheel goes forward and which goes backwards.
    - Either one counts as a nice *spin move!*

Goal:

- Rotate your 'bot by turning the wheels in equal and opposite directions.

Tools Found: Motors, Loops, Time Module, API

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5  motors.run(LEFT, 50)
6  motors.run(RIGHT, -50)
7
8  sleep(2.0)
```

Objective 2 **- Circle Up**

Circle Up

Can you write Python code to make CodeBot drive forward in a *circle?*

- The wheels will need to move at *different speeds.*
  - *How* different will determine the **diameter** of your circle!

CodeTrek:

```
1  from botcore import *
2  # from ???
```

Besides `botcore` you need another `import` module here...

- Fix this code so you can `sleep()` later!

```
3
4  # Start your engines!
5  motors.enable(True)
6  # motors.run(LEFT, ???)
7  # motors.run(RIGHT, ???)
```

Set power levels to move forward in a circle.

- You decide how your bot's speed and how big the circle is!

```
8
9  # Blink some LEDs
10 leds.user(0b00011000)
```

Customize your flashing lights!

- The motors are still running while you're blinking.

```
11 sleep(2)
12 leds.user(0b11000011)
13 sleep(2)
14 leds.user(0)
```

Hints:

- Both wheels need to move *forward*.
  - That means your `motors.run()` power will be *+positive* for both LEFT and RIGHT motors.
- Blinking LEDs just means changing them, with some `sleep()` in between.

Goals:

- Drive the CodeBot forward in a circle
- Blink some LEDs while moving!

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # Start your engines!
5  motors.enable(True)
6  motors.run(LEFT, 30)
7  motors.run(RIGHT, 40)
8
9  # Blink some LEDs
10 leds.user(0b00011000)
11 sleep(2)
12 leds.user(0b11000011)
13 sleep(2)
14 leds.user(0)
```

Objective 3 - Robot Tag

Robot Tag Race

You have all the skills you need to complete the next Objective of this Mission!

It's simple ☺

- Write a Python program for CodeBot to touch all the tennis balls.
- To make it a bit more challenging, there's a time limit.

30 seconds!

- That's the maximum qualifying time for touching at least **two** of the tennis balls.

Use the RESET button on the *Simulation Toolbar*

- This will **reposition** *CodeBot* and the *tennis balls.*
- Try the ***Universal Camera*** to set an *overhead* view!
- It's probably gonna take a few attempts...

Having trouble?

Check the 💡 **Hint** panel by clicking the icon below.

CodeTrek:

```
1  # Robot Tag Race - Just ONE way to solve it...
2
3  from botcore import *
4  from time import sleep
```

You're going to need some *library modules:*

- `import botcore` to access CodeBot's *motors*
- `import time` so you can access `sleep()`

```
5
6  motors.enable(True)
```

Don't forget to *enable* those motors!

```
7
```

```
8  # Drive to the 1st ball
9  motors.run(LEFT, 60)
10 motors.run(RIGHT, 40)
11 sleep(2.5)
12
13 # Adjust speeds and timing to reach the 2nd ball
14 # motors.run(LEFT, ??)
15 # motors.run(RIGHT, ??)
16 # sleep(??)
17
18 # TODO: more code to reach the 3rd and 4th balls!
19
20
```

There are *many* possible solutions!

Some are elegant, others more *brute-force* like the one shown here.

Hints:

- **Take it one ball at a time**
  - Experiment to find *motor speeds* and *sleep delay* needed to hit the first ball.
  - Then add code to adjust *speeds* and another *delay* to hit the second one.
- **Notice your bot doesn't move exactly the same every time?**
  - In *real-life* robotics, wheels are never perfectly round, and no surface is perfectly smooth.
  - Motors and gears will have slightly different efficiency and friction too.
  - In future projects you'll learn now to *navigate **accurately*** . *Line Sensors* and *Wheel Encoders* will provide the **feedback** your 'bot needs to move *precisely!*

Goals:

- Hit **two** tennis balls!
- *All within a 30 second timeout!*

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5  motors.run(LEFT, 60)
6  motors.run(RIGHT, 40)
7  sleep(2.5)
8
9  motors.run(LEFT, 70)
10 motors.run(RIGHT, 50)
11 sleep(3.5)
12
13 motors.run(LEFT, 70)
14 motors.run(RIGHT, 60)
15 sleep(2.5)
16
17 motors.run(LEFT, 70)
18 motors.run(RIGHT, 55)
19 sleep(5.5)
```

Quiz 1 - Move Your Brain

Question 1: While executing the following code, which direction does **CodeBot** rotate?

```
from botcore import *
from time import sleep

motors.run(LEFT, 50)
motors.run(RIGHT, -50)
sleep(5)
```

✓ It doesn't rotate at all. You forgot to call `motors.enable(True)`

✗ Clockwise

✗ Counter-clockwise

Question 2: What does `sleep(N)` do?

✓ Pauses program execution for `N` seconds, so the program doesn't end and peripherals are still enabled.

✗ Commands the motors to run for the specified time interval `N` in seconds.

✗ Disables all peripherals and enters *power save* mode for `N` seconds.

Question 3: How is your Python code able to call the `sleep()` function?

✓ By importing it from the time module

✗ The `sleep()` function is a Python built-in, so it is always available.

✗ Everything must `sleep()` at some point. Even computers.

Question 4: Where does the `motors.run()` function come from?

✓ The `motors` object is part of the `import botcore` module.

✗ Python was created with **CodeBot** in mind from the beginning, so the `motors` object is always available to every Python program.

✗ Calling `motors.enable()` makes it available.

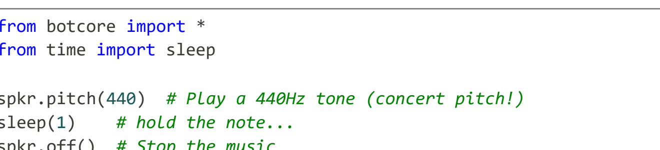
Objective 4 **- Sound Off**

Sound Off!

Okay, here's your final Objective to complete this Mission.

- You've lit the CodeBot LEDs,
- You mastered the motors,
- It's time for you to make some *SOUND* with the speaker!

The API is pretty simple:



```
from botcore import *
from time import sleep

spkr.pitch(440)  # Play a 440Hz tone (concert pitch!)
sleep(1)     # hold the note...
spkr.off()  # Stop the music
```

3D Sound

You may need to use the camera controls to zoom-in near CodeBot to hear the sound better!

Not much to it

- But with this simple capability you can create *infinite melodies!*
- Experiment a little with high and low *frequencies* `spkr.pitch(frequency)`

  - What's the lowest and highest audible frequency you can make?
- Try playing two or more notes separated by `sleep()` delays.
  - Now you're making music!

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  spkr.pitch(440)  # Play a 440Hz tone (concert pitch!)
5  sleep(1)    # hold the note...
6
7  # TODO: Play another note
```

Copy and modify the two lines above to play a different note!

```
8
9  spkr.off()  # Stop the music
```

Goal:

- Play two or more *different* notes separated only by delays

Tools Found: CodeBot LEDs, Motors, Speaker, API

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  spkr.pitch(440)  # Play a 440Hz tone (concert pitch!)
5  sleep(1)    # hold the note...
6
7  spkr.pitch(880)
8  sleep(1)
9
10 spkr.off()  # Stop the music
```

Mission 5 - Dance Bot

Does CodeBot have what it takes to win a dance competition? Only your code can make it happen!

Objective 1 - Ah One-Two-Three!

You’ve gotta count your steps to hit those dance moves just right!

Create a new file!

- Use the File → New File menu to create a new file called "dancebot.py"

Get Flashy

Say you want to flash User LED `0` exactly 8 times.

- You could copy the same ON/OFF code 8 times to make it happen.
  - But that’s a lot of repetitive code!
  - Even worse, what about the extended-play version where you have to flash 40 times?
- There has to be a better way…

🔧Loops let you repeat a block of code.

Python has two kinds of loops: `while` and `for`. Here's an example loop:

```
while count < 8:
    # blink an LED...
    count = count + 1
```

Note two important things here:

1. There is a colon `:` at the end of the line with `while`. That means a new block of code begins on the next line.
2. The code *inside* the 🔧loop is 🔧indented.

Keeping track of the count

So far your programs have run straight through, controlling 🔧motors, 🔧CodeBot LEDs and 🔧speaker.

- You haven't needed to store any information along the way.
- But *now* you have to keep track of a "count" while your program runs!

Your program will need some **memory** to store "count" in.

- That's what 🔧variables are for!
- You'll be using a 🔧variable to keep track of count as you 🔧loop and blink the 🔧CodeBot LEDs.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  count = 0
```

This is how you declare a *variable* in Python!

- Just use a valid 🔧variable name, and assign a value to it.
- In this case, the initial value of `count` is `0`

```
5
6  # Blink User LED-0 exactly 8 times
7  while count < 8:
```

Your `while condition:` loop:

- The loop will repeat *while* the **condition** is `True`
- Since `count` starts at `0` you just need to add 1 to it each time through the loop.
- ...then when it hits `8` the loop will end!

```
8
9      # Blink LED 0 On/Off
10     leds.user_num(0, True)
```

Check out the ***indentation!***

- Use the **TAB** key to 🔧indent all the code you want to run inside the loop.
- If it ain't **indented**, it ain't inside the loop!

```
11     sleep(0.1)
12     leds.user_num(0, False)
13     sleep(0.1)
14
15     # Add one to the count
16     count = count + 1
```

Updating `count` inside your loop

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

That's what is happening with this code.

> *"Add `+1` to `count`, and store the* result *back in `count`."*

Does it look odd to have `count` 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.

`count` **starts at** `0`:

1. Do the **right hand side**: `count + 1` → `0 + 1` → `1`
2. Next, do the **assignment**: `count` ← `1`

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

17

Goals:

- Blink User LED `0` on and off 8 times.
- Use a `while` loop in your program.

Tools Found: Loops, Indentation, Motors, CodeBot LEDs, Speaker, Variables

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # Blink User LED-0 exactly 8 times
5  count = 0
6  while count < 8:
7      print(count)
8      leds.user_num(0, True)
9      sleep(0.1)
10     leds.user_num(0, False)
11     sleep(0.1)
12     count = count + 1  # Gnarly!
```

Objective 2 **- Enter the Debugger**

Your code is getting more complex now!

With 🔧variables and 🔧loops in your toolbox, you have the capability to build much more powerful programs.

- BUT, you can also *easily* create code that's hard to understand.
- And of course, your code can have **bugs!**

If you read the background on 🔧debugging you'll know that a "bug" is when your program is not doing what you *expected* it to do.

> But the computer is *always* doing what you **told** it to do!

What is the computer *really* doing?

To find out, *trace* through your code *one step at a time.* You can do this by reading over your program carefully, making some notes, slowly "running" the program with your brain - like a human computer! But this can take time, like solving a hard puzzle. Fortunately there are tools that can help:

- As it runs, your program can 🔧print text messages about what it's doing.
- Rather than pressing the ▶ RUN button, you can press 🐞 DEBUG and have the computer *step* through your code.
  - As you step, you can inspect 🔧variables and interact on the *console*.
  - Click the ☰ button at the lower-right to open the *console* panel.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  count = 0
5
```

```
6  # Blink User LED-0 exactly 8 times
7  while count < 8:
8
9      # Display the count on the debug console
10     print(count)
```

The `print()` statement

- Python's built-in `print()` function sends *text* output to the **console**.
- Here you are using it in a very simple way, just printing a single integer.

```
11
12     # Blink LED 0 On/Off
13     leds.user_num(0, True)
14     sleep(0.1)
15     leds.user_num(0, False)
16     sleep(0.1)
17
18     # Add one to the count
19     count = count + 1
20
```

Goals:

- Display the running `count` on the console from 0 to 7.
  - Just print the ***numbers***... no other text or spaces.
- Step into your code with the CodeSpace Debugger.
  - First click then use the button to *step* through your code.
  - Don't forget to watch your `count` variable in the **console** panel!
  - It will be under **Globals**

Tools Found: Variables, Loops, Debugging, Print Function, Advanced Debugging

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  count = 0
5
6  # Blink User LED-0 exactly 8 times
7  while count < 8:
8
9      # Display the count on the debug console
10     print(count)
11
12     # Blink LED 0 On/Off
13     leds.user_num(0, True)
14     sleep(0.1)
15     leds.user_num(0, False)
16     sleep(0.1)
17
18     # Add one to the count
19     count = count + 1
20
```

Quiz 1 - Dancin' Data

Question 1: Which two of the following are *valid* Python variable names?

✓ `spam_eggs`

❌ `Baker$Dozen`

❌ `1fineday`

✅ `Number1`

Question 2: What is printed by the following code?

```
i = 0
while i < 5:
    print(i, end=' ')
    i = i + 1

print('and', i)
```

✅ 0 1 2 3 4 and 5

❌ 0 1 2 3 4

❌ 1 2 3 4 and 5

❌ 0,1,2,3,4, and 5

Question 3: What is printed by the following code?

```
n = 15
n = n + 5
print('n= ', n)
```

✅ n= 20

❌ n= 5

❌ n= 15

❌ 20

Objective 3 - Iterate the Beat

Another type of loop

The `for` 🔧loop is made for looping across a range of numbers, or 🔧iterating over other kinds of sequences you will soon be learning about.

Use the 🔧built-in 🔧range function to specify the sequence of numbers you need.

- The `for` loop saves you the trouble of initializing and updating the loop 🔧variable
  - It automatically takes the next value from the sequence on each iteration through the loop.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  # Iterate the beat, with a for loop!
5  for count in range(8):
```

Here's your `for` loop

- It iterates over `range(8)`
- ...meaning `count` will range from 0-7
- It's automatically assigned the next value each time around the loop.

```
6       print(count)
7       leds.user_num(0, True)
8       sleep(0.1)
9       leds.user_num(0, False)
10      sleep(0.1)
11
```

No need to update `count`

- The `for` loop takes care of that!

Goals:

- Blink **User LED-0** on and off 8 times.
- Display the running `count` on the *console* from 0 to 7.
  - You can only print the numbers... no other text or spaces.
- Remove the `while` loop and use a `for` loop instead.

Tools Found: Loops, Iterable, Built-In Functions, Ranges, Variables, Readability

Solution:

```
1   from botcore import *
2   from time import sleep
3
4   # Iterate the beat, with a for loop!
5   for count in range(8):
6       print(count)
7       leds.user_num(0, True)
8       sleep(0.1)
9       leds.user_num(0, False)
10      sleep(0.1)
```

Objective 4 - Begin the Wave

Teach CodeBot a *classic* robot dance move

First you need to make the User LED sweep across from right (bit-0) to left (bit-7).

- Your 🔧loop is counting from 0 to 7, so you're nearly there already!

CodeTrek:

```
1   from botcore import *
2   from time import sleep
3
4   # Iterate the beat, with a for loop!
5   for count in range(8):
6       print(count)
7       leds.user_num(count, True)
```

Do something with `count`!

- The LEDs are numbered 0-7, so `count` will be *perfect* here.

```
8       sleep(0.1)
9       leds.user_num(count, False)
10      sleep(0.1)
```

Do you *really* need to `sleep()` here?

- Sometimes the best improvement is to **delete** a line of code!

There's always an LED ***on*** in this dance...

Hints:

- Use `count` as the LED number in the `leds.user_num(num, is_on)` function.
- You don't need to `sleep()` at all after you turn the LED *off.*
  - There's always an LED shining in this *animated display!*

Goals:

- Use a `for` loop in your program.
- Sweep a single 🔧User LED from right to left.

The sequence must be:

1. LED-0 on and all other User LEDs off
2. LED-1 on and all other User LEDs off

   ...

8. LED-7 on and all other User LEDs off

Tools Found: Loops, CodeBot LEDs

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # Iterate the beat, with a for loop!
5  for count in range(8):
6      print(count)
7      leds.user_num(count, True)
8      sleep(0.1)
9      leds.user_num(count, False)
10     # sleep(0.1)
11
```

Objective 5 - Complete the Wave

Now you'll need to add code to make the LED "sweep back" the other direction.

Can you make a `for` 🔧loop count **backwards**?

- Of course you can! It's Python 😊
- Check out the power of the 🔧range tool. It has what you need.

The full 🔧range function: `range(start, stop, step)`

- `start` and `step` are both 🔧optional arguments

Watch the count-up and count-down on the Console.

- Is the program doing what you expect?

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  # Iterate the beat, with a for loop!
```

```
5   for count in range(8):
6       print(count)
7       leds.user_num(count, True)
8       sleep(0.1)
9       leds.user_num(count, False)
10
11  for count in range(6, -1, -1):
```

A **down-counting** `for` loop!

- The `range()` has a `step = -1`
- So `count` will decrease by 1 each time around the loop.

⚠ *Warning:*
The `start` and `stop` values above may not be *exactly* right. Debug this to make sure you are meeting all the *Goals*.

- Observing your `print()` statements on the console will help.

```
12      print(count)
13      leds.user_num(count, True)
14      sleep(0.1)
15      leds.user_num(count, False)
```

Hint:

- Counting ***Down*** Tips:
  - Start at `6`
  - Stop at `-1` → because `range()` does *not* include the `stop` value.
  - Step by `-1`

Goals:

- Add a second `for` loop to your program.
- Sweep the LED from right (LED-0) to left (LED-7) and then from **left to right**.
- Display the running `count` on the **console** from 0 to 7 and then back to 0.
  - Just 🔧print the numbers... no other text or spaces.
  - 7 should be printed *once only* as you sweep from left to right.

Tools Found: Loops, Ranges, Default function parameters, Print Function

Solution:

```
1   from botcore import *
2   from time import sleep
3
4   # Iterate the beat, with a for loop!
5   for count in range(8):
6       print(count)
7       leds.user_num(count, True)
8       sleep(0.1)
9       leds.user_num(count, False)
10
11  for count in range(6, -1, -1):
12      print(count)
13      leds.user_num(count, True)
14      sleep(0.1)
15      leds.user_num(count, False)
16
```

Objective 6 - Funky Functions

You need to add some movement to that flashy wave!

That means controlling the 🔧motors.

For this dance the *LED sweeps* will pace your movement.

Your dance 🔧algorithm is:

1. Wait for BTN-0 to be pressed
2. Start moving
3. Sweep the LEDs
4. Change movement
5. Sweep the LEDs
6. Change movement
7. Sweep the LEDs
8. …and so on!

You could just copy your "sweep" code over and over, in between motor commands…

- But that would add a lot of redundant code. The solution?

You can package your code into 🔧functions!

- Dive into your ***Toolbox*** to learn more, then you'll be ready to complete this Objective!

Text Editing Tip: When you're moving code around you may want to use the 🔧Editor Shortcuts.

First step: Re-write your existing "sweep" code.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  # Define functions to sweep left and right
5  def sweep_left():
```

Define a function to *sweep left:*

- `def` means "define function".
- Function names follow the same rules as **variables.**
- The **parameters** are inside parenthesis.
  - You have *no* parameters, but still gotta have parenthesis!
- End the line with a *colon* - which is always followed by an **indented** block of code.

```
6      for count in range(8):
7          print(count)
8          leds.user_num(count, True)
9          sleep(0.1)
10         leds.user_num(count, False)
11
```

This is your first loop from the last Objective.

- It must be *indented* beneath the function `def`.
- Standard Python indentation style is 4-characters *(use the TAB key!)*
- Notice the helpful *vertical lines* showing **indentation levels**.
  - *Indentation must be neat and consistent!*

Tip: Select a block of code and press SHIFT-TAB to indent it.

```
12
13 def sweep_right():
```

Define a function to *sweep right:*

- Just like you did above...

- **Note:** When you *define* a function, the code it contains doesn't actually *run* until you ***call*** it later.

```
14      for count in range(6, -1, -1):
15          print(count)
16          leds.user_num(count, True)
17          sleep(0.1)
18          leds.user_num(count, False)
19
20  # Call the functions
21  sweep_left()
22  sweep_right()
```

Finally ***call*** the functions you defined above.

- Notice even though you have no *arguments* to pass to these functions, you must still use *parenthesis*.
  - That's how Python knows it's a *function call*.

```
23
24
```

Goals:

- Define a `sweep_left()` function.
- Define a `sweep_right()` function.
- Call your new `sweep_left()` function.
- Call the `sweep_right()` function after the `sweep_left()` function.
- Verify your LEDs sweep from ***right*** → ***left*** → ***right*** just like before.

Tools Found: Motors, Algorithm, Functions, Editor Shortcuts, Divide and Conquer

Solution:

```
1   from botcore import *
2   from time import sleep
3
4   def sweep_left():
5       # Iterate the beat, with a for loop!
6       for count in range(8):
7           print(count)
8           leds.user_num(count, True)
9           sleep(0.1)
10          leds.user_num(count, False)
11
12  def sweep_right():
13      for count in range(6, -1, -1):
14          print(count)
15          leds.user_num(count, True)
16          sleep(0.1)
17          leds.user_num(count, False)
18
19  sweep_left()
20  sweep_right()
```

Quiz 2 - Extending your range

Question 1: What is printed by the following code?

```
for x in range(4):
    print(x, end=',')
```

✓ 0,1,2,3,

❌ 1,2,3,4,

❌ 0,1,2,3,4,

Question 2: What is printed by the following code?

```
for x in range(10, 1, -1):
    print(x, end=',')
```

✔ 10,9,8,7,6,5,4,3,2,

❌ 10,9,8,7,6,5,4,3,2,1,

❌ 9,8,7,6,5,4,3,2,1,

Question 3: What does the following program do?

```
from botcore import *
from time import sleep

def spin(speed, duration):
    motors.run(LEFT, speed)
    motors.run(RIGHT, -speed)
    sleep(duration)

motors.enable(True)
spin(50, 5)
```

✔ Spins CodeBot clockwise at 50% power for 5 seconds

❌ Spins CodeBot counter-clockwise at 50% power for 5 seconds

❌ *Nothing.* 🔧Motors are not enabled when 🔧function is called.

Objective 7 **- Just Waiting for a Button**

Okay, just one more thing before you rev up those dancin' motors…

You're supposed to wait for someone to press button **BTN-0** before moving.

- Your code will need to loop while checking **BTN-0**.
- Display your ***flashy wave*** while you wait!

Check out the 🔧CodeBot buttons to see how your code can check the state of BTN-0 and BTN-1.

Use a 🔧loop to wait while checking for a button press.

- Keep looping as long as BTN-0 was 🔧not pressed.
- Inside the loop, call your `sweep_left()` and `sweep_right()` functions.

For now, just let the program END when BTN-0 is pressed!

Tip: You can click *BTN-0* on **CodeBot** in the 3D window...

- But **Keyboard 0** on your PC will *also* activate BTN-0 after you click in the 3D view.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  def sweep_left():
5      for count in range(8):
6          print(count)
7          leds.user_num(count, True)
8          sleep(0.1)
```

```
9           leds.user_num(count, False)
10
11  def sweep_right():
12      for count in range(6, -1, -1):
13          print(count)
14          leds.user_num(count, True)
15          sleep(0.1)
16          leds.user_num(count, False)
17
18  while not buttons.was_pressed(0):
19      sweep_left()
20      sweep_right()
```

Your new `while` loop.

- Be sure to **indent** the function calls so they're both inside the loop.

```
21
```

Hint:

- **Click BTN-0 in the 3D view to simulate a button press.**
  - You can also press the **0** or **1** keys on your keyboard to press your 'bots buttons!

Goal:

- Continuously sweep LEDs left and right until BTN-0 is pressed using the `not` operator.

Tools Found: Buttons, Loops, Logical Operators

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  def sweep_left():
5      # Iterate the beat, with a for loop!
6      for count in range(8):
7          print(count)
8          leds.user_num(count, True)
9          sleep(0.1)
10         leds.user_num(count, False)
11
12 def sweep_right():
13     for count in range(6, -1, -1):
14         print(count)
15         leds.user_num(count, True)
16         sleep(0.1)
17         leds.user_num(count, False)
18
19 while not buttons.was_pressed(0):
20     sweep_left()
21     sweep_right()
```

Objective 8 - Beautiful Moves!

Add some *movement* to your dance

You already know how to control the 🔧motors.

- Previously you used `sleep()` for 🔧pacing to control how long the motors go at a certain speed.
- Now instead of just sleeping, you can sweep your LEDs while moving.
  - After all, dancing is just moving *with style!*

It's dance competition time!

Make your best dance moves after the button is pressed!

You'll need to touch some balloons as you sweep across the stage.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  def sweep_left():
5      for count in range(8):
6          print(count)
7          leds.user_num(count, True)
8          sleep(0.1)
9          leds.user_num(count, False)
10
11 def sweep_right():
12     for count in range(7, -1, -1):
13         print(count)
14         leds.user_num(count, True)
15         sleep(0.1)
16         leds.user_num(count, False)
17
18 while not buttons.was_pressed(0):
19     sweep_left()
20     sweep_right()
21
22 # Motor up!
23 motors.enable(True)
```

Dancers, *start your engines!*

- No movement yet... but your **motors** are enabled!

```
24
25 # Function to move motors
26 def go(left, right):
27     motors.run(LEFT, left)
28     motors.run(RIGHT, right)
```

A *convenience* function

- You could call `motors.run()` over and over again as you dance...
- But this will save you some typing!

```
29
30 # Function to sweep left-right N times
31 def sweep(num):
32     for i in range(num):
33         sweep_left()
34         sweep_right()
```

Another helpful function to *sweep* while you move

- You will be using `sweep_left()` and `sweep_right()` to time your moves.
- Just tell this function how many *sweep cycles* ya want!

```
35
36 # Turn a bit to the left
37 go(-10,10)
38 sweep(1)
```

Feel free to experiment here.

- This is **your** dance after all!

- *(but I found that turning a bit helped set my 'bot up for an elegant arc)*

```
39
40  # Make a big clockwise circle
41  go(??, ??)
42  sweep(8)
```

Choose your *speeds*

You'll probably need to move pretty fast to get across the stage!

```
43
44  # Pirouette for style points
45  go(-80, 80)
46  sweep(2)
47
48  # More movement needed?
49
```

Your move!

```
50
```

Hints:

- **Test a Lot!**

  Each time you test this code you will:

  1. Hit the **RESET** button on the *Scene Toolbar,*
  2. Press ▶ **RUN**,
  3. Press **BTN-0** on your CodeBot. (*You can click in the 3D window and use keyboard '0' to do this*)

- **Take it *one move* at a time**

  First, try just making ***one*** turn.

  - Did you get pointed in a good direction?
  - Adjust your code and *test again!*
    - Once you're happy with the first move, go to the next!

Goals:

- Touch at least ***5 balloons*** as you sashay around the stage!
- Dance across the stage to meet your *Balloon* goal within 30 seconds
- Keep that ***LED Wave*** going!

Tools Found: Motors, Time Module

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  def sweep_left():
5      # Iterate the beat, with a for loop!
6      for count in range(8):
7          print(count)
8          leds.user_num(count, True)
9          sleep(0.1)
```

```
10          leds.user_num(count, False)
11
12  def sweep_right():
13      for count in range(7, -1, -1):
14          print(count)
15          leds.user_num(count, True)
16          sleep(0.1)
17          leds.user_num(count, False)
18
19  while not buttons.was_pressed(0):
20      sweep_left()
21      sweep_right()
22
23  # Motor up!
24  motors.enable(True)
25
26  # Function to move motors
27  def go(left, right):
28      motors.run(LEFT, left)
29      motors.run(RIGHT, right)
30
31  # Function to sweep left-right N times
32  def sweep(num):
33      for i in range(num):
34          sweep_left()
35          sweep_right()
36
37  # Turn a bit to the left
38  go(-10,10)
39  sweep(1)
40
41  # Make a big clockwise circle
42  go(100, 80)
43  sweep(8)
44
45  # Pirouette for style points
46  go(-80, 80)
47  sweep(2)
48
```

Mission 6 **- Robot Metronome**

Write code to make a time-keeping Python Maestro!

Objective 1 **- Flash! Ah-ahh!**

Create a new file!

- Use the File→ New File menu to create a new file called "metronome.py"

The first step in creating your metronome:

Flash the **User LEDs** on and off at a specific rate!

- In musical terms, this rate is called the *"tempo"*.
- Music tempo is given in **BPM**, which stands for **Beats per Minute**.
- So if you flash the LEDs once every second, that's 60 BPM *baaaby.*

*Wait – what does **"flash"** mean anyway? How **long** should the LEDs stay on?*

- *Long* enough to be visible, but *short* enough to punctuate the beat.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  # Flash the red User LEDs
5  leds.user(0b11111111)
6  # Pause long enough to see the beat
7  sleep(0.1)
```

```
8  # Turn off lights
9  leds.user(0)
```

Hint:

- Use `sleep(0.1)` to delay for one tenth of a second.

Goal:

- Turn all the **User LEDs** on for **a tenth of a second**, then back off.

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # Flash the red User LEDs
5  leds.user(0b11111111)
6  # Pause long enough to see the beat
7  sleep(0.1)
8  # Turn off lights
9  leds.user(0)
```

Objective 2 - Metro Beat

Now it's time to add some sound to your metronome.

- Use the CodeBot 🔧speaker to play a *pitch* during your LED flash.
  - Remember to turn the `spkr.off()` when you turn the LEDs off!

What's the frequency, Kenneth?

- Well, I think a G above middle C would be nice. **That's 784 Hz.**

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  # Turn on the red user LEDs
5  leds.user(0b11111111)
6
7  # TODO: Turn on the speaker to hear a tone
```

> Turn the speaker on to a **freq** of your choice. Use the `spkr.pitch(freq)` function!

```
8
9  # Pause long enough to see/hear the beat
10 sleep(0.1)
11
12 # TODO: Turn off the speaker
```

> Don't forget to turn the speaker off. It's as simple as `spkr.off()`!

```
13
14 # Turn off the user LEDs
15 leds.user(0)
```

Goal:

- Add a **beep** to your metronome flash code.

Tools Found: Speaker

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # Flash the red User LEDs
5  leds.user(0b11111111)
6  # Beep on the Beat!
7  spkr.pitch(784)
8  # Pause long enough to see/hear the beat
9  sleep(0.1)
10 # Turn off sound and lights
11 spkr.off()
12 leds.user(0)
13
14 sleep(0.1)
15
16 # TODO: Metronome needs to repeat the beat! ...indefinitely!
```

Objective 3 - Loop the Beat

So now you can mark the beat with lights and sound.

- *But what about the tempo?*
  - You need to repeat the beat at exactly the right 🔧interval to achieve the desired **BPM**.

If you're going for 60 BPM that means you repeat every 1 second.

You can use an **infinite** 🔧loop!

You need to move your **flash/beep** code inside a loop.

In Python the usual way to code an *infinite loop* is:

```
while True:
    # beep and flash
    # pause to maintain desired tempo
```

Note two important things here:

1. There is a colon (`:`) at the end of the line with `while`. That means a new block of code begins on the next line.
2. The beep/flash/pause code is 🔧indented on the next lines following the `while True:` Indentation is how you tell Python what belongs inside the 🔧loop.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  while True:
```

Your *infinite loop*

Everything *indented* beneath this loop will repeat forever.

```
5      # Flash the red User LEDs
6      leds.user(0b11111111)
7      # Beep on the Beat!
8      spkr.pitch(784)
9      # Pause long enough to see/hear the beat
10     sleep(0.1)
11     # Turn off sound and lights
12     spkr.off()
13     leds.user(0)
```

Indent your beep/flash code so it's inside the `while` loop.

- *Select* the block of code and press SHIFT-TAB to do this easily!

```
14
15      # Pause to maintain the tempo
16      sleep(???)  # 60 BPM
17
```

Add a delay to set the *tempo*

- How many *seconds* to delay?
- What's 1/60 of a minute?

*Make sure your `sleep()` is also **indented** inside the loop!*

```
18
```

Hint:

- If you're having trouble, try *stepping through the code* using the debugger.

Goal:

- Move your code inside an infinite loop, so it runs forever at 60 BPM.
  - Use the 🔧editor shortcuts to make this easier!

Tools Found: Time Module, Loops, Indentation, Editor Shortcuts

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  while True:
5      # Flash the red User LEDs
6      leds.user(0b11111111)
7      # Beep on the Beat!
8      spkr.pitch(784)
9      # Pause long enough to see/hear the beat
10     sleep(0.1)
11     # Turn off sound and lights
12     spkr.off()
13     leds.user(0)
14
15     # Pause to maintain the tempo
16     sleep(1.0)  # 60 BPM
```

Objective 4 - Tighten up the Tempo

This project is starting to come together!

You have a pretty good metronome already… but there are some problems:

1. The tempo is not exactly 60 bpm.

   The "flash" time adds 0.1 second delay, so the tempo is really: $\frac{60 \cancel{sec}}{1 min} \cdot \frac{1 beat}{1.1 \cancel{sec}}$

   Tempo = $\frac{60}{(1.1)} = 54.5 bpm$

2. Each time you want to change the tempo, you must calculate a new delay and modify the code.
   - You're going to add the capability to adjust the tempo, so this needs to be automatic!

There are two numbers in your code that a user might want to vary: `tempo`, and `beat_duration`.

- Instead of putting those *literal* numbers throughout your program, you should make them 🔧variables.

Then you can calculate the "pause" time after each beat: `pause = 60 / tempo - beat_duration`

> **NOTE:** you can use parenthesis to make it clearer that the division happens before subtraction:
> `pause = (60 / tempo) - beat_duration`
>
> - But actually it works properly as shown due to the 🔧precedence rules.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 60
5  beat_duration = 0.1
6
7  while True:
8      # Flash the red User LEDs
9      leds.user(0b11111111)
10     # Beep on the Beat!
11     spkr.pitch(784)
12     # Pause long enough to see/hear the beat
13     sleep(beat_duration)
14     # Turn off sound and lights
15     spkr.off()
16     leds.user(0)
17
18     # Pause to maintain the tempo
19     pause = 60 / tempo - beat_duration
20     sleep(pause)
```

Hint:

- If you're on a slow computer, smaller `sleep()` delays can be inaccurate.
    - This might prevent you from achieving the accurate BPM needed to pass this objective!
    - Try increasing the `beat_duration` to `0.2` seconds if you think that might be happening.

Goals:

- Add a **variable** called `tempo`
- Add a **variable** called `beat_duration`
- Adjust the "pause" between beats so it accurately accounts for the `beat_duration`.
    - I'll watch it over a 5-second interval to be sure!

Tools Found: Variables, Math Operators

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 60
5  beat_duration = 0.1
6
7  while True:
8      # Flash the red User LEDs
9      leds.user(0b11111111)
10     # Beep on the Beat!
11     spkr.pitch(784)
12     # Pause long enough to see/hear the beat
13     sleep(beat_duration)
14     # Turn off sound and lights
15     spkr.off()
16     leds.user(0)
17
18     # Pause to maintain the tempo
```

```
19      pause = 60 / tempo - beat_duration
20      sleep(pause)
```

***Quiz 1* - Variables**

Question 1: Which two of the following are valid variable names?

❌ 1beep

✅ beepTime

✅ delay_beep

❌ beep$_duration

Question 2: The following “blink” code is broken! The LEDs never turn off. Why?

```
while True:
    leds.user(0b11111111)
    sleep(0.1)
leds.user(0)
sleep(1.0)
```

✅ The code to turn LEDs off and delay is not indented, so it’s outside the infinite loop and it never runs.

❌ The `sleep(1.0)` when LEDs are off is not long enough.

❌ The `leds.user(0)` function argument should be `0b00000000` instead.

Question 3: What is the value of `delay` after the following statement: `delay = 99 + 1 / 10`?

✅ 99.1

❌ 99

❌ 100

❌ 10

❌ 10.0

***Objective 5* - Sound Control**

Now it’s time to start adding user controls to your metronome.

The first controllable feature will be to toggle the sound ON and OFF: a **"mute button"**

- Each time the button is pressed, the *state* of your program changes:
- `sound_on = True` ⟶ `sound_on = False`

Your program has 🔧state? Yes! Stop your program at any point in time and what does it know?

- The `tempo`
- The `beat_duration`

  Every moment your loop is running, it has those 🔧variables in memory.

Enable Sound: *True or False?*

Before you worry about detecting a button press, ask yourself:

Q. What **state** do I want the button to change?

A. The *"Sound is Enabled"* state!

So before you hook up the button, ***add a "sound is enabled" state*** to your code.

- Initialize the variable to a 🔧Boolean `True` like so: `sound_on = True`
- Now you can use the 🔧variable to control whether or not the sound actually plays!
- To do that, you'll need a 🔧control flow statement.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 60
5  beat_duration = 0.1
6  sound_on = True
```

Your new *state* variable.

- Initialize to `True` so the sound is ON by default.

```
7
8  while True:
9      # Flash the red User LEDs
10     leds.user(0b11111111)
11
12     # Beep on the Beat!
13     if sound_on:
14         spkr.pitch(784)
```

Just a simple `if` statement is needed.

- Now the sound is controlled by a **variable!**

```
15
16     # Pause long enough to see/hear the beat
17     sleep(beat_duration)
18     # Turn off sound and lights
19     spkr.off()
20     leds.user(0)
21
22     # Pause to maintain the tempo
23     pause = 60 / tempo - beat_duration
24     sleep(pause)
```

Hint:

- Try initializing `sound_on = False` to see if this actually works!
  - Your metronome should run silently in this case.

Goals:

- Add a 🔧variable called `sound_on` above your while loop.
- Use the `sound_on` variable in an `if` statement to control the speaker.

Tools Found: State, Variables, bool, Branching

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 60
5  beat_duration = 0.1
6  sound_on = True
```

```
7
8  while True:
9      # Flash the red User LEDs
10     leds.user(0b11111111)
11
12     # Beep on the Beat!
13     if sound_on:
14         spkr.pitch(784)
15
16     # Pause long enough to see/hear the beat
17     sleep(beat_duration)
18     # Turn off sound and lights
19     spkr.off()
20     leds.user(0)
21
22     # Pause to maintain the tempo
23     pause = 60 / tempo - beat_duration
24     sleep(pause)
```

Objective 6 - Mute Button

The API for CodeBot buttons consists of just two functions:

1. `buttons.was_pressed(n)`
2. `buttons.is_pressed(n)`

See the CodeBot buttons tool to explore the differences between these two functions.

- Also notice that they both return a True or False bool value.
  - Be sure to read about this data type.

Add another `if` statement

You already have code that controls the speaker based on `sound_on`.

- Now, add a *new* `if` statement that checks for a button press!

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 60
5  beat_duration = 0.1
6  sound_on = True
7
8  while True:
9      # Flash the red User LEDs
10     leds.user(0b11111111)
11
12     # Check "mute" button
13     if buttons.was_pressed(0):
14         sound_on = False
```

This is a new `if` statement.

- Use it to control the `sound_on` state when a button is pressed.

```
15
16     # Beep on the Beat!
17     if sound_on:
18         spkr.pitch(784)
19
20     # Pause long enough to see/hear the beat
21     sleep(beat_duration)
22     # Turn off sound and lights
23     spkr.off()
24     leds.user(0)
25
26     # Pause to maintain the tempo
```

```
27      pause = 60 / tempo - beat_duration
28      sleep(pause)
```

Hints:

- You can directly use a **bool** as the `if` statement expression: `if buttons.was_pressed(0):`
- It's okay if you can't yet re-enable the sound.
  - Just stop and restart your program to test it again for now…

Goal:

- Use the `buttons.was_pressed(0)` function to set `sound_on = False` if **BTN-0** was pressed.
  - **Demonstrate by pressing BTN-0 after 5 beeps.**

Tools Found: Buttons, bool, Data Types, Speaker

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 60
5  beat_duration = 0.1
6  sound_on = True
7
8  while True:
9      # Flash the red User LEDs
10     leds.user(0b11111111)
11
12     # Check "mute" button
13     if buttons.was_pressed(0):
14         sound_on = False
15
16     # Beep on the Beat!
17     if sound_on:
18         spkr.pitch(784)
19
20     # Pause long enough to see/hear the beat
21     sleep(beat_duration)
22     # Turn off sound and lights
23     spkr.off()
24     leds.user(0)
25
26     # Pause to maintain the tempo
27     pause = 60 / tempo - beat_duration
28     sleep(pause)
```

Objective 7 - Un-Mute

Okay, it would be nice if BTN-0 could turn the sound back ON when pressed again.

- Can you make it **toggle** ON/OFF like a light switch?
- "If it's OFF turn it ON. If it's ON turn it OFF."

When the button is pressed, you need to *flip* the `True`/`False` value of `sound_on`.

- Python has a 🔧logical operator made just for that purpose: the 🔧not operator.
  - It converts `True` to `False`, and vice versa.
  - It is a 🔧unary operator that goes in front of a 🔧Boolean expression, similar to a (-) sign for numeric expressions.

So, when `sound_on` is `True` → `not sound_on` will be `False`!

And *(this might blow your mind):*

- You can not the *current* value, and store it right back in the same variable like this:

```
sound_on = not sound_on
```

In the above statement the key thing to realize is:

- *The right-hand side of the assignment statement executes first*
  - ...before the resulting value is assigned to the variable on the *left-hand side.*

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 60
5  beat_duration = 0.1
6  sound_on = True
7
8  while True:
9      # Flash the red User LEDs
10     leds.user(0b11111111)
11
12     # Check "mute" button
13     if buttons.was_pressed(0):
14         # Toggle sound ON/OFF
15         sound_on = ???
```

Toggle the `sound_on` variable.

- Don't just set it to `False`...
- Instead set it to `not` what it was before!

```
16
17     # Light PWR LED when muted
18     leds.pwr(???)
```

The PWR LED turns ON when you call `leds.pwr(True)`

- This is the *"Muted"* indicator light.
- So if you wrote `leds.pwr(sound_on)` then it would be the *opposite* of what you want.

Is there an operator you can insert before `sound_on` that will flip the value?

```
19
20     # Beep on the Beat!
21     if sound_on:
22         spkr.pitch(784)
23
24     # Pause long enough to see/hear the beat
25     sleep(beat_duration)
26     # Turn off sound and lights
27     spkr.off()
28     leds.user(0)
29
30     # Pause to maintain the tempo
31     pause = 60 / tempo - beat_duration
32     sleep(pause)
```

Hints:

- Use your logical `not` skills to toggle the value when a button is pressed:

```
# Toggle a boolean value
value = not value
```

- For the PWR LED, you don't need to *toggle* a variable.
  - Just use `not` to flip the value before it's passed to `leds.pwr()`

Goals:

- Change your `sound_on = False` code when the button-press is detected to **toggle** the current `sound_on` value instead.
- Turn on the PWR LED whenever the sound is muted.
    - **Demonstrate by pressing BTN-0 after 5 beeps.**

Tools Found: Logical Operators, Unary and Binary Operators, bool, Variables

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 60
5  beat_duration = 0.1
6  sound_on = True
7
8  while True:
9      # Flash the red User LEDs
10     leds.user(0b11111111)
11
12     # Check "mute" button
13     if buttons.was_pressed(0):
14         # Toggle sound ON/OFF
15         sound_on = not sound_on
16
17     # Light PWR LED when muted
18     leds.pwr(not sound_on)
19
20     # Beep on the Beat!
21     if sound_on:
22         spkr.pitch(784)
23
24     # Pause long enough to see/hear the beat
25     sleep(beat_duration)
26     # Turn off sound and lights
27     spkr.off()
28     leds.user(0)
29
30     # Pause to maintain the tempo
31     pause = 60 / tempo - beat_duration
32     sleep(pause)
```

Objective 8 - Tempo List

It's time to turn your attention to the final feature of this Mission:

- Use **BTN-1** to *change the tempo* and *show* the current selection (0 - 4) on the **LS LEDs.**

That means you will need another 🔧variable to track which of the 5 tempos is selected.

- A variable called `tempo_select` that ranges from 0 - 4 would be perfect for this.
- Use the 🔧bit-shift operator `<<` to light the LS LED based on the selection.

What 5 tempos do you need?

- A quick call to the Band Director revealed the following:
    - Largo (50bpm), Adagio (70bpm), Andante (100bpm), Allegro (140bpm), Presto (180bpm)
- *So that's your list of tempos!*

*How can you code a **list** of things in Python?*

Python's 🔧list data type!

Just put your list of tempos in *square brackets* like so:

```
tempo_list = [50, 70, 100, 140, 180]
```

Check out the 🔧list tool to see how to access the items in the list.

Your `tempo_select` variable will be the *index* into this list, pointing to the currently selected tempo.

- Since lists start with index `0`, make *that* the initial value of `tempo_select`.
- So now the `tempo` variable is set by reading the selected value from the list, like so:
  - `tempo = tempo_list[tempo_select]`

At this point you haven't hooked up the *button press* to select the tempo yet, but your program has all the state you'll need to make it happen.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo_list = [50, 70, 100, 140, 180]
```

Your new Python **list**!

- Lists can contain items of any *data type*
- You just need 5 *integers* for your `tempo_list`

```
5  tempo_select = 0
```

A new variable to track *which* item in `tempo_list` is selected.

- Later you can use this as the [*index*] into your list.

```
6  tempo = 60
7  beat_duration = 0.1
8  sound_on = True
9
10 while True:
11     # Flash the red User LEDs
12     leds.user(0b11111111)
13
14     # Check "mute" button
15     if buttons.was_pressed(0):
16         # Toggle sound ON/OFF
17         sound_on = not sound_on
18
19     # Light PWR LED when muted
20     leds.pwr(not sound_on)
21
22     # Beep on the Beat!
23     if sound_on:
24         spkr.pitch(784)
25
26     # Pause long enough to see/hear the beat
27     sleep(beat_duration)
28     # Turn off sound and lights
29     spkr.off()
30     leds.user(0)
31
32     # Show current tempo selection on LS LEDs
33     leds.ls(1 << tempo_select)
```

You'll be setting the **Line Sensor LEDs** `leds.ls()` with a *binary* value:

- A value of `0b00001` sets the first LED (LS-0)
- Shifting this *left* by `tempo_select` will light up the LED corresponding to that *tempo*.

```
34
35     # Pause to maintain the tempo
36     tempo = tempo_list[tempo_select]
```

Choose Your Tempo

Use `tempo_select` as an *index* into your **list**.

- Remember, *list* indexing starts with 0
- So your initial value `tempo_list[0]` is `50` bpm

```
37      pause = 60 / tempo - beat_duration
38      sleep(pause)
```

Goals:

- Define a **list** of tempos named `tempo_list`
- Define a variable `tempo_select`
  - This is your list *index*, initially set to `0`
  - ...the 0th item is the *first* one in your list!
- Light the LS LED corresponding to the currently selected `tempo_select`

Tools Found: Variables, Bitwise Operators, list, State

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo_list = [40, 70, 100, 140, 180]
5  tempo_select = 0
6  tempo = 60
7  beat_duration = 0.1
8  sound_on = True
9
10 while True:
11     # Flash the red User LEDs
12     leds.user(0b11111111)
13
14     # Check "mute" button
15     if buttons.was_pressed(0):
16         # Toggle sound ON/OFF
17         sound_on = not sound_on
18
19     # Light PWR LED when muted
20     leds.pwr(not sound_on)
21
22     # Beep on the Beat!
23     if sound_on:
24         spkr.pitch(784)
25
26     # Pause long enough to see/hear the beat
27     sleep(beat_duration)
28     # Turn off sound and lights
29     spkr.off()
30     leds.user(0)
31
32     # Show current tempo selection on LS LEDs
33     leds.ls(1 << tempo_select)
34
35     # Pause to maintain the tempo
36     tempo = tempo_list[tempo_select]
37     pause = 60 / tempo - beat_duration
38     sleep(pause)
```

Quiz 2 **- Bitwise shift, not, and lists**

Question 1: What's the value of `tempo` after the following bit-shift statement?

```
tempo = 1 << 3
```

✓ `0b01000`

✗ `0b00100`

✗ `0b10000`

Question 2: What's the value of `sound_on` after the following code runs?

```
sound_on = True
sound_on = not sound_on
```

✓ `False`

✗ `True`

✗ `not`

Question 3: What is the value of `tempo_list[1]` ?

```
tempo_list = [50, 70, 100, 140, 180]
```

✓ 70

✗ 50

✗ 100

✗ 140

✗ 180

Objective 9 - Tempo Select

Now to hook BTN-1 up to your `tempo_select` index variable

If **BTN-1** was pressed, change to the next tempo.

- You can do that by adding `1` to the `tempo_select` 🔧variable.

```
tempo_select = tempo_select + 1
```

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo_list = [40, 70, 100, 140, 180]
5  tempo_select = 0
6  tempo = 60
7  beat_duration = 0.1
8  sound_on = True
9
10 while True:
11     # Flash the red User LEDs
12     leds.user(0b11111111)
13
14     # Check "mute" button
15     if buttons.was_pressed(0):
16         # Toggle sound ON/OFF
17         sound_on = not sound_on
18
19     # Light PWR LED when muted
20     leds.pwr(not sound_on)
21
22     # Beep on the Beat!
23     if sound_on:
24         spkr.pitch(784)
```

```
25
26      # Pause long enough to see/hear the beat
27      sleep(beat_duration)
28      # Turn off sound and lights
29      spkr.off()
30      leds.user(0)
31
32      # Select Tempo
33      if buttons.was_pressed(1):
34          tempo_select = tempo_select + 1
```

Increment `tempo_select`

That means *add 1 to it.*

- Add one to the current value, and assign the result *back* to `tempo_select`

```
35
36      # Show current tempo selection on LS LEDs
37      leds.ls(1 << tempo_select)
38
39      # Pause to maintain the tempo
40      tempo = tempo_list[tempo_select]
41      pause = 60 / tempo - beat_duration
42      sleep(pause)
```

Hints:

- You will encounter an **Error**! Do not despair...
- Step through your code and watch the `tempo_select` count up to **oblivion!**

Goal:

- When **BTN-1** was pressed, increase the `tempo` to the next value in `tempo_list`.
    - You should see the LS LED for each index lighting up as the tempo increases!

Tools Found: Variables

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo_list = [40, 70, 100, 140, 180]
5  tempo_select = 0
6  tempo = 60
7  beat_duration = 0.1
8  sound_on = True
9
10 while True:
11     # Flash the red User LEDs
12     leds.user(0b11111111)
13
14     # Check "mute" button
15     if buttons.was_pressed(0):
16         # Toggle sound ON/OFF
17         sound_on = not sound_on
18
19     # Light PWR LED when muted
20     leds.pwr(not sound_on)
21
22     # Beep on the Beat!
23     if sound_on:
24         spkr.pitch(784)
25
26     # Pause long enough to see/hear the beat
```

```
27      sleep(beat_duration)
28      # Turn off sound and lights
29      spkr.off()
30      leds.user(0)
31
32      # Select Tempo
33      if buttons.was_pressed(1):
34          tempo_select = tempo_select + 1
35
36      # Show current tempo selection on LS LEDs
37      leds.ls(1 << tempo_select)
38
39      # Pause to maintain the tempo
40      tempo = tempo_list[tempo_select]
41      pause = 60 / tempo - beat_duration
42      sleep(pause)
```

Objective 10 **- Wrapping the Metronome**

Okay, time to fix that Bug!

When your `tempo_select` goes past the end of your list, you should set it back to `0`.

An `if` statement is a nice way to do this:

```
if tempo_select > 4:
    tempo_select = 0
```

But wait!

- That `4` is a ***magic number*** in your code, and that kind of magic always leads to trouble.
  - For instance if later you add a couple more tempo values to your list, you might forget to change the number.
- The solution is to use Python's 🔧built-in function `len()` which will always give you the exact number of items in the list!

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo_list = [40, 70, 100, 140, 180]
5  tempo_select = 0
6  tempo = 60
7  beat_duration = 0.1
8  sound_on = True
9
10 while True:
11     # Flash the red User LEDs
12     leds.user(0b11111111)
13
14     # Check "mute" button
15     if buttons.was_pressed(0):
16         # Toggle sound ON/OFF
17         sound_on = not sound_on
18
19     # Light PWR LED when muted
20     leds.pwr(not sound_on)
21
22     # Beep on the Beat!
23     if sound_on:
24         spkr.pitch(784)
25
26     # Pause long enough to see/hear the beat
27     sleep(beat_duration)
28     # Turn off sound and lights
29     spkr.off()
30     leds.user(0)
31
32     # Select Tempo
33     if buttons.was_pressed(1):
34         tempo_select = tempo_select + 1
35         # Wrap around to zero at end of list
36         if tempo_select == len(tempo_list):
```

```
37              tempo_select = 0
```

The **len**gth of the list is `5`.

- The items are *indexed* 0 - 4
- ...so when you reach 5 it's time to *wrap!*

```
38
39      # Show current tempo selection on LS LEDs
40      leds.ls(1 << tempo_select)
41
42      # Pause to maintain the tempo
43      tempo = tempo_list[tempo_select]
44      pause = 60 / tempo - beat_duration
45      sleep(pause)
```

Hint:

- Lists are indexed starting with *zero*
    - So `tempo_list[5]` is *past the end* since your list only contains 5 items!

Goals:

- Use the `len()` function to detect when the **BTN-1** selection has incremented past the end of the `tempo_list`.
- Add code so your tempo selection wraps around to the beginning of the list when you go past the end.
    - *Prove it* by pressing **BTN-1** until it wraps back to `0`

Tools Found: Built-In Functions

Solution:

```
1   from botcore import *
2   from time import sleep
3
4   tempo_list = [40, 70, 100, 140, 180]
5   tempo_select = 0
6   tempo = 60
7   beat_duration = 0.1
8   sound_on = True
9
10  while True:
11      # Flash the red User LEDs
12      leds.user(0b11111111)
13
14      # Check "mute" button
15      if buttons.was_pressed(0):
16          # Toggle sound ON/OFF
17          sound_on = not sound_on
18
19      # Light PWR LED when muted
20      leds.pwr(not sound_on)
21
22      # Beep on the Beat!
23      if sound_on:
24          spkr.pitch(784)
25
26      # Pause long enough to see/hear the beat
27      sleep(beat_duration)
28      # Turn off sound and lights
29      spkr.off()
30      leds.user(0)
31
32      # Select Tempo
33      if buttons.was_pressed(1):
34          tempo_select = tempo_select + 1
35          # Wrap around to zero at end of list
```

```
36          if tempo_select == len(tempo_list):
37              tempo_select = 0
38
39      # Show current tempo selection on LS LEDs
40      leds.ls(1 << tempo_select)
41
42      # Pause to maintain the tempo
43      tempo = tempo_list[tempo_select]
44      pause = 60 / tempo - beat_duration
45      sleep(pause)
```

Mission 7 - Line Sensors

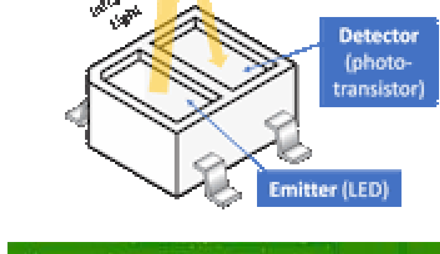
Use the line sensors to navigate your robot. It's time for autonomous robotics!

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.

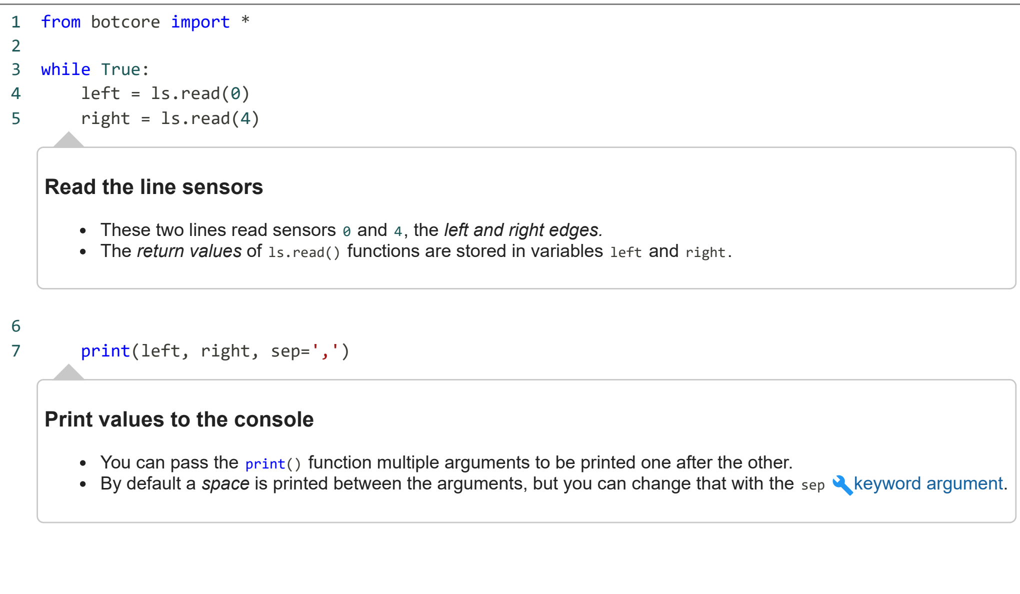
Your code can **read** the *brightness* level of the reflected *infrared* light as an analog value with the function:

```
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 and name it "line_sense.py".

CodeTrek:



```
1  from botcore import *
2
3  while True:
4      left = ls.read(0)
5      right = ls.read(4)
```

Read the line sensors

- These two lines read sensors `0` and `4`, the *left and right edges*.
- The *return values* of `ls.read()` functions are stored in variables `left` and `right`.

```
6
7      print(left, right, sep=',')
```

Print values to the console

- You can pass the `print()` function multiple arguments to be printed one after the other.
- By default a *space* is printed between the arguments, but you can change that with the `sep` keyword argument.

```
8
```

Goals:

- Write an infinite loop that *reads* the value of 🔧line sensors `0` and `4` with `ls.read(n)`
- Use the 🔧print function to display sensor values on the console.
  - Values must be formatted like so: `1234,5678`
  - *No spaces* are allowed! (use the `sep` 🔧parameter)

Tools Found: Line Sensors, Analog to Digital Conversion, int, Binary Numbers, Print Function, Parameters, Arguments, and Returns, Keyword and Positional Arguments

Solution:

```
1  from botcore import *
2
3  while True:
4      left = ls.read(0)
5      right = ls.read(4)
6
7      print(left, right, sep=',')
8
```

Quiz 1 - Line Sensor Analog Values

Question 1: The 🔧line sensor functions return an 🔧int value corresponding to the amount of light reflected back from a surface. The value is between 0 and 4095

- What does a *higher* value mean?

✓ Higher numbers mean **less** light is reflected back to sensor.

✗ Higher numbers mean **more** light is reflected back to sensor.

Question 2: The 🔧line sensor readings range from 0 to 4095, which is **4096** levels of reflectivity. What's so special about **4096**?

- Why not use a nice *round* number like 4000?

✓ 4096 *is* a nice round number in 🔧binary!

✗ Light travels in packets of 4096 photons each, not round waves.

✗ The maximum size for an integer in Python is 4096, so the full range is 0 - 4095.

Objective 2 - Sensorial Geographic

Your 'bot is navigating a *compass rose*

Apparently someone has taken a *map* poster down from the *World Geography* classroom.

- The compass rose is laid out in 8 shades of gray, ranging from black to white.

Can you detect which direction CodeBot is facing based on 🔧line sensor readings?

The *ultimate* Mission goal is to allow a user to type in a direction like 'N', 'E', 'SW',... and have CodeBot automatically rotate to face that direction.

First step: *Chart the Territory*

You need to rotate your bot 360° and write down sensor readings.

- Just add some 🔧motor *spin* code before your `while` 🔧loop
- Watch the values 🔧printing on the **console** as you spin.
- **Stop** your program after a full 360° rotation, and *scroll back* through the console to see the values.
  - There will be lots of duplicates. Find the *stable* value for each shade!
  - *Note:* the readings are raw 🔧ADC *"counts"*, just relative light levels with no physical units.

Grab a piece of paper and make a *table* of your results.

- You will have 8 rows in the table, one for each *cardinal* and *intermediate* direction.
- The first 3 rows of my table are below. Your sensor readings may vary!

Direction	Left	Right
W	2517	3043
NW	3043	3569
N	3569	4080
...		

CodeTrek:

```
1  from botcore import *
2
3  # Spin clockwise, slowly
4  motors.enable(True)
5  motors.run(LEFT, 10)
6  motors.run(RIGHT, -10)
```

Just your typical everyday spin-in-place code!

```
7
8  while True:
9      left = ls.read(0)
10     right = ls.read(4)
11     print(left, right, sep=',')
12
```

Goal:

- Add some 🔧motor *spin* code before your `while` 🔧loop

Tools Found: Line Sensors, Motors, Loops, Print Function, Analog to Digital Conversion

Solution:

```
1  from botcore import *
2
3  # Spin clockwise, slowly
4  motors.enable(True)
5  motors.run(LEFT, 10)
6  motors.run(RIGHT, -10)
7
8  while True:
9      left = ls.read(0)
10     right = ls.read(4)
11     print(left, right, sep=',')
12
```

Objective 3 - Go North - v1

Rotating to Face North

Here is ancient coding wisdom:

"Do the simplest thing that could possibly work."

What could be simpler than just using a *single* sensor value to find North?

Add a branching `if` statement to your loop

- Use `break` to exit the loop when your **LEFT** sensor is *approximately* at the expected **North** reading.

Take a look at your *table of sensor readings:*

- What's the Left sensor value at the **N**orth position?
- How *different* are the values from one position to the next?
  - My readings show the *shades of gray* about 500 counts apart.

Based on my values, if the sensor is within 100 counts of the expected **N**orth value it can be considered on-target.

CodeTrek:

```
1  from botcore import *
2
3  # Spin clockwise, slowly
4  motors.enable(True)
5  motors.run(LEFT, 10)
6  motors.run(RIGHT, -10)
7
8  while True:
9      left = ls.read(0)
10     right = ls.read(4)
11     print(left, right, sep=',')
12
13     # At North postion, my left sensor = 3569.
14     # Check for left reading within 100 counts of target.
15     if 3469 < left < 3669:
16         # We're in the neighborhood of North...
17         motors.enable(False)  # Stop the motors!
18         break
```

Python supports *chaining* of comparison operators.

- That means expressions like a < b < c have the interpretation that is conventional in mathematics.

In this case I've manually calculated limits below (-100) and above (+100) my target sensor reading, and I'll `break` out of the loop if the `left` sensor is in that range.

Goal:

- Use `break` to exit the loop when your LEFT sensor is approximately at the expected North reading.

Tools Found: Branching, Loops, Math Operators

Solution:

```
1  from botcore import *
2
3  # Spin clockwise, slowly
4  motors.enable(True)
5  motors.run(LEFT, 10)
6  motors.run(RIGHT, -10)
7
8  while True:
9      left = ls.read(0)
10     right = ls.read(4)
11     print(left, right, sep=',')
12
13     # At North postion, my left sensor = 3569.
14     # Check for left reading within 100 counts of target.
15     if 3469 < left < 3669:
16         motors.enable(False)
```

```
17        break
18
```

Quiz 2 - Break Those Chains

Question 1: What is the purpose of the `break` statement?

✓ It breaks out of the nearest enclosing loop.

✗ It stops the program and exits immediately to the Operating System (OS).

✗ It allows another process to execute momentarily, like a *"commercial break"*.

Question 2: Branching and comparison with `if` statements are very powerful tools. Use your knowledge of these techniques to determine what is printed by the code below:

```
x = 42
y = 100
if  30 < x < 50:
    if y < 100:
        print('ONE')
    else:
        print('TWO')
else:
    print('THREE')
```

✓ TWO

✗ ONE

✗ THREE

Objective 4 - Go North - v2

Split the *Difference*

The *simplest thing that could work* is a good place to start, because it quickly shows you what needs to be improved! Sometimes that's nothing at all... *but not this time!*

- **Your 'bot stops rotating too soon.** In fact it stops before the **right** sensor crosses the *center-line* of North.

Straddle the Line

The first improvement to make in *finding North* is to have CodeBot continue rotating until the **right** sensor *crosses the line* to the next section.

- Each shaded section reads about ***500 counts* different** than its neighbor.
- So it's safe to set a minimum difference of **100 counts** before you consider the right sensor to have crossed the line.
  - That would make a good constant to define in your code!

```
# Minimum difference (counts) to be considered in the next section
MIN_DIFF = 100
```

CodeTrek:

```
1  from botcore import *
2
3  # Sensor readings are around 500 counts apart.
4  # Minimum difference (counts) to be considered in the next section.
5  MIN_DIFF = 100
```

Defining a constant

Often you'll put the constants and global variables near the top of your code.

- That way, all the code below can access them.
- And you can easily make changes if needed later, without having to sort through the code to remember where you defined them!

```
6
7  # Spin clockwise, slowly
8  motors.enable(True)
9  motors.run(LEFT, 10)
10 motors.run(RIGHT, -10)
11
12 while True:
13     left = ls.read(0)
14     right = ls.read(4)
15     print(left, right, sep=',')
16
17     # First make sure sensors are "straddling" different sections.
18     if abs(left - right) > MIN_DIFF:
```

Do the **left** and **right** sensor values *differ* more than `MIN_DIFF`?

- Subtract them to find the *difference*.
- It doesn't matter which is larger, so take the *absolute value* of the result.
  - Python's built-in `abs()` function will make sure it's positive.

```
19         # At North postion, my left sensor = 3569.
20         # Check for left reading within 100 counts of target.
21         if 3469 < left < 3669:
22             motors.enable(False)
23             break
```

Goal:

- Check that the `left` and `right` sensor readings *differ* by more than `MIN_DIFF` before deciding that **North** has been reached.
  - Press **RESET** between each attempt. Your starting position must be *West.*

Tools Found: Constants, Locals and Globals, Built-In Functions

Solution:

```
1  from botcore import *
2
3  # Sensor readings are around 500 counts apart.
4  # Minimum difference (counts) to be considered in the same region.
5  MIN_DIFF = 100
6
7  # Spin clockwise, slowly
8  motors.enable(True)
9  motors.run(LEFT, 10)
10 motors.run(RIGHT, -10)
11
12 while True:
13     left = ls.read(0)
14     right = ls.read(4)
15     print(left, right, sep=',')
16
17     # First make sure sensors are "straddling" different sections.
18     if abs(left - right) > MIN_DIFF:
19         # At North postion, my left sensor = 3569.
20         # Check for left reading within 100 counts of target.
21         if 3469 < left < 3669:
22             motors.enable(False)
23             break
```

Objective 5 - Go North - v3

Next Iteration...

Tighten it up!

There's an easy change to improve your accuracy:

- Currently you're using sensors **LS-0** and **LS-4**.
  - Those are *really* far apart!
  - Instead, use 2 *adjacent* line sensors to *straddle the line*

And one more thing -

- You defined `MIN_DIFF = 100` but there is still a *manual calculation* in your code.
  - That `if 3469 < left < 3669:` hurts your code's readability.
  - Take a look at the CodeTrek for one way to fix that.

CodeTrek:

```
1  from botcore import *
2
3  # Sensor readings are around 500 counts apart.
4  # Define a minimum difference (counts) to be considered in the same region.
5  MIN_DIFF = 100
6
7  # At North postion, my left sensor = 3569.
8  target_left = 3569
```

Define the `target_left` sensor value up here, so it's easy to change if you want to *target* a different direction.

```
9
10 # Spin clockwise, slowly
11 motors.enable(True)
12 motors.run(LEFT, 10)
13 motors.run(RIGHT, -10)
14
15 while True:
16     left = ls.read(1)
17     right = ls.read(2)
```

Tighten it up!

Sensors 1 and 2 are close together.

- Let's see *these* sensors straddle the line!
- ...this should improve your accuracy. *(but it still won't be perfect)*

```
18     print(left, right, sep=',')
19
20     # First make sure sensors are "straddling" different sections.
21     if abs(left - right) > MIN_DIFF:
22         # Check for left reading within MIN_DIFF counts of target.
23         if abs(left - target_left) < MIN_DIFF:
24             motors.enable(False)
25             break
```

Look familiar?

Using `abs()` to get the *magnitude* of a difference in sensor values is a technique worth remembering!

Hint:

- ## Observe and Adjust

  If your bot is not stopping where you want it to, try to observe what's happening.

  - Does it *overshoot* or *undershoot* the target?

- Are you rotating too quickly?
- Should you change which sensors are being used?

Goal:

- Tighten up your code and see if you can get a little closer to *True **North***

Tools Found: Line Sensors, Readability

Solution:

```
from botcore import *

# Sensor readings are around 500 counts apart.
# Define a minimum difference (counts) to be considered in the same region.
MIN_DIFF = 100

# At North postion, my left sensor = 3569.
target_left = 3569

# Spin clockwise, slowly
motors.enable(True)
motors.run(LEFT, 10)
motors.run(RIGHT, -10)

while True:
    left = ls.read(1)
    right = ls.read(2)
    print(left, right, sep=',')

    # First make sure sensors are "straddling" different sections.
    if abs(left - right) > MIN_DIFF:
        # Check for left reading within MIN_DIFF counts of target.
        if abs(left - target_left) < MIN_DIFF:
            motors.enable(False)
            break
```

Objective 6 - Compass Navigator

Interactive Nav-Bot!

Now that you can follow your sensors accurately, it's time to *unleash the power* on all the other *compass* directions!

- You can't let **North** have all the fun ;-)
- The goal is for a *user* to be able to input any *cardinal or intermediate direction:*
  - Clockwise around the compass rose: **N, NE, E, SE, S, SW, W, NW**

Got Data?

Do you still have that scrap of paper with all the sensor readings?

- You only need the *Left* sensor reading for each direction

Index	Direction	Left
0	W	2517
1	NW	3043
2	N	3569
...	...	

- There should be 8 rows in your table (Index 0 - 7).

In Python the table above can be coded as a list of lists:

- This is also known as a *2-dimensional Array* or *Matrix*.

```
sensor_data = [
    ['W',  2517],
```

```
    ['NW', 3043],
    ['N',  3569],
]
```

So `sensor_data[0]` is the first *row* of the table.

- It is itself a list, containing the *name* and *left value*.

```
# Example:
row = sensor_data[0]   # row is ['W', 2517]
name = row[0]          # name is 'W'
value = row[1]         # value is 2517
```

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  # Sensor readings are around 500 counts apart.
5  # Define a minimum difference (counts) to be considered in the same region.
6  MIN_DIFF = 100
7
8  # A matrix of values [ [direction_name, left_sensor_value],... ]
9  sensor_data = [
10     ['N',  3569],
11     ['NE', 4080],
12     ['E',  369],
13     ['SE', ???],
14     ['S',  ???],
15     ['SW', ???],
16     ['W',  ???],
17     ['NW', ???]
18 ]
```

The Matrix!

- This is your *data table* coded as a 2-dimensional *array*.
- That's just a list
  - ...where each *item* is also a list

The outer list is **"columns"** and the inner one is **"rows"**.

- Notice you can write this on multiple lines for readability.
- That also makes the *array* look more like a written table!

```
19
20 # Prompt user for direction
21 target_direction = input("Enter target direction: ")
```

Prompt for input on the Console

- This will return a string
- Save it in a global variable `target_direction`
- The user will have to type an *exact match* for the *name* in your data table.
  - The comparison is *case-sensitive* too!

```
22
23 def find_name(left):
24     """Search for a direction name given a left-side sensor reading"""
25     for d in sensor_data:
26         if abs(left - d[1]) < MIN_DIFF:
27             # Found it! Return the name.
28             return d[0]
```

Define a function `def find_name():` to *search* your data table.

- Loop through each *row.*
- Check if the *sensor value* is close to the row ***value.***
- If it is, return the row ***name.***

```
29
30 # Spin clockwise, slowly
31 motors.enable(True)
32 motors.run(LEFT, 10)
33 motors.run(RIGHT, -10)
34
35 while True:
36     left = ls.read(1)
37     right = ls.read(2)
38     # print(left, right, sep=',')
```

Comment out this 🔧print function.

```
39
40     # First make sure sensors are "straddling" different sections.
41     if abs(left - right) > MIN_DIFF:
42         found = find_name(left)
```

Search your data table!

- You pass it a sensor reading
- ...it gives you the direction *name!*
- See how the function 🔧return value replaces the call?
  - In this case the value is assigned to `found`.

```
43         if found:
44             print(found)
45             if found == target_direction:
46                 break
```

When you reach the ***target***, `break` out of the 🔧loop!

- Note: *Leave the motors enabled here!*

```
47
48 # Charge ahead bravely!
49 motors.run(LEFT, 40)
50 motors.run(RIGHT, 40)
51 sleep(3.0)
```

Charge forward to hit the *water bottle!*

- Remember to `sleep()` for a bit, since the motors stop when your program ends.

```
52
53
```

Hints:

- ## Observe and Adjust

  If your bot is not stopping where you want it to, try to observe what's happening.

  - Does it *overshoot* or *undershoot* the target?
  - Are you rotating too quickly?
  - Should you change which sensors are being used?

- ## Possible Adjustments

  - Try slowing your rotation speed.
  - Use a different pair of 🔧line sensors.

- Use 🔧print statements to better understand what your code is doing.

- ## Function Return Values

  Your `find_name()` function does some work, and 🔧returns a value when you call it.

  - A `return` statement can appear anywhere in a 🔧function.
  - It *ends* the execution of the function, so your program resumes right where the function was called.
  - The *return value* replaces the function call!

Goals:

- Create a 2D Array of `sensor_data` with 8 rows.
  - Each row should contain `[name, left_sensor_value]`
- Define a 🔧function `def find_name(left):` that 🔧returns a direction name given a left-side sensor reading.
  - This will use a `for` 🔧loop to search through `sensor_data`.
- Use the 🔧input function to prompt the user to enter a direction name on the console.
  - After a direction is entered, *get moving!*
- After your 'bot rotates, move forward to knock the **water bottle** down!
  - You'll need to type in the correct **direction** on the *console*.

Tools Found: Input Function, list, Functions, Parameters, Arguments, and Returns, Loops, Readability, str, Locals and Globals, Print Function

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # Sensor readings are around 500 counts apart.
5  # Define a minimum difference (counts) to be considered in the same region.
6  MIN_DIFF = 100
7
8  # A matrix of values [ [direction_name, left_sensor_value],... ]
9  sensor_data = [
10     ['N',  3569],
11     ['NE', 4080],
12     ['E',  369],
13     ['SE', 940],
14     ['S',  1450],
15     ['SW', 1991],
16     ['W',  2517],
17     ['NW', 3043]
18 ]
19
20 # Prompt user for direction
21 target_direction = input("Enter target direction: ")
22
23 def find_name(left):
24     """Search for a direction name given a left-side sensor reading"""
25     for d in sensor_data:
26         if abs(left - d[1]) < MIN_DIFF:
27             # Found it! Return the name.
28             return d[0]
29
30 # Spin clockwise, slowly
31 motors.enable(True)
32 motors.run(LEFT, 10)
33 motors.run(RIGHT, -10)
34
35 while True:
36     left = ls.read(1)
37     right = ls.read(2)
```

```
38      # print(left, right, sep=',')
39
40      # First make sure sensors are "straddling" different sections.
41      if abs(left - right) > MIN_DIFF:
42          found = find_name(left)
43          if found:
44              print(found)
45              if found == target_direction:
46                  break  # Leave motors enabled!
47
48  # Charge ahead bravely!
49  motors.run(LEFT, 40)
50  motors.run(RIGHT, 40)
51  sleep(3.0)
```

***Quiz 3* - Just One More Thing**

Question 1: You used a new Python built-in function: `abs()` What is the value of `x` after this code runs?

```
x = abs(-123)
```

✓ `123`

✗ `246`

✗ `-12.3`

✗ `-12.3`

Question 2: You used lists to build a *2-dimensional array* also known as a "Matrix".

- In the following 2D array, how would you reference the element `'g'`?

```
matrix = [
    ['a', 'b', 'c'],
    ['d', 'e', 'f'],
    ['g', 'h', 'i'],
]
```

✓ `matrix[2][0]`

✗ `matrix[3][1]`

✗ `matrix[2[0]]`

✗ `matrix[2,0]`

Question 3: What is the value of `in_zone` after the following code runs?

```
def check_zone(left, right):
    return abs(left - right) > 100

in_zone = check_zone(300, 400)
```

✓ False

✗ True

✗ -100

Question 4: What is the value of `target_direction` after the following code runs and the user enters `W` on the console?

```
def get_dir():
    input("Enter target direction: ")
```

```
target_direction = get_dir()
```

✓ `None`

✗ `"W"`

✗ `"w"`

✗ An error occurs: `UnknownReturnValue: Function definition without return statement.`

Mission 8 - Boundary Patrol

Program your CodeBot to roam a fenced area, using the line sensors to stay in bounds.

Objective 1 - Into the Unknown

Into the Unknown

Using CodeBot's 🔧line sensors to *explore your environment.*

Create a new file!

- Use the File → New File menu to create a new file called "boundary.py"

As you can see in the 3D view, your 'bot is positioned on a piece of white *sign-board*

- This stuff comes in a standard size for yard signs, and it's a nice surface for robots too :-)
- A rectangular border has been marked with black electrical tape.

You already know how to use the 🔧line sensors. So the first step in navigating across this new surface is to figure out what *sensor readings* you get from the white *surface* versus the black *lines*.

- Remember, *brighter reflection* → *lower* sensor reading
- So when you hit a border line, the sensor reading should be much larger.

Off the Table?

Drive forward slowly, taking sensor readings, until you *plunge* off the table.

> ***FOR SCIENCE!***

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5  motors.run(LEFT, 10)
6  motors.run(RIGHT, 10)
```

Keep your speed down here. Under 20% power will let you watch the console and track variations in surface reflectivity.

```
7
8  while True:
9      val = ls.read(0)
```

Just need to read a single value from 🔧line sensors LS-0 here. The other sensors would be reporting very nearly the same value, right?

```
10     print(val)
11     sleep(0.1)
```

🔧print the raw 🔧ADC value to the console, so you can watch it!

- The `sleep(0.1)` here is *optional* but without it you may have *thousands* of samples to scroll through on the console!

12

Hint:

- **Need to *clear* the console?**

  Just ***right-click*** in the console window and choose **"Terminal Clear"** from the pop-up options.

Goals:

- Drive straight ahead *slowly.* Keep your 🔧motor power below 20%
- Run an infinite 🔧loop, checking **LS-0** and 🔧printing the value to the console.
  - Just print the single sensor value alone, so I can check it easily.
  - Delay for 0.1 second between samples.

Tools Found: Line Sensors, Motors, Loops, Print Function, Analog to Digital Conversion

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5  motors.run(LEFT, 10)
6  motors.run(RIGHT, 10)
7
8  while True:
9      val = ls.read(1)
10     print(val)
11     sleep(0.1)
```

Objective 2 **- Toe the Line**

Toe the Line!

Okay, so now you have the *data.* Really it's just two numbers you'll need:

1. Sensor reading for the `surface = ??`
2. Sensor reading for the `line = ??`

   Based on that information, are the numbers far enough apart that your code can detect the *line?*

Careful, the numbers might change a little as you roll along!

- Color variations might occur on the *surface*.
- The front of your 'bot might *bounce* a little, causing the reading to be "darker" because the sensor is farther away.

Split the Difference

Choose a `threshold` value about midway between the *surface* and the *line.*

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  # Anything over this is a Line
5  threshold = 2000
```

Define your *threshold*

Higher than a valid surface reading. Anything above this value *must* be a Line!

```
6
7  # Drive straight ahead
8  motors.enable(True)
9  motors.run(LEFT, 20)
10 motors.run(RIGHT, 20)
11
12 while True:
13     val = ls.read(0)
14     print(val)
15     sleep(0.1)
16     if val > threshold:
17         break
```

Have you crossed the threshold?

Check the current sensor `val` against your pre-set `threshold`.

- If it looks like a Line, exit the loop so your program ends and motors stop.

```
18
```

Goal:

- Add an if condition to your code to `break` out of the loop if the sensor value exceeds your `threshold`.
  - Stop your motors right on the *line!*

Tools Found: Branching, Loops, Motors

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  threshold = 2000
5
6  motors.enable(True)
7  motors.run(LEFT, 20)
8  motors.run(RIGHT, 20)
9
10 while True:
11     val = ls.read(0)
12     print(val)
13     sleep(0.1)
14     if val > threshold:
15         break
```

Quiz 1 - Break Out

Question 1: What value of `threshold` would cause the following to print `0 1 2 3`?

```
for i in range(5):
    if i > threshold:
        break
    print(i, end=' ')
```

✗ `threshold = 2`

✓ `threshold = 3`

✗ `threshold = 4`

Question 2: How many times will the following loop read the line sensor?

```
i = 0
while i < 10:
    val = ls.read(0)
    print(val)

print("Done!")
```

✓ An *infinite* number of times, since the variable `i` is never changed.

✗ 9

✗ 10

Objective 3 - Speedy Stops

Try increasing the speed of your 'bot.

How about 100% power to the motors? *Go for it!*

Crossing the Line?

It's okay for the front edge of your bot to go over the line *a little*, but try not to leave the board!

> How can you increase speed *without* crossing the boundary line?

Sorry, but 100% power is *too fast* such a small board... But you might be able to run with 50% power, **if** you can **put on the *brakes!***

Strategy

1. Read the line sensors more quickly, to improve CodeBot's reaction time.
2. Increase the *stopping power* of the wheels by reversing the motors briefly.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  threshold = 2000
5
6  # A function to put on the brakes
7  def brake():
8      motors.run(LEFT, -70)
9      motors.run(RIGHT, -70)
10     sleep(0.3)
```

> Define a new function to *"brake"* by reversing the motors briefly.
>
> - Experiment to find the *power* and *duration* that work best for you.

```
11
12 # Drive straight ahead
13 motors.enable(True)
14 motors.run(LEFT, 50)
```

> Increase your speed. How fast can *you* go without leaving the board?

```
15 motors.run(RIGHT, 50)
16
17 while True:
18     val = ls.read(0)
19     if val > threshold:
20         break
```

> I completely removed `sleep()` and `print()` here.

- You can 🔧comment-out those lines instead if you prefer.

```
21
22  brake()
```

Call the `brake()` function immediately when your sensing loop ends.

Hint:

- ## Commenting-out Code

  An easy way to remove the `sleep()` and `print()` lines *temporarily* is to

  🔧comment them out

  Do this by placing a `#` in front of those lines.

  There's even a handy 🔧editor shortcut for doing just that for a line or selected block of code. Try **CTRL-/**.

Goals:

- Define a 🔧function called `def brake():` that reverses both motors briefly, to give you some stopping power.
  - Call your `brake()` function when you `break` out of the `while` loop.
- Remove the `sleep()` and `print()` function calls from your code. *Brake the **instant** your sensor detects a line!*

Tools Found: Motors, Line Sensors, Functions, Comments

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  threshold = 2000
5
6  def brake():
7      motors.run(LEFT, -70)
8      motors.run(RIGHT, -70)
9      sleep(0.3)
10
11
12 motors.enable(True)
13 motors.run(LEFT, 50)
14 motors.run(RIGHT, 50)
15
16 while True:
17     val = ls.read(0)
18     if val > threshold:
19         break
20
21 brake()
```

Objective 4 - Turn and Burn

Keep Moving - *Turn and Burn!*

The overall goal of this **Mission** is for CodeBot to continuously *roam* around inside the lines.

Here's a simple *algorithm* to do that:

1. Drive forward until you hit a line.
2. Slam on the brakes!
3. Back up a bit.
4. Turn *right.*

5. Repeat!

Complete this **Objective** by coding this algorithm.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  threshold = 2000
5  SPEED = 50  # motor % when driving forward
6
```

A constant SPEED

Often you will have some global settings that control how your code runs. Your code will be much more readable and easy to modify if you define these with *meaningful names.* That's much better than having "magic numbers" scattered throughout your code.

- There are still some *magic numbers* in the code below. Maybe you should fix those too!

```
7
8  # A function to simplify motor commands
9  def go(left, right, delay=0):
10     motors.run(LEFT, left)
11     motors.run(RIGHT, right)
```

A utility function `go()`

Often you'll want to set new motor LEFT/RIGHT speeds, sometimes followed by a `sleep()` delay.

- This function allows the caller to *omit* the delay if it's not needed.
- Read about default parameters to learn more.

```
12
13     if delay:
14         sleep(delay)
```

Last step in the `go()` function is to deal with `delay`.

- See how the integer value is used as a boolean here?
- Read about boolean to understand *truthy* and *falsy* values.

```
15
16 # A function to put on the brakes
17 def brake():
18     go(-70, -70, 0.3)
```

Check it out. Your `go()` function took 3 lines of code down to 1.

```
19
20 # Back up a bit and turn
21 def back_turn():
22     go(-50, -50, 0.5)
23     go(50, -50, 0.5)
24
25 motors.enable(True)
26
27 # Drive straight ahead
28 go(SPEED, SPEED)
```

Using the `SPEED` constant here.

```
29
30 while True:
31     val = ls.read(0)
```

```
32      if val > threshold:
33          # Hit a line: stop, turn, and go again
34          brake()
35          back_turn()
36          go(SPEED, SPEED)
```

Follow the *algorithm:*

- When a line is hit: brake, back up, turn, and go forward again!

```
37
38
```

Hint:

- ## Study the new concepts
  - Learn about constants.
  - Understand default parameters.

Goals:

- Define a function `def go(left, right, delay=0):` that uses a default parameter for `delay`. This should run the motors at the given speeds, then *optionally* `sleep(delay)`.
- Rewrite your `brake()` to use the new `go()` function. Also define a function to backup and turn right that also uses `go()`.
- Use a constant called `SPEED` to set the forward speed, near the top of your code so it's easy to change.
- **Don't `break`... *`brake()`*!** Remove the `break` from your `while` loop. Instead, call `brake()` then backup and turn inside your loop.

Tools Found: Default function parameters, Motors, Functions, Constants, Loops, Locals and Globals, Readability, int, bool

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  threshold = 2000
5  SPEED = 50  # motor % when driving forward
6
7  # A function to simplify motor commands
8  def go(left, right, delay=0):
9      motors.run(LEFT, left)
10     motors.run(RIGHT, right)
11
12     if delay:
13         sleep(delay)
14
15 # A function to put on the brakes
16 def brake():
17     go(-70, -70, 0.3)
18
19 # Back up a bit and turn
20 def back_turn():
21     go(-50, -50, 0.5)
22     go(50, -50, 0.5)
23
24 motors.enable(True)
25
26 # Drive straight ahead
27 go(SPEED, SPEED)
28
29 while True:
30     val = ls.read(0)
31     if val > threshold:
```

```
32          # Hit a line: stop, turn, and go again
33          brake()
34          back_turn()
35          go(SPEED, SPEED)
36
37
```

Quiz 2 **- Function Junction**

Question 1: Which *two* of the following cause CodeBot to rotate *clockwise* (right turn)?

```
def go_bot(left=50, right=50):
    motors.enable(True)
    motors.run(LEFT, left)
    motors.run(RIGHT, right)
```

✓ `go_bot(right = -50)`

✓ `go_bot(50, -50)`

✗ `go_bot(left = 50, -50)`

✗ `go_bot(right = 50)`

Question 2: Which *two* of the following variable names follow the style convention for Python constants?

✓ `IDLE_MODE`

✓ `BASIC`

✗ `Active_State`

✗ `boundary_line`

Question 3: What is the difference between function *arguments* and *parameters?*

✓ *Parameters* are named variables you list in the function **definition**. *Arguments* are the values passed when you **call** the function.

✗ *Parameter* is just another name for *Argument.* The terms are interchangeable.

✗ *Arguments* are innappropriate, and have no place in Python coding. Conversely, *Parameters* provide firm boundaries so that code can run more efficiently.

Question 4: What style of argument passing is this?

```
go_bot(30, 10)
```

✓ Positional

✗ Keyword

✗ Both *Positional* and *Keyword*

Question 5: What style of argument passing is this?

```
go_bot(left = 25)
```

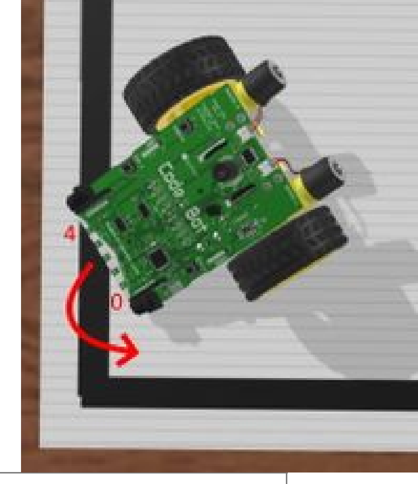
✓ Keyword

✗ Positional

❌ Both *Positional* and *Keyword*

***Objective 5* - Smarter Turns**

Smarter Turns



You have an *autonomous rover!*

> And it *almost* does a good job of staying in a bounded area. But sometimes your robot gets a little confused.

Your code always turns **right**, even if it hits the line with its right-front sensor first.

- A better plan would be to *turn away* from the corner that hits first. *(see picture)*
- Also you could make *smaller turns* when you know you aren't hitting the line *head-on*.

Using More Sensors

You will need to read the other 🔧line sensors to make better turning decisions.

- There are *several* improvements and new concepts in the **CodeTrek**!

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  threshold = 2000
5  SPEED = 50  # motor % when driving forward
6
7  def go(left, right, delay=0):
8      """A function to simplify motor commands"""
```

What's this? A 🔧docstring!

- Python provides a special way to document 🔧functions. *(other objects too!)*
- These *triple-quotes* are placed on either side, surrounding your **documentation comments** which can even span multiple lines.

For documenting 🔧functions this is preferred over regular comments.

```
9      motors.run(LEFT, left)
10     motors.run(RIGHT, right)
11
12     if delay:
13         sleep(delay)
14
15 def brake():
16     """A function to put on the brakes"""
17     go(-70, -70, 0.3)
18
19 def back_turn(turn_power=50):
```

Add a 🔧default parameter to your `back_turn()` 🔧function.

- This will work exactly the same as before when it's called with no 🔧arguments.
- But you can also pass it `turn_power` to change the *amount* and/or *direction* of the turn!

```
20     """Back up a bit and turn. Positive power turns right, negative turns left"""
21     go(-50, -50, 0.5)
22     go(turn_power, -turn_power, 0.5)
23
24 def scan_lines():
25     """Read all line sensors, compare with threshold, and return a list
26       of 5 bool results.
27     """
```

A new function that will check *all 5 sensors* and return a list of bool results.

```
28     sensors = []  # Start with an empty list
29     for i in range(5):
30         val = ls.read(i)
31         is_line = val > threshold
32         sensors.append(is_line)  # Fill list with is_line bools
33     return sensors
```

Study this *carefully*

Each time the function is called it starts with an *empty* list.

- The `for` loop ranges `i` from 0 to 4.
- The first time through the loop, **LS-0** is *read* and compared with `threshold`.
    - The bool result of this comparison becomes the *first* item in the list
- ...and so it continues until all 5 line sensors have been read!

All functions have just one return value. But you *can* return a single list with lots of values packed in it!

```
34
35 motors.enable(True)
36
37 # Drive straight ahead
38 go(SPEED, SPEED)
39
40 while True:
41     vals = scan_lines()
42     leds.ls(vals)
```

Let Your Line Lights Shine!

Those Line Sensor LEDs are positioned directly above the line sensors for a reason.

- And besides being able to handle a binary value, the `leds.ls(n)` API also works great if you pass it a list of bools.
- *Perfect* for showing the state of *all 5 sensors* in one simple function call!

```
43     if any(vals):
44         brake()
```

Know `any` Python built-ins?

This is a good one to know! The `any(iterable)` function returns `True` if *any* item in your list is `True`.

- That makes this a *super-fast* way to find out if CodeBot has hit a line.

First thing when you hit a line, `brake()`!

```
45         if vals[0] and not vals[4]:
46             # Left corner hit, turn right
47             back_turn(30)
```

Smart Turns

Check if *one* corner hit, but not the *other*.

- Notice if the *left* corner hit, you turn *right*.
- ...and turn with a little less speed.

```
48         elif vals[4] and not vals[0]:
49             # Right corner hit, turn left
50             back_turn(-30)
51         else:
52             back_turn()
53
54         go(SPEED, SPEED)
```

Don't forget to *drive forward*

Notice this is still indented under the `if any(vals):` check.

- Gotta resume *driving forward* after *braking* and *turning*.

55

Hint:

- **Step into your code with the *CodeSpace Debugger*.**
  - First click then use the button to *step* through your code.
  - Don't forget to watch your variables in the **console** panel!

Goals:

- Add a default parameter to your `back_turn()` function. Just one parameter to set both *power* and *direction* of the turn.
- Code a new function called `def scan_lines():` that reads all line sensors, compares with `threshold`, and returns a list of 5 bool results.
- Use the `any()` built-in to check if a line was hit.
- Add smarter turns so you stay on the board and **hit the *Water Bottle!***
- Succeed within a 30 second timeout

Tools Found: Line Sensors, Default function parameters, Functions, list, bool, Built-In Functions, Comments, Keyword and Positional Arguments, Loops, Comparison Operators, Parameters, Arguments, and Returns, CodeBot LEDs, Binary Numbers, API, Indentation

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  threshold = 2000
5  SPEED = 50  # motor % when driving forward
6
7  def go(left, right, delay=0):
8      """A function to simplify motor commands"""
9      motors.run(LEFT, left)
10     motors.run(RIGHT, right)
11
12     if delay:
13         sleep(delay)
14
15 def brake():
16     """A function to put on the brakes"""
17     go(-70, -70, 0.3)
18
19 def back_turn(turn_power=50):
20     """Back up a bit and turn. Positive power turns right, negative turns left"""
21     go(-50, -50, 0.5)
22     go(turn_power, -turn_power, 0.5)
23
24 def scan_lines():
25     """Read all line sensors, compare with threshold, and return a list
26       of 5 bool results.
27     """
28     sensors = []  # Start with an empty list
29     for i in range(5):
30         val = ls.read(i)
31         is_line = val > threshold
32         sensors.append(is_line)  # Fill list with is_line bools
```

```
33      return sensors
34
35  motors.enable(True)
36
37  # Drive straight ahead
38  go(SPEED, SPEED)
39
40  while True:
41      vals = scan_lines()
42      leds.ls(vals)
43      if any(vals):
44          print(vals, ls.check(0))
45          brake()
46          if vals[0] and not vals[4]:
47              # Left corner hit, turn right
48              back_turn(30)
49          elif vals[4] and not vals[0]:
50              # Right corner hit, turn Left
51              back_turn(-30)
52          else:
53              back_turn()
54
55          go(SPEED, SPEED)
56
```

Objective 6 **- Enter the Dohyō**

Get Your *Sumo* On!

A classic robot competition event is **Robot-Sumo**. It is a sport where two robots attempt to push each other out of a Dohyō, which is a circular area like the one shown in the 3D view.

- Your CodeBot is sitting at the center of a regulation *Mini-Sumo* Dohyō.
- You've been invited to compete against the *dreaded* Water Bottle!

This is your final Objective of this Mission

And you just have a *little* more code to write in order to achieve it!

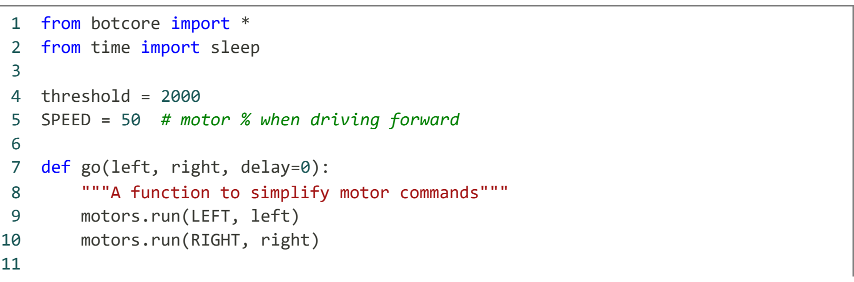
Hint:

- **Your code from the previous Objective is *very* close!**
    - Take a look at where you are comparing the sensor reading against a `threshold`.
    - Remember, *brighter reflection* → *lower* sensor reading
    - So when you hit a border line, the sensor reading should be much **lower** than the `threshold`.

Goals:

- Modify your code to detect a *reflective* Line against a *dark* surface.
    - *Roam* until you defeat the water bottle!
- Complete the battle within 30 seconds

Solution:



```
1   from botcore import *
2   from time import sleep
3
4   threshold = 2000
5   SPEED = 50  # motor % when driving forward
6
7   def go(left, right, delay=0):
8       """A function to simplify motor commands"""
9       motors.run(LEFT, left)
10      motors.run(RIGHT, right)
11
```

```
12      if delay:
13          sleep(delay)
14
15  def brake():
16      """A function to put on the brakes"""
17      go(-70, -70, 0.3)
18
19  def back_turn(turn_power=50):
20      """Back up a bit and turn. Positive power turns right, negative turns left"""
21      go(-50, -50, 0.5)
22      go(turn_power, -turn_power, 0.5)
23
24  def scan_lines():
25      """Read all line sensors, compare with threshold, and return a list
26         of 5 bool results.
27      """
28      sensors = []  # Start with an empty list
29      for i in range(5):
30          val = ls.read(i)
31          is_line = val < threshold
32          sensors.append(is_line)  # Fill list with is_line bools
33      return sensors
34
35  motors.enable(True)
36
37  # Drive straight ahead
38  go(SPEED, SPEED)
39
40  while True:
41      vals = scan_lines()
42      leds.ls(vals)
43      if any(vals):
44          brake()
45          if vals[0] and not vals[4]:
46              # Left corner hit, turn right
47              back_turn(30)
48          elif vals[4] and not vals[0]:
49              # Right corner hit, turn left
50              back_turn(-30)
51          else:
52              back_turn()
53
54          go(SPEED, SPEED)
55
```

Mission 9 - Line Following

Tune up your Line Sensors and hit the road on the biggest and baddest line-course around. Can your Python code master this challenge?

Objective 1 - Sensors Ready

Create a new file!

- Use the File → New File menu to create a new file called "line_scanner.py"

Ready Your Sensors

Your **line follower** 'bot will need to continuously check for the *presence* of a line beneath **all 5** 🔧line sensors.

- You did that in the previous mission with the `scan_lines()` 🔧function.

Begin this Mission by:

1. Defining a `threshold` sensor value midway between the white sign-board *surface* and the black electrical tape *line* readings.
2. Calling `scan_lines()` continuously inside an infinite `while` 🔧loop.
3. Displaying the result on the 🔧line sensor LEDs.

Are your LS LEDs LOL?

LOL, like: *"Lit On the Line"*

CodeTrek:

```
1  from botcore import *
2
3  threshold = 2000
```

This is about midway between *surface* and *line* sensor reading for my 'bot.

```
4
5  def scan_lines():
6      """Read all line sensors, compare with threshold, and return a list
7         of 5 bool results.
8      """
9      sensors = []  # Start with an empty list
10     for i in range(5):
11         val = ls.read(i)
12         is_line = val > threshold
13         sensors.append(is_line)  # Fill list with is_line bools
14     return sensors
```

The function returns a list.

Step through this code and watch the list grow in the Console *Variables* panel!

- Variables defined inside a function, including the parameters, are local variables.
- If you're interested, read up on the difference between local and global. The knowledge will soon come in handy!

```
15
16 while True:
17     vals = scan_lines()
18     leds.ls(vals)
```

Pretty simple loop

All it's doing is *reading the sensors* and *displaying the values* on the LEDs.

- That's a good start!

```
19
```

Goals:

- Run your *infinite* *loop* displaying `scan_lines()` result on the Line Sensor LEDs.
  - On RESET your 'bot is positioned on a line, so you should see at least **one** of the *middle-three* **LS LEDs** lit up!
- **Step into your code with the *Debugger*.**
  - Use the "Step In" button, and step into `scan_lines()`.
  - Watch the `sensors` list grow in the Console *Variables / Locals* view.

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

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  threshold = 2000
5
6  def scan_lines():
7      """Read all line sensors, compare with threshold, and return a list
```

```
8         of 5 bool results.
9     """
10    sensors = []  # Start with an empty list
11    for i in range(5):
12        val = ls.read(i)
13        is_line = val > threshold
14        sensors.append(is_line)  # Fill list with is_line bools
15    return sensors
16
17 while True:
18    vals = scan_lines()
19    leds.ls(vals)
20
```

Objective 2 - Sensor Hacking on the REPL

Hacking Sensors and Lists

As you've seen, converting the line sensor values into a list is quite useful.

- Really, this is all you need to build an excellent *Line Follower*.
- But from here on, your code will only grow more *complex*.
- ...so you should explore ways to further simplify and *optimize* what you have!

There *are* ways to optimize your code. To learn them you must go...

To the REPL!

So far you have used the ≡ **Console** to:

1. *Output* messages using the print function.
2. Get keyboard *input* strings using the input function.

But there is an even more powerful capability hidden there. You can enter Python code *interactively!* Learn more in the REPL tool. *You can:*

- Test Python functions, expressions, and data types.
- import libraries and experiment with APIs.
- Use it as a *calculator!*

Be sure to *stop* your program before continuing

Then open the **Console** panel, click there, and *interact* to complete this Objective.

Goals:

- It's a *calculator!* Click in the REPL panel and type: `2 + 2` then press ENTER.
- Watch your 'bot as you enter these 2 lines:

```
from botcore import *
leds.user(1)
```

  If the LED is already On, turn it Off first

- Type: `ls.read(0)` to instantly read sensor 0 and see the *result* right on the **Console**.
  - You can use the *Up-Arrow* ⇧ key to repeat commands.
- With a list comprehension you can code your whole `scan_lines()` function in *one line!*
  - Type: `[ls.read(i) > 2000 for i in range(5)]`
- Finally, the line sensors **API** has a built-in function that does what you need!
  - Type: `ls.check()`

Tools Found: Line Sensors, list, REPL, Print Function, str, Input Function, import, API, List Comprehension, Loops

Solution:

```
1  # All code entered on the REPL interactively
```

Quiz 1 - List comprehensions and Tuples

Question 1: What is the result of the following 🔧list comprehension?

```
[i**2 for i in range(5)]
```

(Try it on the REPL if you like!)

✓ [0, 1, 4, 9, 16]

✗ [0, 2, 4, 6, 8]

✗ (0, 1, 4, 9, 16)

Question 2: Explore the 🔧line sensors documentation.

- What type of data does `ls.check()` return?

✓ 🔧tuple

✗ 🔧list

✗ 🔧string

Question 3: A 🔧tuple is similar to a 🔧list, but with important differences.

```
# Define a tuple using parenthesis rather than square brackets.
t = ("Zero", "One", "Two")
```

- Which TWO of the following can you *NOT* do with a 🔧tuple?

✓ Change an item: `t[1] = "Uno"`

✓ Append an item: `t.append("Three")`

✗ Retrieve an item: `result = t[2]`

✗ Check the length: `n = len(t)`

Objective 3 - Bang Bang Control

Simple Bang-Bang Line Follower

Ready to Code a Line Follower?

To begin with you will just use the 2 outermost 🔧line sensors

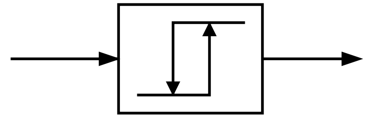
LS-0 and **LS-4**

Your *algorithm* is simple:

- Left sensor hit line? *Turn LEFT.*
- Right sensor hit line? *Turn RIGHT.*
- ...otherwise go straight.

This is actually a type of *control system* called a *"bang-bang controller"*.

You can think of the robot "banging" against the LEFT and RIGHT sensors as it swerves down the line!



- There are much more sophisticated *control systems*, some of which you will be exploring soon.

Create a new file!

- Use the File → New File menu to create a new file called "bang_bang.py"

CodeTrek:

```
1  from botcore import *
2
3  # Set overall speed and turning parameters
4  SPEED = 30
5  TURN_FACTOR = 0.2  # Lower value is sharper turn
6
```

These are the constants that determine the *top speed* and how sharp the *turns* are as you navigate the line.

- **Note:** You will need to *experiment* with these numbers to stay on the line consistently.
- The simple *bang-bang* controller can't handle high speeds...

```
7
8  motors.enable(True)
9
10 while True:
11     vals = ls.check(2000)
12     leds.ls(vals)
```

Use the built-in line sensors `ls.check()` API for speed and simplicity.

```
13
14     if vals[0]:
15         # Hit left --> turn left
16         motors.run(LEFT, SPEED * TURN_FACTOR)
17         motors.run(RIGHT, SPEED)
```

Bang-Bang

Just use the *outside sensors:*

- LS-0 → LEFT side
- LS-4 → RIGHT side

```
18     elif vals[4]:
19         # Hit right --> turn right
20         motors.run(LEFT, SPEED)
21         motors.run(RIGHT, SPEED * TURN_FACTOR)
22     else:
23         # Default to driving forward at full speed
24         motors.run(LEFT, SPEED)
25         motors.run(RIGHT, SPEED)
26
```

Hint:

- You will probably need to reduce both the `SPEED` and `TURN_FACTOR` values shown in the **CodeTrek**.
    - It's okay to go slowly and turn sharply!

Goals:

- Code your *bang-bang* line follower, and attempt to drive the Line Follow Course. *Make it at least as far as the **first checkpoint.***
- Get to the checkpoint in 45 seconds or less

Tools Found: Line Sensors, Constants, API

Solution:

```
1  from botcore import *
2
3  SPEED = 30
4  TURN_FACTOR = 0.2
5
6  motors.enable(True)
7
8  while True:
9      vals = ls.check(2000)
10     leds.ls(vals)
11
12     if vals[0]:
13         # Hit left --> turn left
14         motors.run(LEFT, SPEED * TURN_FACTOR)
15         motors.run(RIGHT, SPEED)
16     elif vals[4]:
17         # Hit right --> turn right
18         motors.run(LEFT, SPEED)
19         motors.run(RIGHT, SPEED * TURN_FACTOR)
20     else:
21         motors.run(LEFT, SPEED)
22         motors.run(RIGHT, SPEED)
23
```

Objective 4 - Smarter Turns

Smarter Turns

Try running your bot a few times on the course using the *Attached* camera view.

- Pay close attention when your bot "loses" the line.
- What do the motors do in this case?

Problem:

When your bot loses the line it just keeps driving forward at full speed!

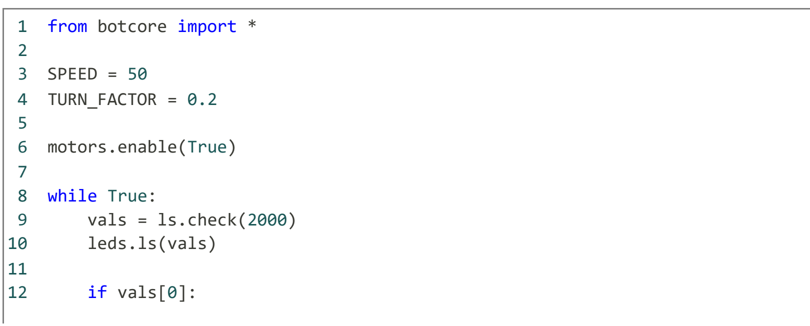
Solution:

Only drive forward if you're *on* the line.

What if you're completely *off* the line?

- Usually this happens when you missed a turn.
- But usually the bot *starts* to turn, and just overshoots. So if you lose the line, ***keep turning!***

CodeTrek:



```
1  from botcore import *
2
3  SPEED = 50
4  TURN_FACTOR = 0.2
5
6  motors.enable(True)
7
8  while True:
9      vals = ls.check(2000)
10     leds.ls(vals)
11
12     if vals[0]:
```

```
13          # Hit left --> turn left
14          motors.run(LEFT, SPEED * TURN_FACTOR)
15          motors.run(RIGHT, SPEED)
16      elif vals[4]:
17          # Hit right --> turn right
18          motors.run(LEFT, SPEED)
19          motors.run(RIGHT, SPEED * TURN_FACTOR)
20      elif vals[1] or vals[2] or vals[3]:
21          # Go straight only if on Line
22          motors.run(LEFT, SPEED)
23          motors.run(RIGHT, SPEED)
```

Replace your `else` with an `elif`

Check to see if any of the middle 3 line sensors detect a line. Only drive forward if this is true!

- This is a nice use of the or operator.

Now what happens if you lose the line?

- Your code just keeps doing what it *was* doing... hopefully *turning!*

```
24
```

Hint:

- **Add an `elif` that confirms you're on the line**

  If you've lost the line, just keep acting on the most recent sensor data you have.

  - Hopefully you were already turning, just not quite sharp enough.

Goals:

- Use the **CodeTrek** to Improve your *bang-bang* line follower and reach **Checkpoint 1**
- With your improved code, reach **Checkpoint 2**
- Get to Checkpoint 2 in 90 seconds or less

Tools Found: Logical Operators

Solution:

```
1  from botcore import *
2
3  SPEED = 50
4  TURN_FACTOR = 0.2
5
6  motors.enable(True)
7
8  while True:
9      vals = ls.check(2000)
10     leds.ls(vals)
11
12     if vals[0]:
13         # Hit left --> turn left
14         motors.run(LEFT, SPEED * TURN_FACTOR)
15         motors.run(RIGHT, SPEED)
16     elif vals[4]:
17         # Hit right --> turn right
18         motors.run(LEFT, SPEED)
19         motors.run(RIGHT, SPEED * TURN_FACTOR)
20     elif vals[1] or vals[2] or vals[3]:
21         # Go straight only if on Line
22         motors.run(LEFT, SPEED)
23         motors.run(RIGHT, SPEED)
24
```

Quiz 2 - Get Logical

Question 1: You used the Python operator `or` in the previous Objective. Take a look at 🔧logical operators in your *Toolbox* if needed to answer this:

Which 3 of the following are `True`?

✓ `True or False`

✗ `True and False`

✓ `True and True`

✗ `False or False`

✓ `False or True`

Question 2: What is printed by the following?

```
for i in range(5):
    if i < 2:
        print('A', end='.')
    elif i > 3:
        print('B', end='.')
    else:
        print('C', end='.')
```

✓ A.A.C.C.B.

✗ A A C B B

✗ A.A.B.C.C.

✗ A.B.C.C.B.

Objective 5 - Sharpen Your Sensors

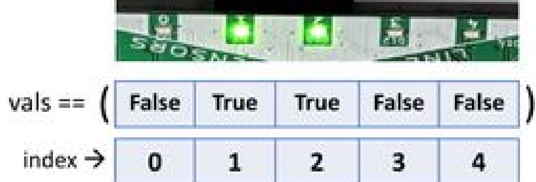
Extracting Information from the Sensors

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 the line of course!

Run your last program, and observe the 🔧Line Sensor LEDs as your bot moves across the line.

- What are ***all*** the sensor combinations you see while following a line?



Example: *My Collected Data*

Using standard 3/4" black electrical tape on a white surface, I got the following:

Line Pos	LEDs (vals)	Error
Right	(0,0,0,0,1)	+5
↑	(0,0,0,1,1)	+4
¦	(0,0,1,1,1)	+3
↓	(0,0,0,1,0)	+2
Right	(0,0,1,1,0)	+1
Center	(0,1,1,1,0)	0
Center	(0,0,1,0,0)	0

Note 1: I'm using 1 and 0 rather than True and False.

Note 2: The *Error* value is a measure of how far off-center the bot is.

Line Pos	LEDs (vals)	Error
Left	(0,1,1,0,0)	-1
↑	(0,1,0,0,0)	-2
¦	(1,1,1,0,0)	-3
↓	(1,1,0,0,0)	-4
Left	(1,0,0,0,0)	-5

As the table above shows, I can detect **5 steps** of *off-center* in both *Left* and *Right* directions.

Do your results agree?

Write code to 🔧print sensor 🔧tuples to the console.

CodeTrek:

```
1  from botcore import *
2
3  SPEED = 50
4  TURN_FACTOR = 0.2
5
6  prev_vals = None
```

Use this 🔧variable to keep track of the previous `vals`

- Initialize to 🔧None.
- Each time you read the line sensors, you'll save the results here.

```
7
8  motors.enable(True)
9
10 while True:
11     vals = ls.check(2000)
12     leds.ls(vals)
13
14     # Print the sensor data when it changes
15     if vals != prev_vals:
16         print(vals)
17         prev_vals = vals  # Save as "previous" value
18
```

Print the data

But ***only*** if it has changed!

- The very first time, `prev_vals == None` so you will definitely print.
- After that, you print only `if` the `vals` are different than last time.

```
19
20     if vals[0]:
21         # Hit left --> turn left
22         motors.run(LEFT, SPEED * TURN_FACTOR)
23         motors.run(RIGHT, SPEED)
24     elif vals[4]:
25         # Hit right --> turn right
26         motors.run(LEFT, SPEED)
27         motors.run(RIGHT, SPEED * TURN_FACTOR)
28     elif vals[1] or vals[2] or vals[3]:
29         # Go straight only if on line
30         motors.run(LEFT, SPEED)
31         motors.run(RIGHT, SPEED)
32
```

Goal:

- Add a `print()` statement to your code to show the 🔧line sensor `vals`.
    - *BUT* only display new values when they've changed!

Tools Found: CodeBot LEDs, Print Function, tuple, Line Sensors, Variables, None

Solution:

```
1  from botcore import *
2
3  SPEED = 50
4  TURN_FACTOR = 0.2
5  prev_vals = None
6
7  motors.enable(True)
8
9  while True:
10     vals = ls.check(2000)
11     leds.ls(vals)
12     if vals != prev_vals:
13         print(vals)
14         prev_vals = vals
15
16     if vals[0]:
17         # Hit left --> turn left
18         motors.run(LEFT, SPEED * TURN_FACTOR)
19         motors.run(RIGHT, SPEED)
20     elif vals[4]:
21         # Hit right --> turn right
22         motors.run(LEFT, SPEED)
23         motors.run(RIGHT, SPEED * TURN_FACTOR)
24     elif vals[1] or vals[2] or vals[3]:
25         # Go straight only if on line
26         motors.run(LEFT, SPEED)
27         motors.run(RIGHT, SPEED)
28
```

***Objective 6* - Proportional Data**

Proportional Data

Use your *data* about the line sensors to tune the turning so your bot can go faster while staying on the line.

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 turn *in proportion* to how far off-center CodeBot is?

Refer to your *table* of sensor data from the last Objective.

- When you get new `vals` from the line sensors you need to find the corresponding *Error* number.
- This could be done by searching through a list, like you did in the **Line Sensors** mission. (Remember searching for "N", "S", "E", "W"?)

But there is a better way...

A Python dictionary! Dictionaries are an excellent choice when you need to look-up values from a table.

- The *keys* will be tuples straight from `ls.check()`
- The *values* will be the **Error** int numbers from your table.

Create a new file!

- Use the File → New File menu to create a new file called "line_follower.py"

CodeTrek:

```
1  from botcore import *
2
3  SPEED = 50
4  TURN_FACTOR = 0.2
5
6  ls_err = {
7      (0,0,1,0,0) : 0,
8      (0,1,1,1,0) : 0,
9
10     (0,0,1,1,0) : 1,
```

Your Data Dictionary

Comprised of `{key: value, }` pairs that map the line sensor readings to *Error* numbers.
This is the *data table* from the previous Objective, encoded as a Python dictionary.

- Read about Python's built-in dictionary type for more information.

```
11     (0,0,0,1,0) : 2,
12     (0,0,1,1,1) : 3,
13     (0,0,0,1,1) : 4,
14     (0,0,0,0,1) : 5,
15
16     (0,1,1,0,0) : -1,
17     (0,1,0,0,0) : -2,
18     (1,1,1,0,0) : -3,
19     (1,1,0,0,0) : -4,
20     (1,0,0,0,0) : -5,
21 }
22
23 motors.enable(True)
24
25 while True:
26     vals = ls.check(2000)
27     leds.ls(vals)
28
29     # Look up error value in dictionary
30     err = ls_err[vals]
31     print(vals, err)
```

Just like accessing items from a list or tuple, you can use *square brackets* to lookup values in a dictionary.

```
value = dict[key]
```

- But what happens if the *key* is not in the dictionary?
- Don't worry about that for now... just code it as shown!

```
32
33     # Use error value to turn (simple version for now...)
34     if err < 0:
35         # Line on left --> turn left
36         motors.run(LEFT, SPEED * TURN_FACTOR)
37         motors.run(RIGHT, SPEED)
38     elif err > 0:
39         # Line on right --> turn right
40         motors.run(LEFT, SPEED)
41         motors.run(RIGHT, SPEED * TURN_FACTOR)
42     else:
43         # Go straight only if zero error
44         motors.run(LEFT, SPEED)
45         motors.run(RIGHT, SPEED)
46
```

Hints:

- **Dictionary Format**

```
ls_err = {
    # tuple : int
    (0,0,1,0,0) : 0,
    (0,1,1,1,0) : 0,

    (0,0,1,1,0) : 1,
    ...
}
```

- **Expect a KeyError!**

  This Objective is leading you to experience a runtime error.

  - Your 🔧dictionary doesn't include every possible 🔧tuple the 🔧line sensors can return.
  - When a key lookup fails, you'll see a `KeyError` occur, which will stop your bot in its tracks!

Goals:

- Encode your sensor data table as a 🔧dictionary
- Run the Line Following course using 🔧dictionary lookup `ls_err[vals]` to retrieve error values for steering.

Tools Found: Line Sensors, dictionary, tuple, int, list

Solution:

```
1  from botcore import *
2
3  SPEED = 30
4  TURN_FACTOR = 0.1
5
6  ls_err = {
7      (0,0,1,0,0) : 0,
8      (0,1,1,1,0) : 0,
9
10     (0,0,1,1,0) : 1,
11     (0,0,0,1,0) : 2,
12     (0,0,1,1,1) : 3,
13     (0,0,0,1,1) : 4,
14     (0,0,0,0,1) : 5,
15
16     (0,1,1,0,0) : -1,
17     (0,1,0,0,0) : -2,
18     (1,1,1,0,0) : -3,
19     (1,1,0,0,0) : -4,
20     (1,0,0,0,0) : -5,
21 }
22
23 motors.enable(True)
24
25 while True:
26     vals = ls.check(2000)
27     leds.ls(vals)
28
29     # Look up error value in dictionary
30     err = ls_err[vals]
31     print(vals, err)
32
33     # Use error value to turn (simple version for now...)
34     if err < 0:
35         # Line on Left --> turn Left
36         motors.run(LEFT, SPEED * TURN_FACTOR)
37         motors.run(RIGHT, SPEED)
38     elif err > 0:
39         # Line on right --> turn right
40         motors.run(LEFT, SPEED)
41         motors.run(RIGHT, SPEED * TURN_FACTOR)
42     else:
43         # Go straight only if zero error
```

```
44          motors.run(LEFT, SPEED)
45          motors.run(RIGHT, SPEED)
46
```

Quiz 3 - Sensor Dictionaries

Question 1: Given the dictionary: `basket = {'apples': 12, 'bananas': 5}`

What is `basket['apples']` ?

- ✓ 12
- ✗ 5
- ✗ KeyError
- ✗ bananas

Question 2: Given the dictionary: `basket = {'apples': 12, 'bananas': 5}`

What is `basket['oranges']` ?

- ✓ KeyError
- ✗ 12
- ✗ 5
- ✗ 0

Question 3: Given the dictionary: `basket = {'apples': 12, 'bananas': 5}`

What is `basket.get('oranges', 0)` ?

- ✓ 0
- ✗ 12
- ✗ 5
- ✗ KeyError

Objective 7 - Proportional Control

Proportional Control Algorithm

You encountered a *KeyError* in the last Objective when you hit a combination of line sensors that weren't in your dictionary.

- What do you *want* to happen when "invalid" sensor values are seen?

Well, chances are CodeBot has just overshot a turn. A reasonable response would be to *keep turning!*

Avoiding KeyError

Review the `get()` API described in the dictionary tool.

- You can provide a *default* value to be used when the **key** is not found!
- Modify your code to use this method rather than *square brackets* `[ ]` to access the dictionary.

Putting the "P" in PID Controller

You are building a *Control System*. The principles are the same as if you were building a giant robotic arm that can swing around full-sized car frames, or creating a navigation system for a massive ship!

- You can read more about **PID Controllers** at the link above, but for now I'll walk you through creating one in Python for your bot!
- Start with the *Proportional* term, which uses the current `err` value to control the amount of turning.
- A constant called `Kp` is multiplied by the `err` to control the influence of this term on steering. $P = K_p e(t)$

See the *CodeTrek* for more details!

After you run this code, try *tuning* the K_p factor.

- Higher values increase the turning power, so you can manage sharper turns.
- But turning too hard also has negative consequences...

CodeTrek:

```
1  from botcore import *
2
3  SPEED = 30
4
5  # Dictionary mapping {sensors_tuple : error_int}
6  ls_err = {
7      (0,0,1,0,0) : 0,
8      (0,1,1,1,0) : 0,
9
10     (0,0,1,1,0) : 1,
11     (0,0,0,1,0) : 2,
12     (0,0,1,1,1) : 3,
13     (0,0,0,1,1) : 4,
14     (0,0,0,0,1) : 5,
15
16     (0,1,1,0,0) : -1,
17     (0,1,0,0,0) : -2,
18     (1,1,1,0,0) : -3,
19     (1,1,0,0,0) : -4,
20     (1,0,0,0,0) : -5,
21 }
22
23 def drive(speed, turn_ratio):
24     """Drive, using a fraction of the speed for turning."""
25     # speed: 0-100; turn_ratio: L=-1, R=+1, 0=straight
26     turn_spd = speed * turn_ratio
27     fwd_spd = speed - abs(turn_spd)
28     motors.run(LEFT, fwd_spd + turn_spd)
29     motors.run(RIGHT, fwd_spd - turn_spd)
```

A *steering* function: drive()

This makes it easy to steer using your +/- *error* value.

- Give it a max `speed`, and use `turn_ratio` to adjust the steering.
- A `turn_ratio` of `0` will go straight. *Do the math!*
- Remember the `abs()` 🔧built-in?

```
30
31 def apply_control(err):
32     """Control steering based on error"""
```

Control the Action

This is the "brains" of your control algorithm!

- It figures out how to drive the motors...
- based on the error *feedback* from the 🔧line sensors.

```
33     Kp = 0.1    # Proportional factor
```

```
34      steering = err * Kp
```

A "proportional" controller.

- This is the "P" term of your PID controller. $P = K_p e(t)$
- `err` is the instantaneous error $e(t)$ at time t

```
35      drive(SPEED, steering)
36
37  motors.enable(True)
38
39  # In case we start off the line, init with small error to cause turn in big circle.
40  err = 1
```

An initial value for `err`

In case your first dictionary lookup fails!

```
41
42  # Main Loop
43  while True:
44      # Read the sensors and display on LEDs
45      vals = ls.check(2000)
46      leds.ls(vals)
47
48      # Lookup error value (default to previous value)
49      err = ls_err.get(vals, err)
```

Dictionary Lookup

Use the `get(key, default)` method of 🔧dictionary objects.

- Just like `ls_err[key]` this will return an item when `key` is found.
- But it has the advantage of returning your supplied `default` when the lookup fails, rather than causing a *KeyError*.

Your previous value of `err` is used as the *default*, so the bot keeps using the last error value when it loses the line.

```
50      print(vals, err)
51
52      # Control based on error
53      apply_control(err)
```

Pass the new error value to your control algorithm!

- A nice, simple main loop!

```
54
```

Goals:

- Modify your code to use the 🔧dictionary `get()` method for looking up the sensor error value.
- Reach *Checkpoint 1*
- Reach *Checkpoint 2*
- Reach *Checkpoint 3*
- Reach *Checkpoint 4*
- Clear all *Checkpoints* in 3½ minutes or less

Tools Found: Line Sensors, dictionary, API, Built-In Functions

Solution:

```
1  from botcore import *
2
3  SPEED = 50
4  # Dictionary mapping {sensors_tuple : error_int}
5  ls_err = {
6      (0,0,1,0,0) : 0,
7      (0,1,1,1,0) : 0,
8
9      (0,0,1,1,0) : 1,
10     (0,0,0,1,0) : 2,
11     (0,0,1,1,1) : 3,
12     (0,0,0,1,1) : 4,
13     (0,0,0,0,1) : 5,
14
15     (0,1,1,0,0) : -1,
16     (0,1,0,0,0) : -2,
17     (1,1,1,0,0) : -3,
18     (1,1,0,0,0) : -4,
19     (1,0,0,0,0) : -5,
20 }
21
22 def drive(speed, turn_ratio):
23     """Drive, using a fraction of the speed for turning."""
24     # speed: 0-100; turn_ratio: L=-1, R=+1, 0=straight
25     turn_spd = speed * turn_ratio
26     fwd_spd = speed - abs(turn_spd)
27     motors.run(LEFT, fwd_spd + turn_spd)
28     motors.run(RIGHT, fwd_spd - turn_spd)
29
30 def apply_control(err):
31     """Control steering based on error"""
32     Kp = 0.1    # Proportional factor
33     steering = err * Kp
34     drive(SPEED, steering)
35
36 motors.enable(True)
37
38 # In case we start off the line, init with small error to cause turn in big circle.
39 err = 1
40
41 # Main Loop
42 while True:
43     # Read the sensors and display on LEDs
44     vals = ls.check(2000)
45     leds.ls(vals)
46
47     # Lookup error value (default to previous value)
48     err = ls_err.get(vals, err)
49     print(vals, err)
50
51     # Control based on error
52     apply_control(err)
53
```

Objective 8 **- Stats on the Line**

Line Up!

The final challenge of this Mission is to complete the full course at the fastest speed you can manage.

- To do that you'll need to add the remaining terms to your *PID Controller* algorithm.

Proportional: *The "P" in PID*

Your algorithm *already* uses the proportial term. $= K_p e(t)$

- This lets your CodeBot react to its **present** (instantaneous) position on the line.

Integral: *The "I" in PID*

The integral term accumulates **past** *error* values over a period of time. $= K_i \int e(t)dt$

- This gives your CodeBot some "memory" about the what the line has been doing in the **past**.

Derivative: *The "D" in PID*

The derivative term represents the rate of change in the *error* value. $= K_d \frac{de(t)}{dt}$

- This lets your CodeBot predict the **future** based on the *trend* in the line.

Remembering History, Tracking Trends

Right now your code considers only the **current** value of *error*. Is that all the data your sensors have to offer? ***No way!*** *As* ***time*** *passes you can collect more insights from those sensors!*

You will need to start collecting ***statistics*** on the *error* value.

- Follow the *CodeTrek* to get started!

CodeTrek:

```
1  from botcore import *
2  import time
3
4  SPEED = 50
5
6  # Dictionary mapping {sensors_tuple : error_int}
7  ls_err = {
8      (0,0,1,0,0) : 0,
9      (0,1,1,1,0) : 0,
10
11     (0,0,1,1,0) : 1,
12     (0,0,0,1,0) : 2,
13     (0,0,1,1,1) : 3,
14     (0,0,0,1,1) : 4,
15     (0,0,0,0,1) : 5,
16
17     (0,1,1,0,0) : -1,
18     (0,1,0,0,0) : -2,
19     (1,1,1,0,0) : -3,
20     (1,1,0,0,0) : -4,
21     (1,0,0,0,0) : -5,
22 }
23
24 def drive(speed, turn_ratio):
25     """Drive, using a fraction of the speed for turning."""
26     # speed: 0-100; turn_ratio: L=-1, R=+1, 0=straight
27     turn_spd = speed * turn_ratio
28     fwd_spd = speed - abs(turn_spd)
29     motors.run(LEFT, fwd_spd + turn_spd)
30     motors.run(RIGHT, fwd_spd - turn_spd)
31
32
33 def apply_control(err):
34     """Control steering based on error"""
35     Kp = 0.1    # Proportional factor
36     steering = err * Kp
37     drive(SPEED, steering)
38
39 # Error statistics (global state variables)
40 err_avg = 0
41 err_trend = 0
42 t_prev = 0  # ms time of previous sample
43
```

Initialize your "statistics" variables.

- These need to exist for the life of the program, so they must be defined *outside* of your function.
- That means they are global variables, rather than just *local* to the function.

```
44
```

```
45 def collect_stats(err):
46     """Update global stats based on stream of err values"""
```

A new function that gathers *statistics* on your Error values.

- You'll call it continuously with the latest `err` value.
- It should update the global variables above.

```
47
48     # This function updates global variables...
49
#@3
50
51     # Sensor only updates every 10 to 30ms
52     SAMPLE_INTERVAL = 10  # ms
53     HISTORY_FACTOR = 0.1  # Higher is more forgetful
54
```

Some constants this function needs. *More explanation below.*

```
55
56     t_now = time.ticks_ms()
57     if t_now - t_prev > SAMPLE_INTERVAL:
58         t_prev = t_now
```

Right on Schedule

These statistics are collected over time, and it's important that the time period of each "sample" is consistent.

This code checks the current time `ticks_ms()` and the `if` statement makes sure the stats are updated only if SAMPLE_INTERVAL milliseconds have elapsed since the last sample time `t_prev`.

Be sure to add `import time` at the top of your file!

```
59         err_avg_prev = err_avg
60         # Exponentially weighted moving average
61         err_avg = err * HISTORY_FACTOR + err_avg * (1 - HISTORY_FACTOR)
62         # Trend is based on last 2 average values
63         err_trend = (err_avg - err_avg_prev) * 10
```

Stats

`err_avg` is an *Exponentially Weighted Moving Average*. That means more recent samples have more influence.

- Think of the HISTORY_FACTOR as a percentage of influence the new sample has against the historical average.

`err_trend` is the rate of change in *Error* based on the last two `err_avg` values.

- The x10 scaling factor just helps keep K_d in a more convenient range.

```
64
65         # DEBUG - dump some variables to the console.
66         print(t_now, err, err_avg, err_trend)
```

Print these variables to the console

- This will give you a better feel for the sensor data you are working with!

```
67
68
69 motors.enable(True)
70
71 # In case we start off the line, set small error to cause turn in big circle.
72 err = 1
73
74 while True:
75     # Read the sensors and display on LEDs
```

```
76      vals = ls.check(2000)
77      leds.ls(vals)
78
79      # Lookup error value (default to previous value)
80      err = ls_err.get(vals, err)
81
82      # Collect stats and apply controls
83      collect_stats(err)
84      apply_control(err)
```

Don't forget to call `collect_stats(err)` as the sensor values stream by!

```
85
```

Hint:

- **Fixing the *UnboundLocalError***

  This error occurs when a 🔧function tries to read a 🔧variable which has never been assigned to.

  - In Python, a variable is said to **"bind"** a *name* to a *value*. So "unbound" really means "no value assigned."
  - But you DID assign values! These are 🔧globals, right?!?

  If a function ever *assigns* to a variable, then it's considered a *local* variable by default.

  You have to explicitly *declare* those variables as global, using the `global` keyword:

  ```
  global t_prev, err_avg, err_trend
  ```

  Put that line at the start of your 🔧function, and the global 🔧variables will be used.

Goals:

- Follow the CodeTrek to add a `collect_stats()` function

  Encounter an *UnboundLocalError* when you RUN the code!

  Modify your code to fix the error. See the 🔧global tool and check Hints for guidance.

- **Print Stats to the Console**

  In your `collect_stats()` function, 🔧print milliseconds, current error, average error, and trend.

  - *(Print just the values separated by spaces)*

- Reach **Checkpoint 1**

  - Watch your **debug console** along the way!

Tools Found: Locals and Globals, Print Function, Variables, Functions, Constants

Solution:

```
1  from botcore import *
2  import time
3
4  SPEED = 50
5
6  # Dictionary mapping {sensors_tuple : error_int}
7  ls_err = {
8      (0,0,1,0,0) : 0,
9      (0,1,1,1,0) : 0,
10
11     (0,0,1,1,0) : 1,
```

```
12      (0,0,0,1,0) : 2,
13      (0,0,1,1,1) : 3,
14      (0,0,0,1,1) : 4,
15      (0,0,0,0,1) : 5,
16
17      (0,1,1,0,0) : -1,
18      (0,1,0,0,0) : -2,
19      (1,1,1,0,0) : -3,
20      (1,1,0,0,0) : -4,
21      (1,0,0,0,0) : -5,
22  }
23
24  def drive(speed, turn_ratio):
25      """Drive, using a fraction of the speed for turning."""
26      # speed: 0-100; turn_ratio: L=-1, R=+1, 0=straight
27      turn_spd = speed * turn_ratio
28      fwd_spd = speed - abs(turn_spd)
29      motors.run(LEFT, fwd_spd + turn_spd)
30      motors.run(RIGHT, fwd_spd - turn_spd)
31
32
33  def apply_control(err):
34      """Control steering based on error"""
35      Kp = 0.1    # Proportional factor
36      steering = err * Kp
37      drive(SPEED, steering)
38
39  # Error statistics (global state variables)
40  err_avg = 0
41  err_trend = 0
42  t_prev = 0  # ms time of previous sample
43
44  def collect_stats(err):
45      """Update global stats based on stream of err values"""
46      global t_prev, err_avg, err_trend
47      # Sensor only updates every 10 to 30ms
48      SAMPLE_INTERVAL = 10  # ms
49      HISTORY_FACTOR = 0.1  # Higher is more forgetful
50
51      t_now = time.ticks_ms()
52      if t_now - t_prev > SAMPLE_INTERVAL:
53          t_prev = t_now
54          err_avg_prev = err_avg
55          # Exponentially weighted moving average
56          err_avg = err * HISTORY_FACTOR + err_avg * (1 - HISTORY_FACTOR)
57          # Trend is based on last 2 average values
58          err_trend = (err_avg - err_avg_prev) * 10
59
60          # DEBUG - dump our variables to the console.
61          print(t_now, err, err_avg, err_trend)
62
63
64  motors.enable(True)
65
66  # In case we start off the line, set small error to cause turn in big circle.
67  err = 1
68
69  while True:
70      # Read the sensors and display on LEDs
71      vals = ls.check(2000)
72      leds.ls(vals)
73
74      # Lookup error value (default to previous value)
75      err = ls_err.get(vals, err)
76
77      # Collect stats and apply controls
78      collect_stats(err)
79      apply_control(err)
80
```

Quiz 4 **- Locals and Globals**

Question 1: What is the purpose of the `global` statement?

✓ To declare which 🔧variables inside a 🔧function should be treated as 🔧globals.

✗ To declare that your program can be used *internationally.*

✗ To declare that a 🔧function can be accessed *globally.*

Question 2: What is the output of the following?

```
name = 'Fred'

def foo():
    print(name)

foo()
```

✓ Fred

✗ UnboundLocalError

✗ Name?

Question 3: What is the output of the following?

```
name = 'Fred'

def foo():
    print(name, end=',')
    name = 'Jill'
    print(name)

foo()
```

✓ UnboundLocalError

✗ Fred,

✗ Jill

✗ Fred, Jill

Question 4: What is the output of the following?

```
name = 'Fred'

def foo():
    global name
    print(name, end=',')
    name = 'Jill'
    print(name)

foo()
```

✓ Fred,Jill

✗ UnboundLocalError

✗ Fred,Fred

✗ Jill,Jill

Objective 9 **- Line Drive!**

Line Drive!

All the pieces are now in place for your best *Line Following* algorithm yet!

Put it all together in your `apply_control()` function, bringing in *error statistics* for the remaining two terms in your **PID Controller** algorithm.

- As usual, the **CodeTrek** has the details!

Tuning the Constants

There are a number of 🔧constants in this version of the code. It's up to you to find the best settings for those, and that will take some *experimentation.*

- The ideal settings depend on what type of *Line Course* you are up against.
- Settings that give max speed on straightaways are usually not ideal for tight curves, and vice-versa!

To complete this Objective you will need to tune those constants to run the Classroom Course in top form.

CodeTrek:

```
1  from botcore import *
2  import time
3
4  SPEED = 70
5
6  # Dictionary mapping {sensors_tuple : error_int}
7  ls_err = {
8      (0,0,1,0,0) : 0,
9      (0,1,1,1,0) : 0,
10
11     (0,0,1,1,0) : 1,
12     (0,0,0,1,0) : 2,
13     (0,0,1,1,1) : 3,
14     (0,0,0,1,1) : 4,
15     (0,0,0,0,1) : 5,
16
17     (0,1,1,0,0) : -1,
18     (0,1,0,0,0) : -2,
19     (1,1,1,0,0) : -3,
20     (1,1,0,0,0) : -4,
21     (1,0,0,0,0) : -5,
22 }
23
24 def drive(speed, turn_ratio):
25     """Drive, using a fraction of the speed for turning."""
26     # speed: 0-100; turn_ratio: L=-1, R=+1, 0=straight
27     turn_spd = speed * turn_ratio
28     fwd_spd = speed - abs(turn_spd)
29     motors.run(LEFT, fwd_spd + turn_spd)
30     motors.run(RIGHT, fwd_spd - turn_spd)
31
32 def apply_control(err):
33     """Control steering based on error"""
34     Kp = 0.1  # Proportional factor (current error)
35     Ki = 0.01  # Integral factor (average error)
36     Kd = 0.07  # Derivative factor (trend)
37     steering = err * Kp + err_avg * Ki + err_trend * Kd
```

Add the 2 remaining *PID* terms

You already have the "P", now add the "I" and "D".

- This brings in *two* more constants: K_i and K_d
- Your `err_avg` and `err_trend` are the added control 🔧variables.

```
38
39     # Limit steering to +/- 1.0
40     if abs(steering) > 1:
41         steering = steering / abs(steering)
```

Limit the Turning Factor

Your `drive()` function expects a `turn_ratio` between -1.0 and +1.0

- Depending on the K-factors above, you could exceed this! *Make sure that can't happen.*

```
42
43      speed = SPEED
44      # Slow down when error grows
45      if abs(err_avg) > 0.1:
46          speed = speed * 0.6
```

Bonus Feature!

Tap the brakes if your error grows.

- You could get a lot more sophisticated with this, but even a simple speed decrease will allow your top speed to be much higher.

Note: You should really make new constants for these braking parameters, right?

- Maybe "BRAKING_THRESHOLD" and "BRAKING_SPEED_FACTOR"

  Even more parameters to tune...

```
47
48      drive(speed, steering)
49
50  # Error statistics (global state variables)
51  err_avg = 0
52  err_trend = 0
53  t_prev = 0  # ms time of previous sample
54
55  def collect_stats(err):
56      """Update global stats based on stream of err values"""
57      global t_prev, err_avg, err_trend
58      # Sensor only updates every 10 to 30ms
59      SAMPLE_INTERVAL = 10  # ms
60      HISTORY_FACTOR = 0.1  # Higher is more forgetful
61
62      t_now = time.ticks_ms()
63      if t_now - t_prev > SAMPLE_INTERVAL:
64          t_prev = t_now
65          err_avg_prev = err_avg
66          # Exponentially weighted moving average
67          err_avg = err * HISTORY_FACTOR + err_avg * (1 - HISTORY_FACTOR)
68          # Trend is based on last 2 average values
69          err_trend = (err_avg - err_avg_prev) * 10
70
71
72  motors.enable(True)
73
74  # In case we start off the line, set small error to cause turn in big circle.
75  err = 1
76
77  while True:
78      # Read the sensors and display on LEDs
79      vals = ls.check(2000)
80      leds.ls(vals)
81
82      # Lookup error value (default to previous value)
83      err = ls_err.get(vals, err)
84
85      # Collect stats and apply controls
86      collect_stats(err)
87      apply_control(err)
88
```

Goals:

- Complete your PID Controller by using `err_avg` and `err_trend` in your `apply_control()` function.

  - Reach **Checkpoint 1**
- Reach **CheckPoint 2**
- Reach **CheckPoint 3**
- Reach **CheckPoint 4**
- Reach **CheckPoint 5**
- Reach **CheckPoint 6**
- Complete the course within 5 minutes!

Tools Found: Constants, Variables

Solution:

```
1  from botcore import *
2  import time
3
4  SPEED = 70
5
6  # Dictionary mapping {sensors_tuple : error_int}
7  ls_err = {
8      (0,0,1,0,0) : 0,
9      (0,1,1,1,0) : 0,
10
11     (0,0,1,1,0) : 1,
12     (0,0,0,1,0) : 2,
13     (0,0,1,1,1) : 3,
14     (0,0,0,1,1) : 4,
15     (0,0,0,0,1) : 5,
16
17     (0,1,1,0,0) : -1,
18     (0,1,0,0,0) : -2,
19     (1,1,1,0,0) : -3,
20     (1,1,0,0,0) : -4,
21     (1,0,0,0,0) : -5,
22 }
23
24 def drive(speed, turn_ratio):
25     """Drive, using a fraction of the speed for turning."""
26     # speed: 0-100; turn_ratio: L=-1, R=+1, 0=straight
27     turn_spd = speed * turn_ratio
28     fwd_spd = speed - abs(turn_spd)
29     motors.run(LEFT, fwd_spd + turn_spd)
30     motors.run(RIGHT, fwd_spd - turn_spd)
31
32 def apply_control(err):
33     """Control steering based on error"""
34     Kp = 0.1  # Proportional factor (current error)
35     Ki = 0.01  # Integral factor (average error)
36     Kd = 0.07  # Derivative factor (trend)
37     steering = err * Kp + err_avg * Ki + err_trend * Kd
38
39     # Limit steering to +/- 1.0
40     if abs(steering) > 1:
41         steering = steering / abs(steering)
42
43     speed = SPEED
44     # Slow down when error grows
45     if abs(err_avg) > 0.5:
46         speed = speed * 0.6
47
48     drive(speed, steering)
49
50 # Error statistics (global state variables)
51 err_avg = 0
52 err_trend = 0
53 t_prev = 0  # ms time of previous sample
54
55 def collect_stats(err):
```

```
56      """Update global stats based on stream of err values"""
57      global t_prev, err_avg, err_trend
58      # Sensor only updates every 10 to 30ms
59      SAMPLE_INTERVAL = 10  # ms
60      HISTORY_FACTOR = 0.1  # Higher is more forgetful
61
62      t_now = time.ticks_ms()
63      if t_now - t_prev > SAMPLE_INTERVAL:
64          t_prev = t_now
65          err_avg_prev = err_avg
66          # Exponentially weighted moving average
67          err_avg = err * HISTORY_FACTOR + err_avg * (1 - HISTORY_FACTOR)
68          # Trend is based on last 2 average values
69          err_trend = (err_avg - err_avg_prev) * 10
70
71
72  motors.enable(True)
73
74  # In case we start off the line, set small error to cause turn in big circle.
75  err = 1
76
77  while True:
78      # Read the sensors and display on LEDs
79      vals = ls.check(2000)
80      leds.ls(vals)
81
82      # Lookup error value (default to previous value)
83      err = ls_err.get(vals, err)
84
85      # Collect stats and apply controls
86      collect_stats(err)
87      apply_control(err)
88
```

Mission 10 **- Fido Fetch**

Train your CodeBot to fetch using a dictionary of commands!

Objective 1 **- R-Ready**

The first step is to find out if Fido is online

Create a new file!

- Use the File → New File menu to create a new file called "fido.py".

Your goal is to train Fido to follow commands

- Fido will need to follow commands and explore the cafeteria to complete this mission.

Fido's first command is: `'status'`

- When you send the `'status'` command Fido will respond with `'r-ready'` if he is online.

Open the *Debug Console* Panel ☰

That's where you'll enter commands for Fido!

Follow the CodeTrek to check Fido's `'status'`

CodeTrek:

```
1  from botcore import *
2
```

```
3  response = 'r-ready'
```

Create a `response` 🔧variable to hold Fido's response to your command.

```
4
5  while True:
```

Use a 🔧loop to keep taking in new commands forever.

```
6      # wait for an input from the console
7      command = input("Input Command: ")
```

`input` a command for Fido with the 🔧REPL console window.

```
8
9      if command == 'status':
```

Use a 🔧branching `if` to perform an action if the command is `'status'`.

```
10         print(response)
```

Finally, add a `print` to output the `response` variable to the console!

Goals:

- Make a `command` 🔧variable that takes `input` from the console.
- **Open the *Debug Console* to interact with *Fido***

  Enter the `'status'` command.

  - Fido should output: `'r-ready'`

Tools Found: Variables, Loops, REPL, Branching

Solution:

```
1  from botcore import *
2
3  response = 'r-ready'
4
5  while True:
6      # wait for an input from the console
7      command = input("Input Command: ")
8
9      if command == 'status':
10         print(response)
```

***Objective 2* - Fido Speak**

Now teach Fido to Speak

The next command for Fido is: `'speak'`

- Your CodeBot should play a short tone when you command Fido to `'speak'`.
- After a short tone Fido should go silent.

Remember, to play a tone you should use the `spkr.pitch()` function

See the speaker tool for details.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
```

You will need a pause to let your speaker `'speak'`!

```
3
4  response = 'r-ready'
5  response2 = 440
```

`response2` is a different type for the `'speak'` command.

- `response2` is an integer.

```
6
7  while True:
8      # wait for an input from the console
9      command = input("Input Command: ")
10
11     if command == 'status':
12         print(response)
13
14     elif command == 'speak':
15         spkr.pitch(response2)
16         sleep(0.5)
17         spkr.off()
```

1. Set the speaker to `value2`
2. Delay for 0.5 seconds
3. Turn off the speaker

Goal:

- Enter the `'speak'` command 5 times to get Fido to bark.

Tools Found: Speaker, Time Module, Data Types, int

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  response = 'r-ready'
5  response2 = 440
6
7  while True:
8      # wait for an input from the console
9      command = input("Input Command: ")
10
11     if command == 'status':
12         print(response)
13
14     elif command == 'speak':
15         spkr.pitch(response2)
16         sleep(0.5)
17         spkr.off()
```

Objective 3 **- Organized Commands**

Fido will need quite a few more commands

That might get hard to keep track of

Is there an easy way to keep track of Fido's commands?

- Yes, of course! It's Python after all 😎

You can use a 🔧dictionary!

- A 🔧dictionary is a collection of **key : value** pairs.
- The **keys** can even be human readable 🔧strings like Fido's commands!

The dictionary below has a single key : value pair.

```
commands = {'status': 'r-ready'}
```

- `'status'` is the **key**
- `'r-ready'` is the **value** for the `'status'` key

You can use Fido's commands as keys in a 🔧dictionary.

A 🔧dictionary has many benefits:

1. It is super 🔧readable because it shows all the key:value mappings in one place.
2. It lets you 🔧iterate through all of the keys and/or values if needed.
3. You can store any Python 🔧type as the **value**.
4. Looking up values in a 🔧dictionary is more 🔧efficient for the 🔧CPU than searching through a list of items to find a match.

Use the CodeTrek to create a 🔧dictionary for tracking Fido's commands

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  # create a dictionary
5  commands = {
```

Create a new 🔧dictionary named `commands`.

```
6      'status': 'r-ready',
```

For the first element:

- The command (key) is 'status'
- The response (value) is 'r-ready'

This is a **key:value** pair!

```
7      'speak': 440,
```

You can create a 🔧dictionary on multiple lines for 🔧readability.

key:value pairs must be separated by commas.

A *style convention* is to place a comma after the last element in a multi-line dictionary or list.

- This comma is not required, but it makes it a little easier to add more lines later.

```
8  }
9
10 while True:
```

```
11      # wait for an input from the console
12      command = input("Input Command: ")
13
14      response = commands[command]
```

You can look up the **value** of a **key:value** pair like so:

- `value = dictionary[key]`

```
15
16      if command == 'status':
17          print(response)
```

The `value` in a **key:value** pair can be any type.

- This one is a string.
- The next command's response `value` is an integer.

A dictionary can contain a mix of many different data types. Even other dictionaries!!

```
18
19      elif command == 'speak':
20          spkr.pitch(response)
```

Don't forget to change `response2` to `response`.

```
21          sleep(0.5)
22          spkr.off()
23
```

Goals:

- Create a `commands` dictionary.
  - There are multiple ways to create a dictionary but use curly braces { } for now!
- Create a new `response` variable that gets a **value** from the `commands` dictionary.
- Check to see that Fido is still `'r-ready'` using the `'status'` command.

Tools Found: dictionary, str, Readability, Iterable, Data Types, Efficiency, CPU and Peripherals, Variables, int

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # create a dictionary
5  commands = {
6      'status': 'r-ready',
7      'speak': 440,
8  }
9
10 while True:
11     # wait for an input from the console
12     command = input("Input Command: ")
13
14     response = commands[command]
15
16     if command == 'status':
17         print(response)
18
19     elif command == 'speak':
20         spkr.pitch(response)
```

```
21          sleep(0.5)
22          spkr.off()
```

Quiz 1 - Efficiency

Question 1: Regarding efficiency of code, according to Sir Tony Hoare, *what* is the "root of all evil"?

- ✓ premature optimization
- ✗ optimization
- ✗ money
- ✗ Java

Question 2: A dictionary contains pairs of what?

- ✓ keys and values
- ✗ trousers
- ✗ lists and items
- ✗ keys
- ✗ values

Objective 4 - Fido Come

Time to teach Fido to: `'come'`

- Fido should move forward with both motors when you send the `'come'` command

Try adding the `'come'` command after the dictionary is already created.

- You can add a **key:value** pair with this format:
  - `dictionary['key'] = value`

This time make the value of the `'come'` key be a 2-item list

- The list should contain *speeds* for the left and right motors.
  - Index 0 will be the `LEFT` speed.
  - Index 1 will be the `RIGHT` speed.

```
commands['come'] = [30, 30]
```

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  #?? Enable the motors here
```

Enable your robo-companion's motors.

- This will allow you to change its motion with commands.
- See Hints if you need help!

```
5
6  # create a dictionary
7  commands = {
8      'status': 'r-ready',
```

```
9       'speak': 440,
10  }
11
12  # add new commands
13  commands['come'] = [30, 30]
```

Add a new `'come'` command to the `commands` 🔧dictionary.

- This is the format to add a new key:value pair
- The `'come'` command has a 🔧list value 🔧type

```
14
15  while True:
16      # wait for an input from the console
17      command = input("Input Command: ")
18
19      # use the command as the key
20      response = commands[command]
21
22      if command == 'status':
23          print(response)
24
25      elif command == 'speak':
26          spkr.pitch(response)
27          sleep(0.5)
28          spkr.off()
29
30      elif command == 'come':
31          # left is index 0, right is index 1
32          motors.run(LEFT, response[0])
33          motors.run(??)
```

Can you fill in the missing motor 🔧parameters?

1. Index 0 is the `LEFT` motor value
2. Index 1 is the `RIGHT` motor value

Hints:

- To enable the motors: `motors.enable(True)`
- **Remember to move both motors forward you need to:**
  1. Enable the motors: `'motors.enable(True)'`
  2. Set the value of the `'LEFT'` motor: `'motors.run(LEFT, 30)'`
  3. Set the value of the `'RIGHT'` motor: `'motors.run(RIGHT, 30)'`

Goals:

- Add the `'come'` command to the `commands` 🔧dictionary as a 🔧list of two numbers.
- Use the `'come'` command to get Fido to move forward and get a delicious treat!

Tools Found: dictionary, list, Motors, Data Types, Parameters, Arguments, and Returns

Solution:

```
1   from botcore import *
2   from time import sleep
3
4   motors.enable(True)
5
6   # create a dictionary
7   commands = {
8       'status': 'r-ready',
9       'speak': 440,
10  }
```

```
11
12  # add new commands
13  commands['come'] = [30, 30]
14
15  while True:
16      # wait for an input from the console
17      command = input("Input Command: ")
18
19      # use the command as the key
20      response = commands[command]
21
22      if command == 'status':
23          print(response)
24
25      elif command == 'speak':
26          spkr.pitch(response)
27          sleep(0.5)
28          spkr.off()
29
30      elif command == 'come':
31          # left is index 0, right is index 1
32          motors.run(LEFT, response[0])
33          motors.run(RIGHT, response[1])
```

***Objective 5* - Fido Stay**

Whoa, stay Fido stay!

Fido's next command is: `'stay'`

- Fido should stop moving when he gets the `'stay'` command.
- You still want to keep the motors *enabled* when you stop

For the `'stay'` command you should try something new

- Make the value for the `'stay'` **key:value** pair be a function

```
def fido_stay():
    ...

commands['stay'] = fido_stay
```

Wait, you can *do* that???

- Yes! A function is an object just like a string or list in Python.
    - And since a dictionary can store any type of value...

Pro Tip

Got *super fast* typing skills? For a *faster* way to send commands to Fido, use the REPL up/down arrow capability!

- Try using your keyboard ↑ and ↓ arrow keys to browse previous commands!
- Just hit **ENTER** when you want to execute a command.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5
6  def set_motors(left, right):
7      motors.run(LEFT, left)
8      motors.run(RIGHT, right)
```

Start with a new `set_motors` function.

- This lets you pass a `left` and `right` parameter to set both motors at the same time!

- You will have multiple functions that call this one!

```
9
10 # create a dictionary
11 commands = {
12     'status': 'r-ready',
13     'speak': 440,
14 }
15
16 def fido_stay():
17     set_motors(0, 0)
```

The `'stay'` command is simple, it just sets both motors to 0!

```
18
19 # add a new command
20 commands['come'] = [30, 30]
21
22 # add a function as a response
23 commands['stay'] = fido_stay
```

The **value** in a **key:value** pair can be a function.

- A function object can be passed around just like any other data type.

```
24
25 while True:
26     # wait for an input from the console
27     command = input("Input Command: ")
28
29     # use the command as the key
30     response = commands[command]
31
32     if command == 'status':
33         print(response)
34
35     elif command == 'speak':
36         spkr.pitch(response)
37         sleep(0.5)
38         spkr.off()
39
40     elif command == 'come':
41         # left is index 0, right is index 1
42         motors.run(LEFT, response[0])
43         motors.run(RIGHT, response[1])
44
45     elif command == 'stay':
46         # the response is a function
47         response()
```

The `response` for the `'stay'` command is a function.

- So you can call it just like you would any other function!!

```
48
```

Goals:

- Get Fido moving forward using the `'come'` command.
- Use the `'stay'` command after Fido is moving to get him to stop.

Tools Found: Functions, str, list, dictionary, Data Types, REPL

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5
6  def set_motors(left, right):
7      motors.run(LEFT, left)
8      motors.run(RIGHT, right)
9
10 # create a dictionary
11 commands = {
12     'status': 'r-ready',
13     'speak': 440,
14 }
15
16 def fido_stay():
17     set_motors(0, 0)
18
19 # add a new command
20 commands['come'] = [30, 30]
21
22 # add a function as a response
23 commands['stay'] = fido_stay
24
25 while True:
26     # wait for an input from the console
27     command = input("Input Command: ")
28
29     # use the command as the key
30     response = commands[command]
31
32     if command == 'status':
33         print(response)
34
35     elif command == 'speak':
36         spkr.pitch(response)
37         sleep(0.5)
38         spkr.off()
39
40     elif command == 'come':
41         # left is index 0, right is index 1
42         motors.run(LEFT, response[0])
43         motors.run(RIGHT, response[1])
44
45     elif command == 'stay':
46         # the response is a function
47         response()
```

***Objective 6* - Funky Fido**

Adding a function to the dictionary was awesome!

- It really shows the power of dictionaries!

Why don't you turn all of your commands into functions?

- This will help with readability big time!
- You can also remove all those messy branching statements.

When you make big changes to your code it is called *refactoring*.

- Go ahead and refactor your code now!
- But when you're done, remember to test everything.
  - It is easy to make mistakes when you refactor.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
```

```
5
6  def set_motors(left, right):
7      motors.run(LEFT, left)
8      motors.run(RIGHT, right)
9
10 # create a dictionary
11 commands = {}
```

Start with an empty dictionary so you can add all your commands the same way.

- You could also use this format: `commands = dict()`

```
12
13 # define a function for each command
14 def fido_status():
15     print("r-ready")
```

Now that you're using a function for the `"status"` command, the `"r-ready"` moves directly into the `print()` statement.

```
16
17 def fido_speak():
18     spkr.pitch(440)
```

Don't forget to add the pitch frequency.

- It moves out of the dictionary and into the function too!

```
19     sleep(0.5)
20     spkr.off()
21
22 def fido_come():
23     set_motors(30, 30)
24
25 def fido_stay():
26     set_motors(0, 0)
27
28 # add your commands
29 commands['status'] = fido_status
30 commands['speak'] = fido_speak
31 commands['come'] = fido_come
32 commands['stay'] = fido_stay
```

Now you can add all your commands in one place.

```
33
34 while True:
35     # wait for an input from the console
36     command = input("Input Command: ")
37
38     # use the command as the key
39     response = commands[command]
40
41     # the response is always a function
42     response()
```

The `response` will always be a function.

- You can call it like this: `response()`

Goals:

- Do not use any `if` or `elif` statements in your code.
- Define these functions in your code: `fido_status`, `fido_speak`, `fido_come`, `fido_stay`

- Command Fido to `'speak'` 3 times using your new `fido_speak` 🔧function.
- Command Fido to `'come'` forward using your new `fido_come` 🔧function.

Tools Found: Functions, dictionary, Readability, Branching, Refactoring

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5
6  def set_motors(left, right):
7      motors.run(LEFT, left)
8      motors.run(RIGHT, right)
9
10 # create a dictionary
11 commands = {}
12
13 # define a function for each command
14 def fido_status():
15     print("r-ready")
16
17 def fido_speak():
18     spkr.pitch(440)
19     sleep(0.5)
20     spkr.off()
21
22 def fido_come():
23     set_motors(30, 30)
24
25 def fido_stay():
26     set_motors(0, 0)
27
28 # add your commands
29 commands['status'] = fido_status
30 commands['speak'] = fido_speak
31 commands['come'] = fido_come
32 commands['stay'] = fido_stay
33
34 while True:
35     # wait for an input from the console
36     command = input("Input Command: ")
37
38     # use the command as the key
39     response = commands[command]
40
41     # the response is always a function
42     response()
```

***Objective 7* - Commands Help!**

There are so many commands now!

What if I have so many commands that I can't remember them?

- Time to add a `'help'` command to Fido

Help commands are very common in *command-line* programs like this.

- They give developers a list of options for the program.

You will need to 🔧iterate over the `commands` 🔧dictionary for your `'help'` command.

- There are multiple ways to 🔧iterate over a 🔧dictionary.
- You should try a `for` 🔧loop. It is super simple!

This will print all keys to the console:

```
for k in commands:
    print(k)
```

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5
6  def set_motors(left, right):
7      motors.run(LEFT, left)
8      motors.run(RIGHT, right)
9
10 # create a commands dictionary
11 commands = {}
12
13 # define a function for each command
14 def fido_status():
15     print("r-ready")
16
17 def fido_speak():
18     spkr.pitch(440)
19     sleep(0.5)
20     spkr.off()
21
22 def fido_come():
23     set_motors(30, 30)
24
25 def fido_stay():
26     set_motors(0, 0)
27
28 def fido_help():
29     print("Fido Commands:")
30     # loop through all the keys in the dictionary
31     for k in commands:
```

Use a `for` loop to iterate through all the *keys* in the `commands` dictionary.

```
32         # print every command to the console
33         print(k)
```

`print()` all **key** strings to the console window.

```
34
35 # add your commands
36 commands['status'] = fido_status
37 commands['speak'] = fido_speak
38 commands['come'] = fido_come
39 commands['stay'] = fido_stay
40 commands['help'] = fido_help

41
42 while True:
43     # wait for an input from the console
44     command = input("Input Command: ")
45
46     # use the command as the key
47     response = commands[command]
48
49     # the response is always a function
50     response()
```

Goals:

- Add a for loop to iterate through all the keys in the `commands` dictionary.
- Use the `'help'` command to `print` `'help'` to the console.

Tools Found: Iterable, dictionary, Loops

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5
6  def set_motors(left, right):
7      motors.run(LEFT, left)
8      motors.run(RIGHT, right)
9
10 # create a commands dictionary
11 commands = {}
12
13 # define a function for each command
14 def fido_status():
15     print("r-ready")
16
17 def fido_speak():
18     spkr.pitch(440)
19     sleep(0.5)
20     spkr.off()
21
22 def fido_come():
23     set_motors(30, 30)
24
25 def fido_stay():
26     set_motors(0, 0)
27
28 def fido_help():
29     print("Fido Commands:")
30     # Loop through all the keys in the dictionary
31     for k in commands:
32         # print every command to the console
33         print(k)
34
35 # add your commands
36 commands['status'] = fido_status
37 commands['speak'] = fido_speak
38 commands['come'] = fido_come
39 commands['stay'] = fido_stay
40 commands['help'] = fido_help
41
42 while True:
43     # wait for an input from the console
44     command = input("Input Command: ")
45
46     # use the command as the key
47     response = commands[command]
48
49     # the response is always a function
50     response()
```

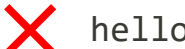
Quiz 2 **- Function Objects**

Question 1: Given the following program, what are 2 statements you could replace `# TODO` with that would result in "Hello, World" being printed?

```
def foo():
    print("Hello, World!")

hello = foo

# Say hello...
# TODO
```

✓ `foo()`

✓ `hello()`

 `hello`

`foo`

Question 2: What is printed by the following?

```
d = {'co': 16, 'ol': 25, 'bea': 32, 'ns': 41}
for k in d:
    print(k, end='')
```

✓ coolbeans

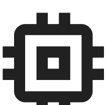
✗ 16253241

✗ co16ol25bea32ns41

✗ (co,16)(ol,25)(bea,32)(ns,41)

Objective 8 - Losing My Mind

Fido's memory is filling up!



What if you need to clear up some memory space?

Time to add a `'forget'` command.

- You can use this to make Fido forget a specified command.

To remove an item from a 🔧dictionary use the `del` keyword.

This code will remove the `'speak'` command:

```
del commands['speak']
```

For this objective you will also be forcing an `Error` to happen.

You are going to forget a command and then try to send it anyway.

- This will cause a `KeyError`.
- A `KeyError` happens when you try to use a **key** that doesn't exist in a dictionary.

  You may have seen this already if you mis-typed a command!

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5
6  def set_motors(left, right):
7      motors.run(LEFT, left)
8      motors.run(RIGHT, right)
9
10 # create a commands dictionary
11 commands = {}
12
13 # define a function for each command
14 def fido_status():
15     print("r-ready")
16
17 def fido_speak():
18     spkr.pitch(440)
```

```
19      sleep(0.5)
20      spkr.off()
21
22  def fido_come():
23      set_motors(30, 30)
24
25  def fido_stay():
26      set_motors(0, 0)
27
28  def fido_help():
29      print("Fido Commands:")
30      # Loop through all the keys in the dictionary
31      for k in commands:
32          # print every command to the console
33          print(k)
34
35  def fido_forget():
36      del_key = input("Command to Forget: ")
```

`input` the *command* (key) you want to remove from the dictionary.

```
37      del commands[del_key]
```

`del` is a Python keyword to remove a **key:value** pair from a dictionary.

```
38
39  # add your commands
40  commands['status'] = fido_status
41  commands['speak'] = fido_speak
42  commands['come'] = fido_come
43  commands['stay'] = fido_stay
44  commands['help'] = fido_help
45  commands['forget'] = fido_forget
46
47  while True:
48      # wait for an input from the console
49      command = input("Input Command: ")
50
51      # use the command as the key
52      response = commands[command]
53
54      # the response is always a function
55      response()
```

Goals:

- Add a `'forget'` command and `fido_forget` function to the `commands` dictionary **after** creating it.
- a) Use the `'forget'` command to `del` the `'speak'` command.

  b) Attempt to use the `'speak'` command after forgetting it

  - This must cause a `KeyError`

Tools Found: dictionary

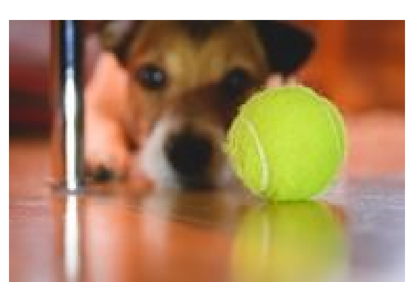
Solution:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5
6  def set_motors(left, right):
7      motors.run(LEFT, left)
8      motors.run(RIGHT, right)
```

```
9
10 # create a commands dictionary
11 commands = {}
12
13 # define a function for each command
14 def fido_status():
15     print("r-ready")
16
17 def fido_speak():
18     spkr.pitch(440)
19     sleep(0.5)
20     spkr.off()
21
22 def fido_come():
23     set_motors(30, 30)
24
25 def fido_stay():
26     set_motors(0, 0)
27
28 def fido_help():
29     print("Fido Commands:")
30     # Loop through all the keys in the dictionary
31     for k in commands:
32         # print every command to the console
33         print(k)
34
35 def fido_forget():
36     del_key = input("Command to Forget: ")
37     del commands[del_key]
38
39 # add your commands
40 commands['status'] = fido_status
41 commands['speak'] = fido_speak
42 commands['come'] = fido_come
43 commands['stay'] = fido_stay
44 commands['help'] = fido_help
45 commands['forget'] = fido_forget
46
47 while True:
48     # wait for an input from the console
49     command = input("Input Command: ")
50
51     # use the command as the key
52     response = commands[command]
53
54     # the response is always a function
55     response()
```

Objective 9 **- Hunting Treats**

Time for treats!



Now its time to send Fido out to explore!

- There are robo-dog treats scattered around the cafeteria
- Fido **WANTS TO EAT TREATS**
    - You should help him out!

Add a few more commands you might before you head out.

- Here are a few that could help:
    - `'left'`
    - `'right'`
    - `'back'`
    - `'fast'`

Check the hints if you get stuck!

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5
6  def set_motors(left, right):
7      motors.run(LEFT, left)
8      motors.run(RIGHT, right)
9
10 # create a commands dictionary
11 commands = {}
12
13 # define a function for each command
14 def fido_status():
15     print("r-ready")
16
17 def fido_speak():
18     spkr.pitch(440)
19     sleep(0.5)
20     spkr.off()
21
22 def fido_come():
23     set_motors(30, 30)
24
25 def fido_stay():
26     set_motors(0, 0)
27
28 def fido_help():
29     print("Fido Commands:")
30     # Loop through all the keys in the dictionary
31     for k in commands:
32         # print every command to the console
33         print(k)
34
35 def fido_forget():
36     del_key = input("Command to Forget: ")
37     del commands[del_key]
38
39 def fido_left():
40     set_motors(10, 30)
41
42 def fido_right():
43     set_motors(30, 10)
44
45 def fido_back():
46     set_motors(-30, -30)
47
48 def fido_fast():
49     set_motors(90, 90)
50
51 # add your commands
52 commands['status'] = fido_status
53 commands['speak'] = fido_speak
54 commands['come'] = fido_come
55 commands['stay'] = fido_stay
56 commands['help'] = fido_help
57 commands['forget'] = fido_forget
58 commands['left'] = fido_left
59 commands['right'] = fido_right
60 commands['back'] = fido_back
61 commands['fast'] = fido_fast
```

Add some new commands to Fido to allow you to navigate the cafeteria!

```
62
63 while True:
64     # wait for an input from the console
65     command = input("Input Command: ")
66
67     # use the command as the key
68     response = commands[command]
69
```

```
70      # the response is always a function
71      response()
```

Hints:

- Search near the tables for scraps left by students!
- You will need to get Fido moving `'fast'` (both motors >80%) to get a good **high-five**.
- ### Use the Keyboard Shortcuts

  Remember, your keyboard ↑ and ↓ arrow keys can recall previous commands!

  - Just hit **ENTER** when you want to execute a command.

  - ***Chase Camera** view is a good choice for this Objective*

Goals:

- Get a **high five** from Fido by lifting the front end of the CodeBot off the ground.
- Help Fido find at least 4 robo-dog treats hidden in the cafeteria.

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  motors.enable(True)
5
6  def set_motors(left, right):
7      motors.run(LEFT, left)
8      motors.run(RIGHT, right)
9
10 # create a commands dictionary
11 commands = {}
12
13 # define a function for each command
14 def fido_status():
15     print("r-ready")
16
17 def fido_speak():
18     spkr.pitch(440)
19     sleep(0.5)
20     spkr.off()
21
22 def fido_come():
23     set_motors(30, 30)
24
25 def fido_stay():
26     set_motors(0, 0)
27
28 def fido_help():
29     print("Fido Commands:")
30     # loop through all the keys in the dictionary
31     for k in commands:
32         # print every command to the console
33         print(k)
34
35 def fido_forget():
36     del_key = input("Command to Forget: ")
37     del commands[del_key]
38
39 def fido_left():
40     set_motors(10, 30)
41
42 def fido_right():
43     set_motors(30, 10)
44
45 def fido_back():
46     set_motors(-30, -30)
```

```
47
48 def fido_fast():
49     set_motors(90, 90)
50
51 # add your commands
52 commands['status'] = fido_status
53 commands['speak'] = fido_speak
54 commands['come'] = fido_come
55 commands['stay'] = fido_stay
56 commands['help'] = fido_help
57 commands['forget'] = fido_forget
58 commands['left'] = fido_left
59 commands['right'] = fido_right
60 commands['back'] = fido_back
61 commands['fast'] = fido_fast
62
63 while True:
64     # wait for an input from the console
65     command = input("Input Command: ")
66
67     # use the command as the key
68     response = commands[command]
69
70     # the response is always a function
71     response()
```

Mission 11 - Airfield Ops

Learn some unique programming concepts to help with airfield runway operations!

Objective 1 - Runway Centerline

The airfield manager needs you to:

- Clear the runway of hazards before any aircraft arrive!

It is critical that you stay on the runway!!!

- The manager will collect your CodeBot after the runway is clear.

You already wrote code to track a black line...

Do you think you can make your 'bot follow a dashed white line?

CodeTrek:

```
1  from botcore import *
2
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
5  SPEED_LIMIT = 0.5
```

Add a speed limit constant to make adjustments easily.

```
6
7  motors.enable(True)
8
9  def drive(left, right):
10     """Set both motors from -100% to +100%, with a speed limit."""
11     motors.run(LEFT, left * SPEED_LIMIT)
12     # TODO: Don't forget the RIGHT motor!
```

This function will save some typing later - you can set both motors in one step!

- Don't forget to activate the right motor as well!

```
13
14 def track_line(ls_vals):
```

```
15      """Drive based on sensor readings."""
16      if ls_vals == (1,0,0,0,0):
17          # TODO: drive(?, ?)
```

Line Following!

You will need to fill in the `drive()` calls based on which line sensors are detecting.

```
18      elif ls_vals == (1,1,0,0,0):
19          # TODO: drive(?, ?)
20      elif ls_vals == (0,1,0,0,0):
21          # TODO: drive(?, ?)
22      elif ls_vals == (0,1,1,0,0):
23          # TODO: drive(?, ?)
24      elif ls_vals == (0,0,1,0,0):
25          drive(100, 100)
```

FULL SPEED AHEAD when only the middle line sensor sees a white line!

```
26      elif ls_vals == (0,0,0,0,0):
27          # TODO: drive(?, ?)
```

No Line?

This happens when the CodeBot is between dashed-lines.

- You probably want to just keep driving straight!

```
28      elif ls_vals == (0,0,1,1,0):
29          # TODO: drive(?, ?)
30      elif ls_vals == (0,0,0,1,0):
31          # TODO: drive(?, ?)
32      elif ls_vals == (0,0,0,1,1):
33          # TODO: drive(?, ?)
34      elif ls_vals == (0,0,0,0,1):
35          # TODO: drive(?, ?)
36
37  # Main program loop
38  while True:
39      vals = ls.check(LS_THRESH, ???)
```

Your Main Loop

First thing is to read the line sensors.

- What does that second parameter do?

(By the way, the white center line IS reflective!)

```
40
41      # Turn on the line sensor leds
42      leds.ls(vals)
43
44      # Keep the bot on centerline
45      track_line(vals)
```

This calls a function to track the centerline based on line sensor values.

Goals:

- Cross the midway point of the runway on the centerline.
- Reach the end of the runway on the centerline.

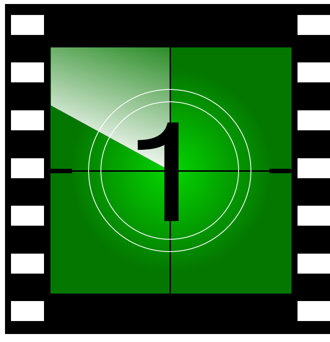
Tools Found: Line Sensors, Parameters, Arguments, and Returns, Constants

Solution:

```
from botcore import *

LS_THRESH = 2300
# Max speed = 1.0, 50% is 0.5
SPEED_LIMIT = 0.5

motors.enable(True)

def drive(left, right):
    motors.run(LEFT, left * SPEED_LIMIT)
    motors.run(RIGHT, right * SPEED_LIMIT)

def track_line(ls_vals):
    # Drive based on sensor readings.
    if ls_vals == (1,0,0,0,0):
        drive(-20, 50)
    elif ls_vals == (1,1,0,0,0):
        drive(0, 60)
    elif ls_vals == (0,1,0,0,0):
        drive(40, 80)
    elif ls_vals == (0,1,1,0,0):
        drive(80, 100)
    elif ls_vals == (0,0,1,0,0):
        drive(100, 100)
    elif ls_vals == (0,0,0,0,0):
        drive(100, 100)
    elif ls_vals == (0,0,1,1,0):
        drive(100, 80)
    elif ls_vals == (0,0,0,1,0):
        drive(80, 40)
    elif ls_vals == (0,0,0,1,1):
        drive(60, 0)
    elif ls_vals == (0,0,0,0,1):
        drive(50, -20)

while True:
    vals = ls.check(LS_THRESH, True)

    # Turn on the line sensor leds
    leds.ls(vals)

    # Keep the bot on centerline
    track_line(vals)
```

Objective 2 - Counting Lines

Dashed line counter!



The airfield manager noticed that your CodeBot ran off the end of the runway...

- They asked if you would please stop at the end next time.

But, how will CodeBot recognize the end?

If you knew how many dashed lines there were, you could stop when you sense the last one.

- Why don't you try counting the lines!?

Keeping Count

You may have guessed that your program will need a `count` 🔧variable that you will add 1 to every time you detect new *dash*.

- Adding 1 to a variable is called *"incrementing"*. There's also a name for subtracting 1: *"decrementing"*.
- So, the tricky part of this Objective is knowing when to ***increment*** your count!!

Watch It!

Once you have the count working properly, be sure to watch closely when your 'bot nears the end of the runway.

- Stop the program and check the `count`.
- You will need it for the next Objective!
- ...and if it goes too far, the other runway markings will mess up the count.

Note: *The runway is a little shorter this time, just to make it easier!*

CodeTrek:

```
1  from botcore import *
2
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
5  SPEED_LIMIT = 0.3
```

It is best if you slow down!

- This code *only* monitors the middle line sensor.
- Slowing down will help keep the sensor inside the white line.

```
6
7  # State variables
8  was_line = ?? # TODO: init this variable
```

The `was_line` variable "remembers" if you were on a line *the last time you checked*.

- Use this to detect the *start* of a dash!
- What should the *initial* boolean value be?
  - Well, your 'bot starts *before* the first line...

```
9  count = ??  # TODO: init this variable
```

The `count` variable will keep track of runway dashes.

- Before your bot starts moving, what should the *count* be?
- *(Hint: it's an integer)*

```
10
11 motors.enable(True)
12
13 def drive(left, right):
14     """Set both motors from -100% to +100%, with a speed limit."""
15     motors.run(LEFT, left * SPEED_LIMIT)
16     motors.run(RIGHT, right * SPEED_LIMIT)
17
18 def track_line(ls_vals):
19     """Drive based on sensor readings."""
20     if ls_vals == (1,0,0,0,0):
21         drive(-20, 50)
22     elif ls_vals == (1,1,0,0,0):
23         drive(0, 60)
24     elif ls_vals == (0,1,0,0,0):
25         drive(40, 80)
26     elif ls_vals == (0,1,1,0,0):
27         drive(80, 100)
28     elif ls_vals == (0,0,1,0,0):
29         drive(100, 100)
30     elif ls_vals == (0,0,0,0,0):
31         drive(100, 100)
32     elif ls_vals == (0,0,1,1,0):
33         drive(100, 80)
34     elif ls_vals == (0,0,0,1,0):
35         drive(80, 40)
36     elif ls_vals == (0,0,0,1,1):
37         drive(60, 0)
38     elif ls_vals == (0,0,0,0,1):
39         drive(50, -20)
```

```
40
41  # Main program loop
42  while True:
43      vals = ls.check(LS_THRESH, True)
44
45      # Turn on the line sensor leds
46      leds.ls(vals)
47
48      # Count the dashed lines
49      is_line = vals[2]
50      if is_line and not was_line:
51          # Beginning of a dash.
52          count = ??  # TODO: increment the count.
```

Remember, *increment* means **add 1** to `count`.

- Consider an augmented assignment statement here.

```
53          print("count = {}".format(count))
```

You can `print` to the console to help watch your `count`.

```
54
55          # Set the user LEDs to the count
56          # TODO
```

Pass your `count` value to `leds.user()`.

```
57
58      # Remember if we were on a line
59      was_line = is_line
```

Your counting is done for this loop.

- Better remember the `was_line` state for next time!

```
60
61      # Keep the bot on centerline
62      track_line(vals)
63
```

Goals:

- Display the "dash count" on the User LEDs (in binary of course).
    - Reach the ***middle of the runway*** with the correct count.
- Continue displaying the "dash count" on the **user LEDs**.
    - Reach the ***end of the runway*** with the correct count.

Tools Found: Variables, CodeBot LEDs, Binary Numbers, int, bool, Assignment

Solution:

```
1  from botcore import *
2
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
5  SPEED_LIMIT = 0.3
6
7  was_line = False
8  count = 0
```

```
9
10  motors.enable(True)
11
12  def drive(left, right):
13      motors.run(LEFT, left * SPEED_LIMIT)
14      motors.run(RIGHT, right * SPEED_LIMIT)
15
16  def track_line(ls_vals):
17      # Drive based on sensor readings.
18      if ls_vals == (1,0,0,0,0):
19          drive(-20, 50)
20      elif ls_vals == (1,1,0,0,0):
21          drive(0, 60)
22      elif ls_vals == (0,1,0,0,0):
23          drive(40, 80)
24      elif ls_vals == (0,1,1,0,0):
25          drive(80, 100)
26      elif ls_vals == (0,0,1,0,0):
27          drive(100, 100)
28      elif ls_vals == (0,0,0,0,0):
29          drive(100, 100)
30      elif ls_vals == (0,0,1,1,0):
31          drive(100, 80)
32      elif ls_vals == (0,0,0,1,0):
33          drive(80, 40)
34      elif ls_vals == (0,0,0,1,1):
35          drive(60, 0)
36      elif ls_vals == (0,0,0,0,1):
37          drive(50, -20)
38
39
40  while True:
41      vals = ls.check(LS_THRESH, True)
42
43      # Turn on the line sensor leds
44      leds.ls(vals)
45
46      is_line = vals[2]
47      if is_line and not was_line:
48          # Beginning of a dash
49          count += 1
50          print("count = {}".format(count))
51          leds.user(count)
52
53      was_line = is_line
54
55      # Keep the bot on centerline
56      track_line(vals)
57
```

Objective 3 - Stop at 09

Put it together!

I hope you wrote down the count from the previous objective!

- You know how many dashes there are on the runway...

So you have all the knowledge you need to stop CodeBot's motors when it gets to the end!

CodeTrek:

```
1  from botcore import *
2
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
```

```
5  SPEED_LIMIT = 0.3
6  TOTAL_LINES = ??  # TODO: Set the total number of lines, counted in last Objective.
```

Set this constant to the total number of dashes you counted.

- You'll be checking against this to see if you're at the end.

```
7
8  # Global state variables
9  was_line = False   # Was a line detected last time?
10 count = 0   # Current count
11
12 motors.enable(True)
13
14 def drive(left, right):
15     """Set both motors from -100% to +100%, with a speed limit."""
16     motors.run(LEFT, left * SPEED_LIMIT)
17     motors.run(RIGHT, right * SPEED_LIMIT)
18
19 def track_line(ls_vals):
20     """Drive based on sensor readings."""
21     if ls_vals == (1,0,0,0,0):
22         drive(-20, 50)
23     elif ls_vals == (1,1,0,0,0):
24         drive(0, 60)
25     elif ls_vals == (0,1,0,0,0):
26         drive(40, 80)
27     elif ls_vals == (0,1,1,0,0):
28         drive(80, 100)
29     elif ls_vals == (0,0,1,0,0):
30         drive(100, 100)
31     elif ls_vals == (0,0,0,0,0):
32         drive(100, 100)
33     elif ls_vals == (0,0,1,1,0):
34         drive(100, 80)
35     elif ls_vals == (0,0,0,1,0):
36         drive(80, 40)
37     elif ls_vals == (0,0,0,1,1):
38         drive(60, 0)
39     elif ls_vals == (0,0,0,0,1):
40         drive(50, -20)
41
42 # Main program loop
43 while True:
44     vals = ls.check(LS_THRESH, True)
45
46     # Turn on the line sensor leds
47     leds.ls(vals)
48
49     # Count the dashed lines
50     is_line = vals[2]
51     if is_line and not was_line:
52         # Beginning of a dash.
53         count = count + 1  # Increment the count.
54         print("count = {}".format(count))
55         leds.user(count)
56
57         # Stop the motors when we reach the last dash!
58         if ???:
59             # TODO: Stop the motors.
```

After you *increment* `count` and update the display,

- It's time to compare `count` against the `TOTAL_LINES`.
- And stop the motors immediately if CodeBot is at the end!

```
60
61
62     # Remember if we were on a line
63     was_line = is_line
64
```

```
65      # Keep the bot on centerline
66      track_line(vals)
```

Goals:

- Reach mid-field with an accurate count on the User LEDs.
- Set `motors.enable(False)` when your CodeBot senses the last dashed line.
  - Do not let your CodeBot go past the large **09** at the the end of the runway.

Tools Found: CodeBot LEDs, Constants, Comparison Operators, Motors

Solution:

```
1  from botcore import *
2
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
5  SPEED_LIMIT = 0.3
6  TOTAL_LINES = 83
7
8  line_detected = False
9  count = 0
10
11 motors.enable(True)
12
13 def drive(left, right):
14     motors.run(LEFT, left * SPEED_LIMIT)
15     motors.run(RIGHT, right * SPEED_LIMIT)
16
17 def track_line(ls_vals):
18     # Drive based on sensor readings.
19     if ls_vals == (1,0,0,0,0):
20         drive(-20, 50)
21     elif ls_vals == (1,1,0,0,0):
22         drive(0, 60)
23     elif ls_vals == (0,1,0,0,0):
24         drive(40, 80)
25     elif ls_vals == (0,1,1,0,0):
26         drive(80, 100)
27     elif ls_vals == (0,0,1,0,0):
28         drive(100, 100)
29     elif ls_vals == (0,0,0,0,0):
30         drive(100, 100)
31     elif ls_vals == (0,0,1,1,0):
32         drive(100, 80)
33     elif ls_vals == (0,0,0,1,0):
34         drive(80, 40)
35     elif ls_vals == (0,0,0,1,1):
36         drive(60, 0)
37     elif ls_vals == (0,0,0,0,1):
38         drive(50, -20)
39
40 while True:
41     vals = ls.check(LS_THRESH, True)
42
43     # Turn on the line sensor leds
44     leds.ls(vals)
45
46     # If the middle line sensor does not see a line
47     if vals[2] == 0:
48         line_detected = False
49
50     elif not line_detected:
51         line_detected = True
52         count = count + 1
53         print("count = {}".format(count))
54         leds.user(count)
55
```

```
56          if count == TOTAL_LINES:
57              motors.enable(False)
58
59      # Keep the bot on centerline
60      track_line(vals)
```

Objective 4 - Progress Bar

Need to See Some Progress

The airfield manager complained they can't see CodeBot's position at night.

- It's a *safety* issue!
- They want you to turn the User LEDs into a **progress bar**.

CodeBot Position	LEDs
Start of Runway	0b00000000
1 / 8 of Runway	0b00000001
2 / 8 of Runway	0b00000011
3 / 8 of Runway	0b00000111
4 / 8 of Runway	0b00001111
5 / 8 of Runway	0b00011111
5 / 8 of Runway	0b00111111
7 / 8 of Runway	0b01111111
End of Runway	0b11111111

There are pleny of ways to do this

- But you need to use the `//` Python operator.
    - OH NO...

Take a look at this code:

- It returns the number of *User LEDs* that should be ON for a given `count`

```
count = 87
TOTAL_LINES = 173
NUM_USER_LEDS = 8
num_leds_on = (count * NUM_USER_LEDS) // TOTAL_LINES
```

But what is the weird `//` symbol?

- That is the symbol for **Integer Division**

Integer Division divides by a number and then **rounds down** to an integer.

- You can learn more here: operators

CodeTrek:

```
1  from botcore import *
2
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
5  SPEED_LIMIT = 0.3
6  TOTAL_LINES = 83
7  NUM_USER_LEDS = 8
8
9  # Global state variables
10 was_line = False   # Was a line detected last time?
11 count = 0   # Current count
12
13 motors.enable(True)
14
15 def show_progress():
16     """Show progress down the runway on the User LEDs."""
17     num_leds_on = (count * NUM_USER_LEDS) // TOTAL_LINES
18     print("num_leds_on=", num_leds_on)
```

Progress Bar Function

This is where the action happens!

- Use the *integer division* operator to figure out the `num_leds_on`.
- print the value on the **console** too, so you can watch it run!

```
19        progress = [True] * num_leds_on
20        leds.user(progress)
```

A list of bools!

Did you know the User LEDs can be controlled with a list or tuple?

- You can use the *multiplication operator* on a list too!
- It works like *multiplication* of an int and a string.

```
21
22    def drive(left, right):
23        """Set both motors from -100% to +100%, with a speed limit."""
24        motors.run(LEFT, left * SPEED_LIMIT)
25        motors.run(RIGHT, right * SPEED_LIMIT)
26
27    def track_line(ls_vals):
28        # Drive based on sensor readings.
29        if ls_vals == (1,0,0,0,0):
30            drive(-20, 50)
31        elif ls_vals == (1,1,0,0,0):
32            drive(0, 60)
33        elif ls_vals == (0,1,0,0,0):
34            drive(40, 80)
35        elif ls_vals == (0,1,1,0,0):
36            drive(80, 100)
37        elif ls_vals == (0,0,1,0,0):
38            drive(100, 100)
39        elif ls_vals == (0,0,0,0,0):
40            drive(100, 100)
41        elif ls_vals == (0,0,1,1,0):
42            drive(100, 80)
43        elif ls_vals == (0,0,0,1,0):
44            drive(80, 40)
45        elif ls_vals == (0,0,0,1,1):
46            drive(60, 0)
47        elif ls_vals == (0,0,0,0,1):
48            drive(50, -20)
49
50    # Main program loop
51    while True:
52        vals = ls.check(LS_THRESH, True)
53
54        # Turn on the line sensor leds
55        leds.ls(vals)
56
57        # Count the dashed lines
58        is_line = vals[2]
59        if is_line and not was_line:
60            # Beginning of a dash.
61            count = count + 1  # Increment the count.
62            print("count = {}".format(count))
63            show_progress()
```

Replace the binary count display with your *progress bar* function call!

```
64
65            if count == TOTAL_LINES:
66                motors.enable(False)
67
68        # Remember if we were on a line
69        was_line = is_line
70
```

```
71      # Keep the bot on centerline
72      track_line(vals)
```

Goals:

- Set User LED "progress bar" as shown in Table.
  - Travel 2 / 8 down the runway → LEDs 0 and 1 on
- Set User LEDs as shown in Table.
  - Travel 4 / 8 down the runway → LEDs 0 - 3 on
- Set User LEDs as shown in Table.
  - Travel 6 / 8 down the runway → LEDs 0 - 5 on
- At the end of runway:
  - All User LEDs on
  - Set `motors.enable(False)`
- Use the `//` operator to calculate a variable called `num_leds_on`.

Tools Found: CodeBot LEDs, Math Operators, int, Binary Numbers, Print Function, list, bool, tuple, str

Solution:

```
1  from botcore import *
2
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
5  SPEED_LIMIT = 0.3
6  TOTAL_LINES = 83
7  NUM_USER_LEDS = 8
8
9  # Global state variables
10 was_line = False   # Was a line detected last time?
11 count = 0   # Current count
12
13 motors.enable(True)
14
15 def show_progress():
16     """Show progress down the runway on the User LEDs."""
17     num_leds_on = (count * NUM_USER_LEDS) // TOTAL_LINES
18     print("num_leds_on=", num_leds_on)
19     progress = [True] * num_leds_on
20     leds.user(progress)
21
22 def drive(left, right):
23     """Set both motors from -100% to +100%, with a speed limit."""
24     motors.run(LEFT, left * SPEED_LIMIT)
25     motors.run(RIGHT, right * SPEED_LIMIT)
26
27 def track_line(ls_vals):
28     # Drive based on sensor readings.
29     if ls_vals == (1,0,0,0,0):
30         drive(-20, 50)
31     elif ls_vals == (1,1,0,0,0):
32         drive(0, 60)
33     elif ls_vals == (0,1,0,0,0):
34         drive(40, 80)
35     elif ls_vals == (0,1,1,0,0):
36         drive(80, 100)
37     elif ls_vals == (0,0,1,0,0):
38         drive(100, 100)
39     elif ls_vals == (0,0,0,0,0):
40         drive(100, 100)
41     elif ls_vals == (0,0,1,1,0):
```

```
42            drive(100, 80)
43        elif ls_vals == (0,0,0,1,0):
44            drive(80, 40)
45        elif ls_vals == (0,0,0,1,1):
46            drive(60, 0)
47        elif ls_vals == (0,0,0,0,1):
48            drive(50, -20)
49
50    # Main program Loop
51    while True:
52        vals = ls.check(LS_THRESH, True)
53
54        # Turn on the Line sensor Leds
55        leds.ls(vals)
56
57        # Count the dashed Lines
58        is_line = vals[2]
59        if is_line and not was_line:
60            # Beginning of a dash.
61            count = count + 1  # Increment the count.
62            print("count = {}".format(count))
63            show_progress()
64
65            if count == TOTAL_LINES:
66                motors.enable(False)
67
68        # Remember if we were on a Line
69        was_line = is_line
70
71        # Keep the bot on centerLine
72        track_line(vals)
```

Objective 5 **- Scared Off**

That was a close one!

Did you see that wild animal dart across the runway?

Animal deterrence

The airfield manager asked us to use the CodeBot to deter wild animals from walking on the runway.

- Fortunately I know an expert in animal-robot relations!
- They recommended playing a *scary sound* every 8 dashed-lines for a duration of 3 dashes.

Detecting Every 8th Dash?

What's the best way in Python to detect every 8th occurrence? There are lots of ways, but for this Objective use the `%` 🔧operator !

The `%` symbol is called **modulo**.

- Sounds like a great Superhero name... or maybe a villain?
- It's nice, really! It gives the *remainder* from a division.

Seriously, *Fractions!?*

You may remember learning about the *remainder* when writing improper fractions as mixed numbers. Python can give you the *quotient* and *remainder* separately with `//` and `%` operators:

$$\frac{17}{5} = 3R2$$

```
# In Python:
17 // 5   # 3  (quotient)
17 % 5    # 2  (remainder)
```

> *Bear with me here. This math is useful for counting dashes!*

Counting Dashes with Modulo

You already have a `count` variable that increments with each dash.

- What if you did `count % 5` ?
- That would be **zero** every 5th dash!
  - Because the remainder is only *zero* when `count` is a multiple of `5`

Could your Python code detect `if count == 0`? *Of course it could!* Maybe now you have a good strategy for detecting every 8th dash :-)

CodeTrek:

```
1  from botcore import *
2
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
5  SPEED_LIMIT = 0.3
6  TOTAL_LINES = 83
7  NUM_USER_LEDS = 8
8
9  # Global state variables
10 was_line = False   # Was a line detected last time?
11 count = 0   # Current count
12
13 motors.enable(True)
14
15 def scary_sounds():
16     """Play a sound every 8th dash, for 3 dashes duration"""
17     remainder = count % 8
18     if remainder == 0:
19         # TODO: play speaker tone
```

Your Scary Sounds Function

Here's where you decide, based on `count`, if it's time to start or stop playing a sound.

- Start playing every 8th dash.
- Check the 🔧speaker tool for help with that.

```
20     elif remainder == 3:
21         # TODO: turn speaker off
```

Notice that `remainder` repeatedly cycles from 0 to 7 as `count` increases.

- Check it out in the *debugger* to confirm that!
- You are turning the speaker *ON* at 0. It plays for (0, 1, 2) - that's 3 dashes.
- So, turn it *OFF* at **3**.

```
22
23 def show_progress():
24     """Show progress down the runway on the User LEDs."""
25     num_leds_on = (count * NUM_USER_LEDS) // TOTAL_LINES
26     print("num_leds_on=", num_leds_on)
27     progress = [True] * num_leds_on
28     leds.user(progress)
29
30 def drive(left, right):
31     """Set both motors from -100% to +100%, with a speed limit."""
32     motors.run(LEFT, left * SPEED_LIMIT)
33     motors.run(RIGHT, right * SPEED_LIMIT)
34
35 def track_line(ls_vals):
36     # Drive based on sensor readings.
37     if ls_vals == (1,0,0,0,0):
38         drive(-20, 50)
39     elif ls_vals == (1,1,0,0,0):
40         drive(0, 60)
41     elif ls_vals == (0,1,0,0,0):
42         drive(40, 80)
43     elif ls_vals == (0,1,1,0,0):
44         drive(80, 100)
45     elif ls_vals == (0,0,1,0,0):
46         drive(100, 100)
```

```
47      elif ls_vals == (0,0,0,0,0):
48          drive(100, 100)
49      elif ls_vals == (0,0,1,1,0):
50          drive(100, 80)
51      elif ls_vals == (0,0,0,1,0):
52          drive(80, 40)
53      elif ls_vals == (0,0,0,1,1):
54          drive(60, 0)
55      elif ls_vals == (0,0,0,0,1):
56          drive(50, -20)
57
58  # Main program loop
59  while True:
60      vals = ls.check(LS_THRESH, True)
61
62      # Turn on the line sensor leds
63      leds.ls(vals)
64
65      # Count the dashed lines
66      is_line = vals[2]
67      if is_line and not was_line:
68          # Beginning of a dash.
69          count = count + 1  # Increment the count.
70          print("count = {}".format(count))
71          show_progress()
72          scary_sounds()
```

One more thing to check each time you update `count`

- This doesn't mean you play a sound every count...
- You're just giving this function a chance to decide if it's time to play or stop a sound.

```
73
74          if count == TOTAL_LINES:
75              motors.enable(False)
76
77      # Remember if we were on a line
78      was_line = is_line
79
80      # Keep the bot on centerline
81      track_line(vals)
```

Goals:

- Make your first scary sound at the 8th line
  - The sound should stay on for three lines and then stop
  - Line 7 = OFF, Line 8 = ON, Line 9 = ON, Line 10 = ON, Line 11 = OFF
- Make a scary sound every 8th line starting at the 8th line.
  - This will surely chase animals off the runway!
- At the last line:
  - Set all User LEDs on
  - Set `motors.enable(False)`
  - Turn the speaker off
- Use the `%` operator to calculate a variable called `remainder`.

Tools Found: Math Operators, Speaker

Solution:

```
1  from botcore import *
2
```

```
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
5  SPEED_LIMIT = 0.3
6  TOTAL_LINES = 83
7  NUM_USER_LEDS = 8
8
9  # Global state variables
10 was_line = False   # Was a line detected last time?
11 count = 0   # Current count
12
13 motors.enable(True)
14
15 def scary_sounds():
16     remainder = count % 8
17     if remainder == 0:
18         spkr.pitch(800)
19     elif remainder == 3:
20         spkr.off()
21
22 def show_progress():
23     """Show progress down the runway on the User LEDs."""
24     num_leds_on = (count * NUM_USER_LEDS) // TOTAL_LINES
25     print("num_leds_on=", num_leds_on)
26     progress = [True] * num_leds_on
27     leds.user(progress)
28
29 def drive(left, right):
30     """Set both motors from -100% to +100%, with a speed limit."""
31     motors.run(LEFT, left * SPEED_LIMIT)
32     motors.run(RIGHT, right * SPEED_LIMIT)
33
34 def track_line(ls_vals):
35     # Drive based on sensor readings.
36     if ls_vals == (1,0,0,0,0):
37         drive(-20, 50)
38     elif ls_vals == (1,1,0,0,0):
39         drive(0, 60)
40     elif ls_vals == (0,1,0,0,0):
41         drive(40, 80)
42     elif ls_vals == (0,1,1,0,0):
43         drive(80, 100)
44     elif ls_vals == (0,0,1,0,0):
45         drive(100, 100)
46     elif ls_vals == (0,0,0,0,0):
47         drive(100, 100)
48     elif ls_vals == (0,0,1,1,0):
49         drive(100, 80)
50     elif ls_vals == (0,0,0,1,0):
51         drive(80, 40)
52     elif ls_vals == (0,0,0,1,1):
53         drive(60, 0)
54     elif ls_vals == (0,0,0,0,1):
55         drive(50, -20)
56
57 # Main program loop
58 while True:
59     vals = ls.check(LS_THRESH, True)
60
61     # Turn on the line sensor leds
62     leds.ls(vals)
63
64     # Count the dashed lines
65     is_line = vals[2]
66     if is_line and not was_line:
67         # Beginning of a dash.
68         count = count + 1  # Increment the count.
69         print("count = {}".format(count))
70         show_progress()
71         scary_sounds()
72
73         if count == TOTAL_LINES:
74             motors.enable(False)
75
76     # Remember if we were on a line
77     was_line = is_line
```

```
78
79      # Keep the bot on centerline
80      track_line(vals)
```

Objective 6 - Pilot Math

One last task

The airfield manager has placed markers on the right side of the runway.

- A pilot uses them to tell how close they are to the end.
- Due to the nature of aircraft speed, the distance between the markers increases *exponentially.*

Check it out in the 3D view. There are 7 red markers numbered 1-7, positioned at dashes based on powers of 2:

Marker	1	2	3	4	5	6	7
Dash = 2^{marker}	2	4	8	16	32	64	128

The airfield manager wants to make sure the markers are placed the correct distances apart. They've asked if CodeBot can measure and signal with the Prox LEDs when it passes the marker positions.

- Starting at Marker number 2 (dash=4), turn the **Prox LEDs** ON at each red marker.
- Turn them OFF 3 dashes past each marker position.

Detecting the Red Markers

You are already tracking the `count` of dashes. Now you need to track the `next_marker` too!

- You will know you've reached a marker when the **dash count** = 2^{next_marker}.
- In Python that would look something like:

```
if count == 2**next_marker:
    leds.prox(0b11)  # Turn on both prox leds
```

That `**` symbol is one of Python's 🔧math operators, known as *exponentiation* or the *power* operator.

The Long Run!

By the way, you will be working with the *full-length runway* this time!

- That's `TOTAL_LINES = 173` if you're counting!

CodeTrek:

```
1  from botcore import *
2
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
5  SPEED_LIMIT = 0.3
6  TOTAL_LINES = 173
```

There's a longer runway this time!

- Be sure to update the total number of dashes.
- Otherwise your 'bot will stop short!

```
7  NUM_USER_LEDS = 8
8
9  # Global state variables
10 was_line = False   # Was a line detected last time?
11 count = 0   # Current count
12 next_marker = 2
```

A 🔧global variable to keep track of the *next marker* you're looking for.

- Start with Marker **2** as instructed.

```
13
14  motors.enable(True)
15
16  def check_markers():
17      """Turn on the Prox LEDs when a Marker is reached."""
18      global next_marker
19
20      # Markers are positioned at dashes with powers of 2 down the runway
21      marker_dash = 2**next_marker
22      print("next marker dash = ", marker_dash)
```

A function to Check for Markers

Use the *exponentiation* 🔧operator to raise 2 to a power.

- Which marker do you seek?

$$marker_dash = 2^{next_marker}$$

```
23
24      # Check to see if our count has reached the next marker
25      if count == marker_dash:
26          leds.prox(3)  # Both LEDs on
```

Reached a Marker!

Turn on *both* **Prox LEDs**.

- You can turn the Prox LEDs on with the function `leds.prox(b)`.

Parameter `b` is a binary value!

```
b = 0 # Turns off both prox lights!
b = 1 # Turns the left prox light on!
b = 2 # Turns the right prox light on!
b = 3 # Turns both prox lights on!

leds.prox(b)
```

```
27      elif count == marker_dash + 3:
28          leds.prox(0)  # LEDs off
29          next_marker += 1  # Ready for next marker number
```

The `count` is ***3 dashes*** beyond the Marker.

- Time to turn the LEDs off!
- Also, begin looking for the *next* marker in sequence.

```
30
31  def scary_sounds():
32      remainder = count % 8
33      if remainder == 0:
34          spkr.pitch(800)
35      elif remainder == 3:
36          spkr.off()
37
38  def show_progress():
39      """Show progress down the runway on the User LEDs."""
40      num_leds_on = (count * NUM_USER_LEDS) // TOTAL_LINES
41      print("num_leds_on=", num_leds_on)
42      progress = [True] * num_leds_on
43      leds.user(progress)
44
45  def drive(left, right):
46      """Set both motors from -100% to +100%, with a speed limit."""
47      motors.run(LEFT, left * SPEED_LIMIT)
48      motors.run(RIGHT, right * SPEED_LIMIT)
49
50  def track_line(ls_vals):
51      # Drive based on sensor readings.
52      if ls_vals == (1,0,0,0,0):
```

```
53          drive(-20, 50)
54      elif ls_vals == (1,1,0,0,0):
55          drive(0, 60)
56      elif ls_vals == (0,1,0,0,0):
57          drive(40, 80)
58      elif ls_vals == (0,1,1,0,0):
59          drive(80, 100)
60      elif ls_vals == (0,0,1,0,0):
61          drive(100, 100)
62      elif ls_vals == (0,0,0,0,0):
63          drive(100, 100)
64      elif ls_vals == (0,0,1,1,0):
65          drive(100, 80)
66      elif ls_vals == (0,0,0,1,0):
67          drive(80, 40)
68      elif ls_vals == (0,0,0,1,1):
69          drive(60, 0)
70      elif ls_vals == (0,0,0,0,1):
71          drive(50, -20)
72
73  # Main program loop
74  while True:
75      vals = ls.check(LS_THRESH, True)
76
77      # Turn on the line sensor leds
78      leds.ls(vals)
79
80      # Count the dashed lines
81      is_line = vals[2]
82      if is_line and not was_line:
83          # Beginning of a dash.
84          count = count + 1  # Increment the count.
85          print("count = {}".format(count))
86          show_progress()
87          scary_sounds()
88          check_markers()
```

When a Dash begins, the action happens!

- One more function call for your *dash-detection-duties.*

```
89
90          if count == TOTAL_LINES:
91              motors.enable(False)
92
93      # Remember if we were on a line
94      was_line = is_line
95
96      # Keep the bot on centerline
97      track_line(vals)
98
```

Goals:

- Make a sound every 8th line starting at the 8th line.
  - The sound should stay on for 3 lines.
- Turn the prox leds on at lines that are powers of 2 starting at Marker 2 (4th dash).
  - Set `leds.prox(0)` 3 lines later
- At the last line:
  - All user LEDs on and all prox LEDs off
  - Set `motors.enable(False)`
  - Turn the speaker off
- Use the `**` operator to calculate a *power of 2.*

Tools Found: Math Operators, Locals and Globals, Functions

Solution:

```
1  from botcore import *
2
3  LS_THRESH = 2300
4  # Max speed = 1.0, 50% is 0.5
5  SPEED_LIMIT = 0.3
6  TOTAL_LINES = 173
7  NUM_USER_LEDS = 8
8
9  # Global state variables
10 was_line = False   # Was a line detected last time?
11 count = 0   # Current count
12 next_marker = 2
13
14 motors.enable(True)
15
16 def check_markers():
17     global next_marker
18     marker_dash = 2**next_marker
19     print("next marker dash = ", marker_dash)
20     if count == marker_dash:
21         leds.prox(3)
22     elif count == marker_dash + 3:
23         leds.prox(0)
24         next_marker += 1
25
26 def scary_sounds():
27     remainder = count % 8
28     if remainder == 0:
29         spkr.pitch(800)
30     elif remainder == 3:
31         spkr.off()
32
33 def show_progress():
34     """Show progress down the runway on the User LEDs."""
35     num_leds_on = (count * NUM_USER_LEDS) // TOTAL_LINES
36     print("num_leds_on=", num_leds_on)
37     progress = [True] * num_leds_on
38     leds.user(progress)
39
40 def drive(left, right):
41     """Set both motors from -100% to +100%, with a speed limit."""
42     motors.run(LEFT, left * SPEED_LIMIT)
43     motors.run(RIGHT, right * SPEED_LIMIT)
44
45 def track_line(ls_vals):
46     # Drive based on sensor readings.
47     if ls_vals == (1,0,0,0,0):
48         drive(-20, 50)
49     elif ls_vals == (1,1,0,0,0):
50         drive(0, 60)
51     elif ls_vals == (0,1,0,0,0):
52         drive(40, 80)
53     elif ls_vals == (0,1,1,0,0):
54         drive(80, 100)
55     elif ls_vals == (0,0,1,0,0):
56         drive(100, 100)
57     elif ls_vals == (0,0,0,0,0):
58         drive(100, 100)
59     elif ls_vals == (0,0,1,1,0):
60         drive(100, 80)
61     elif ls_vals == (0,0,0,1,0):
62         drive(80, 40)
63     elif ls_vals == (0,0,0,1,1):
64         drive(60, 0)
65     elif ls_vals == (0,0,0,0,1):
66         drive(50, -20)
67
68 # Main program loop
```

```
69 while True:
70     vals = ls.check(LS_THRESH, True)
71 
72     # Turn on the line sensor leds
73     leds.ls(vals)
74 
75     # Count the dashed lines
76     is_line = vals[2]
77     if is_line and not was_line:
78         # Beginning of a dash.
79         count = count + 1  # Increment the count.
80         print("count = {}".format(count))
81         show_progress()
82         scary_sounds()
83         check_markers()
84 
85         if count == TOTAL_LINES:
86             motors.enable(False)
87 
88     # Remember if we were on a line
89     was_line = is_line
90 
91     # Keep the bot on centerline
92     track_line(vals)
```

Quiz 1 - Fly with Python

Question 1: What's `1 // 2` ?

✓ 0

✗ 0.5

✗ 1

Question 2: What's `5 % 8` ?

✓ 5

✗ 0

✗ 3

Question 3: What's `4 % 3` ?

✓ 1

✗ 0

✗ 7

Question 4: What's `10 ** 2` ?

✓ 100

✗ 1024

✗ 20

Mission 12 - King of the Hill

Harness the CodeBot's accelerometer to climb to the top of a mountain!

***Objective 1* - Looking Up**

Introducing the 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 *three-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* $9.8 m/s^2$*)*.
- 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)

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:

```
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, so 65,535 (2^{16}) possibilities.
    - The max positive 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 "looking_up.py"

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  # Enable the motors
5  motors.enable(True)
6
7  # Set both motors to 40%
8  motors.run(LEFT, 40)
9  motors.run(RIGHT, 40)
10
11 # Loop forever
12 while True:
13     # Read the accelerometer X, Y and Z values
14     x, y, z = accel.read()
```

`accel.read()` returns 3 values in a tuple.

Assigning `x, y, z` unpacks the tuple and assigns each of the three values to individual variables all in just one line of code!

```
15
16      # Print raw values
17      print(x, y, z, sep=???) # TODO: fix separator string!
```

print your variables as 3 integers, separated by commas.

- What should `sep=???` really be?

```
18
19      # Print values as a list
20      print([x, y, z])
```

Create a list by using *square brackets*

- The `print()` statement will convert this to a properly-formatted string.

```
21
22      # Print values as a tuple
23      print((x, y, z))
```

Create a tuple by using *parentheses*

- The `print()` statement will convert this to a properly-formatted string.

Look odd to see double-nested parentheses? *I agree!*

- Check out what's happening though. The *outer* parentheses are required for arguments to the `print()` function. The *inner* parenthesis are making a tuple!

```
24
25      # Print values in a formatted string
26      accel.dump_axes()
```

The accelerometer function `dump_axes()` prints a string in the console window formatted like this `X=NN, Y=NN, Z=NN`.

```
27
28      sleep(0.1)
```

The `sleep(0.1)` slows down the output to the console window, making it easier to read.

```
29
30      print('----------------')
31
```

Goals:

- Write an infinite loop that: *Reads* the values of the Accelerometer with `accel.read()`
- Print the x, y, z values of the Accelerometer *three ways* in the following order:
  - comma-separated integers *ex: 1, 2, 3*
  - a list *ex: [1, 2, 3]*
  - a tuple *ex: (1, 2, 3)*
- Print the Accelerometer values with `accel.dump_axes()`

Tools Found: Accelerometer, import, tuple, int, float, list, Assignment, Print Function, Keyword and Positional Arguments

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # Enable the motors
5  motors.enable(True)
6
7  # Set both motors to 40%
8  motors.run(LEFT, 40)
9  motors.run(RIGHT, 40)
10
11 # Loop forever
12 while True:
13     # Read the accelerometer X, Y and Z values
14     x, y, z = accel.read() #@1
15
16     # Print raw values
17     print(x, y, z, sep=',') #@2
18
19     # Print values as a list
20     print([x, y, z]) #@3
21
22     # Print values as a tuple
23     print((x, y, z)) #@4
24
25     # Print values in a formatted string
26     accel.dump_axes() #@5
27
28     sleep(0.1) #@6
29
30     print('---------------')
31
```

Quiz 1 - Get Down With Gravity

Question 1: What would the 'Z' value of the accelerometer read if CodeBot's circuit board was sitting horizontally flat?

- That's 1g of gravity pulling straight down!

✓ -16383

✗ 32767

✗ 1

✗ +16383

Question 2: How would you print the values of -679, 4093, -15850 in a tuple?

✓ `print((-679, 4093, -15850))`

✗ `print(-679, 4093, -15850)`

✗ `print([-679, 4093, -15850])`

Objective 2 - Level With Me

Get your climbing gear in order...

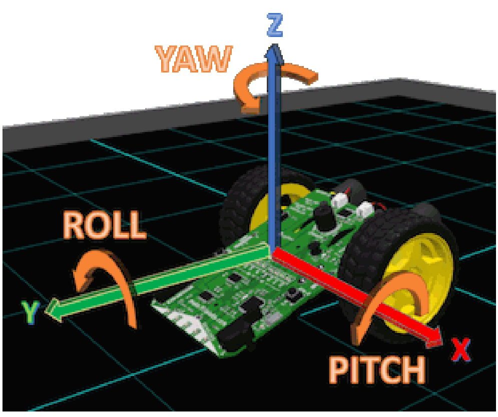
Before you climb a mountain, be sure you have all the tools you need to make it to the top!

While you're on level ground, convert CodeBot's 🔧Accelerometer data into something you can work with.

- The raw numbers it gives by default tell you something about *gravity*, but for **climbing** you need to figure out the ***angles!***
- Is your 'bot on *level ground* or is it on a *steep incline?*

Get ready to convert the accelerometer data into some angles! Here are the *principal axes* used to navigate ships, aircraft, and *robots:*

Pitch, Roll, and Yaw



What's Your Angle?

Now...the geometry of CodeBot's chassis makes this a little more complex. Have you noticed that the CodeBot's nose points downward?

- This design, while incredibly sporty and fun looking, is going to cost you a bit in added calculations.
- Before you can determine the true angles of the plywood mountain you plan to conquer, you're going to have to compensate for that oh-so-cool, signature CodeBot *slant!*

Use the 🔧Accelerometer to find out exactly how many degrees the CodeBot's nose is pointing down while your 'bot is on a flat surface.

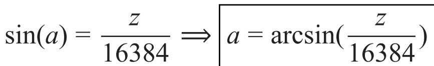
- As you can see from the picture above, the *accelerometer's* Y-axis will be tilted down a little in the Z direction.
- Gravity is pulling in the Z-axis, so the accelerometer tells you the *Z component* of its internal axes.

Gravity Vector

Check out this triangle! The yellow arrow is CodeBot's slant, shown as a *vector.*

- Whent it's horizontal there's no Z component, so your accelerometer-Y reads 0.
- If it's pointed straight down (a = 90°) it's all Z, so you'd read 16384.
- At an angle 'a' it makes a triangle with **both** Y *and* Z components.

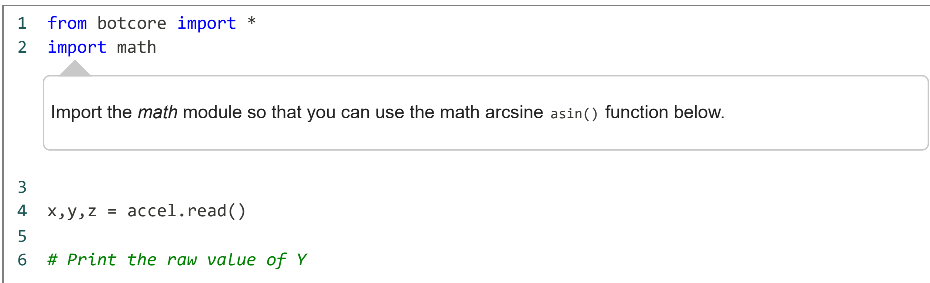
Python's 🔧math module provides functions to calculate angles and sides of triangles.



$$\sin(a) = \frac{z}{16384} \implies \boxed{a = \arcsin(\frac{z}{16384})}$$

- Wait... how did they know we'd need to calculate *gravity vectors?* What *dark* ***code*** is this?
- Relax, it's just *trigonometry* baby!

CodeTrek:



```
1  from botcore import *
2  import math
```

Import the *math* module so that you can use the math arcsine `asin()` function below.

```
3
4  x,y,z = accel.read()
5
6  # Print the raw value of Y
```

```
7  print("y = {}".format(y))
```

Print the "raw" value.

- The accelerometer tells you what component of its Y-axis is pointed down.

```
8
9  # Calculate the pitch angle in radians
10 pitch = math.asin(y / 16384)
```

Calculate the pitch angle.

- Note the `math` functions work in *radians*.

The "magic number" 16384 is what your accelerometer reads for 1g or one earth's gravitational acceleration, so it's the *hypotenuse* of the vector triangle.

```
11
12 print("pitch = {} radians".format(pitch))
13
14 # Convert the pitch angle from radians to degrees
15 pitch = pitch * 180 / math.pi
```

Convert radians to degrees:

$$deg = rad \cdot \frac{180°}{\pi}$$

```
16
17 print("pitch = {} degrees".format(pitch))
18
```

Goals:

- Print the raw Y-axis value to the console: `y = XXX`
  - Use the `format()` method to format your string for printing (see string formatting)
- Calculate the pitch angle using the math module function `asin()`
  - This is the $\arcsin()$ trig function, which gives the angle in *radians*.
  - Format and print the value: `pitch = XXX radians`
- Convert the pitch angle to degrees: $deg = rad \cdot \frac{180°}{\pi}$
  - Format and print the value: `pitch = XXX degrees`

Tools Found: Accelerometer, Math Module, String Formatting

Solution:

```
1  from botcore import *
2  import math
3
4  x,y,z = accel.read()
5
6  # Print the raw value of Y
7  print("y = {}".format(y))
8
9  # Calculate the pitch angle in radians
10 pitch = math.asin(y / 16384)
11
12 print("pitch = {} radians".format(pitch))
13
```

```
14  # Convert the pitch angle from radians to degrees
15  pitch = pitch * 180 / math.pi
16
17  print("pitch = {} degrees".format(pitch))
18
```

Objective 3 - Off-Roading

Pitch Perfect

Now that you've measured your "level" pitch angle, it's time to put that information to use and start climbing some slopes! *Gotta get your 'bot moving up the mountain to check those angles.*

Hmm, if only you had a mountain to climb...

You DO have a mountain to climb!

So here goes. It's time to climb this mountain! To start with, you are going to drive this thing *manually!*

- Did you know that in the simulator you can use the 0 and 1 keys on your keyboard to activate the Codebot's BTN-0 and BTN-1 🔧buttons?
- With this knowledge, all you need to do is check the state of those 🔧buttons in an infinite `while` 🔧loop to release the power to drive this 'bot *like the wind,* simply using your keyboard.

One last thing...

In previous objectives, you may have noticed that your 'bot is POWERFUL!

- If you start out with full power, the nose pops a wheelie.
  - Kind of cool, right? *But not for climbing mountains.*

Don't flip out here.

To keep things more grounded, you need to accelerate more slowly.

- Write some code to accelerate and decelerate CodeBot more gradually, as you control it with the buttons.

Alright, Drive On!

CodeTrek:

```
1  from botcore import *
2  import math
3  from time import sleep_ms
4
5  # Constants
6  SPEED_LIMIT = 70
7  CODEBOT_SLANT = 20  # Measured earlier
```

The 🔧constant value of CodeBot's `pitch`.

- This is the value you measured in the previous Objective!
- You'll need to subtract this from the accelerometer reading to "level it out".

```
8  ONE_G = 16384
```

Another 🔧constant, `16384` is equal to 1g of gravitational acceleration.

- This is a property of the 🔧accelerometer you learned in the first Objective.

```
9
10  # Global variables for motor power
11  left_power = 0
12  right_power = 0
13
```

```
14  # Enable motors
15  motors.enable(True)
16
17  def drive_bot():
18      """Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)"""
19      # TODO: Do we need code here to declare some globals?
```

A 🔧function to drive the 'bot using the 🔧buttons

- It gradually increaseds or decreases the 🔧global *motor power* variables above.

There may be a bug here!

```
20
21      # Accelerate slowly if button is pressed
22      if buttons.is_pressed(LEFT):
23          if left_power < SPEED_LIMIT:
24              left_power = left_power + 1
25      elif left_power > 1:
26          # Decelerate if button not pressed
27          left_power = left_power - 2
```

Check your buttons!

- If the button is pressed, **increase** power!
- If not, **decrease**. But don't let power go below zero. *(no reverse!)*

AND enforce a `SPEED_LIMIT` so it doesn't get too crazy in here.

```
28
29      # Accelerate slowly if button is pressed
30      if buttons.is_pressed(RIGHT):
31          if right_power < SPEED_LIMIT:
32              right_power = right_power + 1
33      elif right_power > 1:
34          # Decelerate if button not pressed
35          right_power = right_power - 2
```

Same deal for the RIGHT motor!

```
36
37      # Apply the power!
38      motors.run(LEFT, left_power)
39      motors.run(RIGHT, right_power)
```

Finally, send those calculated *power* levels to the 🔧motors!

```
40
41  def get_pitch():
42      """Get the current pitch angle of the platform in degrees"""
43      # Read the raw accelerometer data
44      x, y, z = accel.read()
45
46      # Calculate pitch and convert angle to degrees
47      pitch = math.asin(y / ONE_G)
48      pitch = pitch * 180 / math.pi
```

Fancy Maths

- These are the *pitch* calculations you worked out in the last Objective.

```
49
50      # Subtract CodeBot slant
51      pitch = pitch - CODEBOT_SLANT
```

Level it out!

- This should make the `pitch` zero when you're on level ground.

```
52
53      # Make "looking up" a positive angle
54      pitch = -pitch
```

Defy Gravity!

The accelerometer gives *positive* values in the direction of acceleration.

- And *gravity* is accelerating CodeBot toward the *center of the Earth!*
- But humans prefer to consider UP as positive... so **flip the sign**!

```
55      # Round to the nearest integer
56      # TODO: ...what was that "round" function called?
#@10
57
58      return pitch
59
60  # Main Loop
61  while True:
62      # Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)
63      drive_bot()
64
65      # Get the current pitch angle
66      pitch = get_pitch()
67
68      # Print the pitch angle to the console window
69      print("Pitch: ", pitch)
70
71      # Slow down the display for better readability
72      sleep_ms(50)
#@11
73
```

Hints:

- In order for the simulation to capture the 0 and 1 keys on your keyboard, make sure you focus the sim by *clicking in the **3D View*** after you run your code!
- See the built-ins for a function to `round()` your pitch.
- You are going to need to declare those global variables inside the `drive_bot()` function.

Goals:

- Print the CodeBot's `pitch` in a while loop in the format `Pitch: XX`.
  - Pitch value must be rounded to the nearest degree.
  - Nose pointing up should be *positive.*
- Use BTN0 and BTN1 to drive CodeBot up the mountain!
  - I want to see that `pitch` angle INCREASE as your nose points to the sky.
  - Show me at least 30° of *pitch* please!

Tools Found: Buttons, Loops, Constants, Accelerometer, Functions, Locals and Globals, Motors, Built-In Functions

Solution:

```
1  from botcore import *
2  import math
3  from time import sleep_ms
4
5  # Constants
6  SPEED_LIMIT = 70
```

```
7  CODEBOT_SLANT = 20  # Measured earlier
8  ONE_G = 16384
9
10 # Global variables
11 left_power = 0
12 right_power = 0
13
14 # Enable motors
15 motors.enable(True)
16
17 def drive_bot():
18     """Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)"""
19     global left_power, right_power
20
21     # Accelerate slowly if button is pressed
22     if buttons.is_pressed(LEFT):
23         if left_power < SPEED_LIMIT:
24             left_power = left_power + 1
25     elif left_power > 1:
26         # Decelerate if button not pressed
27         left_power = left_power - 2
28
29     # Accelerate slowly if button is pressed
30     if buttons.is_pressed(RIGHT):
31         if right_power < SPEED_LIMIT:
32             right_power = right_power + 1
33     elif right_power > 1:
34         # Decelerate if button not pressed
35         right_power = right_power - 2
36
37     # Apply the power!
38     motors.run(LEFT, left_power)
39     motors.run(RIGHT, right_power)
40
41 def get_pitch():
42     """Get the current pitch angle of the platform in degrees"""
43     # Read the raw accelerometer data
44     x, y, z = accel.read()
45
46     # Calculate pitch and convert angle to degrees
47     pitch = math.asin(y / ONE_G)
48     pitch = pitch * 180 / math.pi
49
50     # Subtract CodeBot slant
51     pitch = pitch - CODEBOT_SLANT
52
53     # Make "looking up" a positive angle
54     pitch = -pitch
55
56     # Round to the nearest integer
57     pitch = round(pitch)
58
59     return pitch
60
61 while True:
62     # Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)
63     drive_bot()
64
65     # Get the current pitch angle
66     pitch = get_pitch()
67
68     # Print the pitch angle to the console window
69     print("Pitch: ", pitch)
70
71     # Slow down the display for better readability
72     sleep_ms(50)
73
```

***Objective 4* - Prettier Pitch**

WOW, you're fast!

I hope you've enjoyed getting behind the wheel of CodeBot!

I don't know about you, but I think the console output was a little weak on presentation.

- Make some improvements by displaying your pitch data in a more graphical and fun manner...
- *It's time to take this display to the next level!*

Packin' a Powerful Pitch

First define the range of pitch possibilities for your roamin' robot!

- The highest your CodeBot's nose can go is straight up, which is 90°.
- The lowest it can go is straight down, which is -90°.

Using this pitch range and string manipulation, you can create a ***bar graph*** display to represent your current pitch relative to 0°. Here are two examples:

```
'[-90°        == -20°           +90°]'

'[-90°            80° ========  +90°]'
```

Format Specifiers

Check out the ***Format Specifiers*** section of the string formatting toolbox entry.

To create the display above, you'll need to *align* the `pitch` value. You can put an "alignment" character just after the colon in the format specifier, and before the **width**.

Ex:

```
pitch = 45
"{:^20}".format(pitch)  # Align center: '         45         '
"{:>20}".format(pitch)  # Align right:  '                  45'
"{:<20}".format(pitch)  # Align left    '45                  '
```

CodeTrek:

```
1  from botcore import *
2  import math
3  from time import sleep_ms
4
5  # Constants
6  SPEED_LIMIT = 70
7  CODEBOT_SLANT = 20
8  ONE_G = 16384
9
10 # Global variables for motor power
11 left_power = 0
12 right_power = 0
13
14 # Enable motors
15 motors.enable(True)
16
17 def drive_bot():
18     """Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)"""
19     global left_power, right_power
20
21     # Accelerate slowly if button is pressed
22     if buttons.is_pressed(LEFT):
23         if left_power < SPEED_LIMIT:
24             left_power = left_power + 1
25     elif left_power > 1:
26         # Decelerate if button not pressed
27         left_power = left_power - 2
28
29     # Accelerate slowly if button is pressed
30     if buttons.is_pressed(RIGHT):
31         if right_power < SPEED_LIMIT:
32             right_power = right_power + 1
33     elif right_power > 1:
34         # Decelerate if button not pressed
35         right_power = right_power - 2
```

```
36
37      # Apply the power!
38      motors.run(LEFT, left_power)
39      motors.run(RIGHT, right_power)
40
41  def get_pitch():
42      """Get the current pitch angle of the platform in degrees"""
43      # Read the raw accelerometer data
44      x, y, z = accel.read()
45
46      # Calculate pitch and convert angle to degrees
47      pitch = math.asin(y / ONE_G)
48      pitch = pitch * 180 / math.pi
49
50      # Subtract CodeBot slant
51      pitch = pitch - CODEBOT_SLANT
52
53      # Make "looking up" a positive angle
54      pitch = -pitch
55
56      # Round to the nearest integer
57      pitch = round(pitch)
58
59      return pitch
60
61  def dashboard(pitch):
62      # Make a bar graph string of up to 30 segments
63      num_bars = abs(pitch) / 3
```

Your **bar graph** will be made of *segments*.

- Use the built-in `abs()` function here to calculate the number of segments to display.
- Dividing by 2 gives you one segment for every 2° of *pitch*.

```
64
65      # Use '=' character for the bar graph segments
66      bar_graph = '=' * num_bars  # TODO: what if num_bars is not an integer?
```

Use string multiplication

- This creates a new string by repeating `'='` a number of times.
- Make sure it's a *round* number though!

(You may have to fix a bug here!)

```
67
68      # Negative on the left, positive on the right!
69      bars_left = bars_right = ''
```

Cascaded assignment statements. *Oh My!*

- Sets both `bars_left` and `bars_right` equal to an empty string.

```
70      if pitch < 0:
71          bars_left = bar_graph
72      else:
73          bars_right = bar_graph
```

Your *bar graph* will be on the *left* or *right* depending on whether `pitch` is positive or negative.

```
74
75      # Use "align" character for LEFT and RIGHT alignment of bars
76      dash = "[-90 {:>30} {:+3} {:<30} +90]".format(bars_left, pitch, bars_right)
```

Fancy Formatting

This is where you use the *left* and *right* **align** characters to format the bars.

```
77
78      print(dash)
79
80  while True:
81      # Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)
82      drive_bot()
83
84      # Get the current pitch angle
85      pitch = get_pitch()
86
87      # Display a beautifully formatted pitch dashboard
88      dashboard(pitch)
```

Finally, display your one-line *pitch dashboard!*

```
89
90      # Slow down the display for better readability
91      sleep_ms(50)
92
```

Hint:

- ## Too Wide for Your Console?

  If your console display doesn't look nice, you may need to widen it.

  - Text printed to the console will "wrap around" if the line length exceeds the available width.
  - If this is happening, try resizing the console panel by dragging the window border.

  OR you can reduce the width of your *dashboard* to fewer than 30 segments on each side.

Goals:

- Use **Format Specifiers** to create a bar graph display for your `pitch`.
- Drive around using the buttons until you see:
  - At least 5-segments of *negative* bar graph.
- Drive around using the buttons until you see:
  - At least 5-segments of *positive* bar graph.

Tools Found: Print Function, str, undefined, String Formatting, Buttons, Built-In Functions, Assignment

Solution:

```
1   from botcore import *
2   import math
3   from time import sleep_ms
4
5   # Constants
6   SPEED_LIMIT = 70
7   CODEBOT_SLANT = 20  # Measured earlier #@1
8   ONE_G = 16384
9
10  # Global variables
11  left_power = 0
12  right_power = 0
13
14  # Enable motors
15  motors.enable(True)
16
17  def drive_bot():
18      """Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)"""
19      global left_power, right_power
```

```
20
21      # Accelerate slowly if button is pressed
22      if buttons.is_pressed(LEFT):
23          if left_power < SPEED_LIMIT:
24              left_power = left_power + 1
25      elif left_power > 1:
26          # Decelerate if button not pressed
27          left_power = left_power - 2
28
29      # Accelerate slowly if button is pressed
30      if buttons.is_pressed(RIGHT):
31          if right_power < SPEED_LIMIT:
32              right_power = right_power + 1
33      elif right_power > 1:
34          # Decelerate if button not pressed
35          right_power = right_power - 2
36
37      # Apply the power!
38      motors.run(LEFT, left_power)
39      motors.run(RIGHT, right_power)
40
41  def get_pitch():
42      """Get the current pitch angle of the platform in degrees"""
43      # Read the raw accelerometer data
44      x, y, z = accel.read()
45
46      # Calculate pitch and convert angle to degrees
47      pitch = math.asin(y / ONE_G)
48      pitch = pitch * 180 / math.pi
49
50      # Subtract CodeBot slant
51      pitch = pitch - CODEBOT_SLANT
52
53      # Make "looking up" a positive angle
54      pitch = -pitch
55
56      # Round to the nearest integer
57      pitch = round(pitch)
58
59      return pitch
60
61  def dashboard(pitch):
62      # Make a bar graph string of up to 30 segments
63      num_bars = abs(pitch) / 3
64
65      num_bars = round(num_bars)
66
67      # Use '=' character for the bar graph segments
68      bar_graph = '=' * num_bars  # TODO: what if num_bars is not an integer?
69
70      # Negative on the left, positive on the right!
71      bars_left = bars_right = ''
72      if pitch < 0:
73          bars_left = bar_graph
74      else:
75          bars_right = bar_graph
76
77      # Use "align" character for LEFT and RIGHT alignment of bars
78      dash = "[-90 {:>30} {:+3} {:<30} +90]".format(bars_left, pitch, bars_right)
79
80      print(dash)
81
82  while True:
83      # Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)
84      drive_bot()
85
86      # Get the current pitch angle
87      pitch = get_pitch()
88
89      # Display a beautifully formatted pitch dashboard
90      dashboard(pitch)
91
92      # Slow down the display for better readability
```

```
93      sleep_ms(50)
94
```

***Objective 5* - Get a Degree**

About your dashboard...

There's something missing. Your *pitch* angle is great, and the *bar graph* is a very helpful visualization of the data.

- But you gotta have UNITS for those *numbers!*
- What's up with `90`? Is that **90 *feet*** or what?

Get your *degree* on!

Your keyboard probably doesn't have a key for the "degree" symbol → °

So how are you going to put it into a Python string?

> **ESCAPE** the limitations of your *keyboard* with...

Escape Sequences

Head over to the string tool in your *toolbox* and scroll down to see the list of *escape sequences.* These special characters following a \ give your strings *superpowers!*

- Internally, strings are really just sequences of *numbers*. They're translated for display using a Character Encoding.
  - The first 128 characters are from the standard ASCII character set.
- You can use the escape sequence \xNN to insert a numeric *character-code* into a string, to represent ASCII characters and beyond!

HEX Me!

The \x escape sequence lets you insert a character code using a number in *base-16*, aka "hexadecimal" or "Hex".

Why base-16?

- Since 16 is a power of 2, it's a *nice round number* in binary!
- A single hex digit holds exactly 4-bits of information ($2^4 = 16$)
- Why not use decimal? A single *decimal* digit holds about 3.3 bits... (exactly $\log_2 10$)
  - Not very convenient if you're filling out a string of fixed-size binary numbers!

Why Care?

Well, so you can put *crazy characters* in your strings for one thing! For example the ***degree*** symbol is *extended ASCII code: 176*, which is B0 in hex.

CodeTrek:

```
1  from botcore import *
2  import math
3  from time import sleep_ms
4
5  # Constants
6  SPEED_LIMIT = 70
7  CODEBOT_SLANT = 20  # Measured earlier
8  ONE_G = 16384
9
10 # Global variables for motor power
11 left_power = 0
12 right_power = 0
13
14 # Enable motors
15 motors.enable(True)
16
17 def drive_bot():
18     """Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)"""
19     global left_power, right_power
20
```

```
21      # Accelerate slowly if button is pressed
22      if buttons.is_pressed(LEFT):
23          if left_power < SPEED_LIMIT:
24              left_power = left_power + 1
25      elif left_power > 1:
26          # Decelerate if button not pressed
27          left_power = left_power - 2
28
29      # Accelerate slowly if button is pressed
30      if buttons.is_pressed(RIGHT):
31          if right_power < SPEED_LIMIT:
32              right_power = right_power + 1
33      elif right_power > 1:
34          # Decelerate if button not pressed
35          right_power = right_power - 2
36
37      # Apply the power!
38      motors.run(LEFT, left_power)
39      motors.run(RIGHT, right_power)
40
41  def get_pitch():
42      """Get the current pitch angle of the platform in degrees"""
43      # Read the raw accelerometer data
44      x, y, z = accel.read()
45
46      # Calculate pitch and convert angle to degrees
47      pitch = math.asin(y / ONE_G)
48      pitch = pitch * 180 / math.pi
49
50      # Subtract CodeBot slant
51      pitch = pitch - CODEBOT_SLANT
52
53      # Make "looking up" a positive angle
54      pitch = -pitch
55
56      # Round to the nearest integer
57      pitch = round(pitch)
58
59      return pitch
60
61  def dashboard(pitch):
62      # Make a bar graph string of up to 30 segments
63      num_bars = abs(pitch) / 3
64
65      num_bars = round(num_bars)
66
67      # Use '=' character for the bar graph segments
68      bar_graph = '=' * num_bars
69
70      # Negative on the left, positive on the right!
71      bars_left = bars_right = ''
72      if pitch < 0:
73          bars_left = bar_graph
74      else:
75          bars_right = bar_graph
76
77      # Use "align" character for LEFT and RIGHT alignment of bars
78      dash = "[-90\xB0 {:>30} {:+3}\xB0 {:<30} +90\xB0]".format(bars_left, pitch, bars_right)
```

Just one line to change here.

- Add the escape sequence for degree: \xB0
- *You will need it in 3 places in your format string.*

```
79
80      print(dash)
81
82  while True:
83      # Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)
84      drive_bot()
85
86      # Get the current pitch angle
87      pitch = get_pitch()
88
```

```
89      # Display a beautifully formatted pitch dashboard
90      dashboard(pitch)
91
92      # Slow down the display for better readability
93      sleep_ms(50)
94
```

Goals:

- Open up the REPL, type the following string, and press ENTER

```
"A 90\xB0 turn"
```

- Modify your *dashboard* format string to add the *degree* symbol.
    - Be sure all 3 angles have a nice ° symbol appended!

Tools Found: str, Character Encoding, Escape Sequences

Solution:

```
1  from botcore import *
2  import math
3  from time import sleep_ms
4
5  # Constants
6  SPEED_LIMIT = 70
7  CODEBOT_SLANT = 20  # Measured earlier #@1
8  ONE_G = 16384
9
10 # Global variables
11 left_power = 0
12 right_power = 0
13
14 # Enable motors
15 motors.enable(True)
16
17 def drive_bot():
18     """Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)"""
19     global left_power, right_power
20
21     # Accelerate slowly if button is pressed
22     if buttons.is_pressed(LEFT):
23         if left_power < SPEED_LIMIT:
24             left_power = left_power + 1
25     elif left_power > 1:
26         # Decelerate if button not pressed
27         left_power = left_power - 2
28
29     # Accelerate slowly if button is pressed
30     if buttons.is_pressed(RIGHT):
31         if right_power < SPEED_LIMIT:
32             right_power = right_power + 1
33     elif right_power > 1:
34         # Decelerate if button not pressed
35         right_power = right_power - 2
36
37     # Apply the power!
38     motors.run(LEFT, left_power)
39     motors.run(RIGHT, right_power)
40
41 def get_pitch():
42     """Get the current pitch angle of the platform in degrees"""
43     # Read the raw accelerometer data
44     x, y, z = accel.read()
45
46     # Calculate pitch and convert angle to degrees
47     pitch = math.asin(y / ONE_G)
48     pitch = pitch * 180 / math.pi
```

```
49
50     # Subtract CodeBot slant
51     pitch = pitch - CODEBOT_SLANT
52
53     # Make "looking up" a positive angle
54     pitch = -pitch
55
56     # Round to the nearest integer
57     pitch = round(pitch)
58
59     return pitch
60
61 def dashboard(pitch):
62     # Make a bar graph string of up to 30 segments
63     num_bars = abs(pitch) / 3
64
65     num_bars = round(num_bars)
66
67     # Use '=' character for the bar graph segments
68     bar_graph = '=' * num_bars  # TODO: what if num_bars is not an integer?
69
70     # Negative on the left, positive on the right!
71     bars_left = bars_right = ''
72     if pitch < 0:
73         bars_left = bar_graph
74     else:
75         bars_right = bar_graph
76
77     # Use "align" character for LEFT and RIGHT alignment of bars
78     dash = "[-90\xB0 {:>30} {:+3}\xB0 {:<30} +90\xB0]".format(bars_left, pitch, bars_right)
79
80     print(dash)
81
82 while True:
83     # Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)
84     drive_bot()
85
86     # Get the current pitch angle
87     pitch = get_pitch()
88
89     # Display a beautifully formatted pitch dashboard
90     dashboard(pitch)
91
92     # Slow down the display for better readability
93     sleep_ms(50)
94
```

Quiz 2 **- String Theory**

Question 1: Which of the following print statements will pad the word "dog" with ***center*** alignment?

❌ `print("{:25}".format("dog"))`

❌ `print("{:>25}".format("dog"))`

✔ `print("{:^25}".format("dog"))`

❌ `print("{:<25}".format("dog"))`

Question 2: What character is displayed by the following escape sequence?

```
"\x40"
```

(try it on the REPL)

✔ @

❌ Q

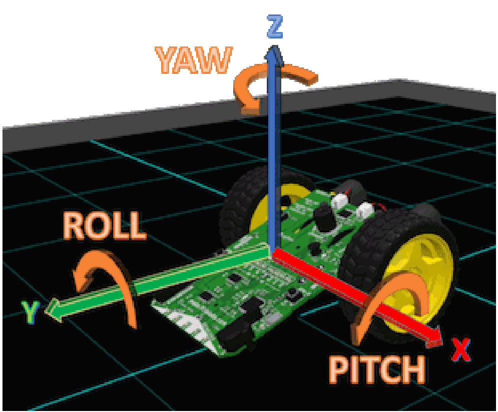
®

Objective 6 **- Roll with the Punches**

Pitch Me a Roll

Now that you have your *pitch* worked out, there's another *principal axis* you need to be aware of for navigation.

- *Roll* is when your 'bot is *leaning* to the **left** or **right**.



From the above diagram, you can see that the **X-axis** tilts *up* and *down* with *roll* just like the Y-axis did with *pitch*.

> You can use the same code, just change Y to X!

One More Thing...

You need to display **BOTH** *pitch* and *roll* on your dashboard!

- You will have to reduce the width.
- Also, enough with the *scrolling!*
  - Use the 🔧escape sequence for ***Carriage Return*** instead.

Carriage Return just moves your cursor back to the beginning of the same line. So the next `print()` statement will write on top of the last one!

Hold On!

You might notice there are *rock climbing* ***holds*** attached to the mountain now. Just a little extra texture to spice up your climb. *Rock and Roll!*

CodeTrek:

```
1  from botcore import *
2  import math
3  from time import sleep_ms
4
5  # Constants
6  SPEED_LIMIT = 70
7  CODEBOT_SLANT = 20  # Measured earlier
8  ONE_G = 16384
9
10 # Global variables for motor power
11 left_power = 0
12 right_power = 0
13
14 # Enable motors
15 motors.enable(True)
16
17 def drive_bot():
18     """Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)"""
19     global left_power, right_power
20
21     # Accelerate slowly if button is pressed
```

```
22      if buttons.is_pressed(LEFT):
23          if left_power < SPEED_LIMIT:
24              left_power = left_power + 1
25      elif left_power > 1:
26          # Decelerate if button not pressed
27          left_power = left_power - 2
28
29      # Accelerate slowly if button is pressed
30      if buttons.is_pressed(RIGHT):
31          if right_power < SPEED_LIMIT:
32              right_power = right_power + 1
33      elif right_power > 1:
34          # Decelerate if button not pressed
35          right_power = right_power - 2
36
37      # Apply the power!
38      motors.run(LEFT, left_power)
39      motors.run(RIGHT, right_power)
40
41  def get_axis(val, offset=0):
42      """Get the given angle of the platform in degrees"""
43
44      # Calculate axis and convert angle to degrees
45      axis = math.asin(val / ONE_G)
46      axis = axis * 180 / math.pi
```

Start 🔧refactoring your code here!

This is the `get_pitch()` code, but renamed and with 🔧parameters.

- Move the `accel.read()` *outside* of this function.
- `val` will be `y` for *pitch*, or `x` for `roll` axis.

```
47
48      # Apply offset
49      axis = axis + offset
```

Replace the `CODEBOT_SLANT` with an `offset` parameter.

- *roll* is not affected by the slant.

```
50
51      # Invert angle
52      axis = -axis
53
54      # Round to the nearest integer
55      axis = round(axis)
56
57      return axis
58
59  def dashboard(val):
60      # Make a bar graph string
61      num_bars = abs(val) / 3
62
63      num_bars = round(num_bars)
64
65      # Truncate at 10 segments
66      num_bars = min(10, num_bars)
```

Your `dashboard()` function needs a little work too.

- Reduce the width to just 10 segments.
- Use the 🔧built-in `min()` function to make sure `num_bars` never exceeds 10.

```
67
68      # Use '=' character for the bar graph segments
69      bar_graph = '=' * num_bars
70
71      # Negative on the left, positive on the right!
72      bars_left = bars_right = ''
73      if val < 0:
```

```
74          bars_left = bar_graph
75      else:
76          bars_right = bar_graph
77
78      # Use "align" character for LEFT and RIGHT alignment of bars
79      dash = "[-90\xB0 {:>10} {:+3}\xB0 {:<10} +90\xB0]".format(bars_left, val, bars_right)
```

Change your *width* values in the format string to **10** also.

```
80
81      # Print on one line with no "newline" at end.
82      print(dash, end='')
```

You are going to print this **twice** on the same line, so remove the default *end=newline* from `print()`.

```
83
84  # Blank line, so dashboard displays below the Python prompt
85  print()
```

When your program starts, go ahead and print a *blank line*, so your dashboard starts on a fresh line of its own!

```
86
87  while True:
88      # Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)
89      drive_bot()
90
91      # Read the raw accelerometer data
92      x, y, z = accel.read()
```

Accelerometer `read()` is moved here, outside of any functions.

```
93
94      # Get the current pitch and roll angles
95      pitch = get_axis(y, -CODEBOT_SLANT)
96      roll = get_axis(x)
```

Look! Using **common code** to calculate *pitch* and *roll!*

- That's the power of refactoring!

```
97
98      # Display a beautifully formatted pitch dashboard
99      print("P", end=':')
100     dashboard(pitch)
101     print(" R", end=':')
102     dashboard(roll)
```

Be sure to ***label*** the parts of your *dashboard.*

- Notice that again you must supply an `end` argument to `print()`
- Otherwise it will skip to the next line, and mess up the display!

```
103
104     # Carriage Return - move back to start of same line
105     print("\r", end='')
#@10
106
107     # Slow down the display for better readability
108     sleep_ms(50)
109
```

Goals:

- Add *roll* to your *dashboard* display. Display should have *both* pitch and roll, labeled as shown:

```
P:[-90°    +13° ====  +90°] R:[-90°    +7° ==   +90°]
```

- Drive up the slope, and rotate until:
  - **Pitch** → + *positive*
  - **Roll** → + *positive*
- Rotate until:
  - **Pitch** → + *positive*
  - **Roll** → - *negative*
- Rotate until:
  - **Pitch** → - *negative*
  - **Roll** → + *positive*
- Rotate until:
  - **Pitch** → - *negative*
  - **Roll** → - *negative*

Tools Found: Escape Sequences, undefined, Refactoring, Parameters, Arguments, and Returns, Built-In Functions, String Formatting, Accelerometer

Solution:

```
1  from botcore import *
2  import math
3  from time import sleep_ms
4
5  # Constants
6  SPEED_LIMIT = 70
7  CODEBOT_SLANT = 20  # Measured earlier #@1
8  ONE_G = 16384
9
10 # Global variables
11 left_power = 0
12 right_power = 0
13
14 # Enable motors
15 motors.enable(True)
16
17 def drive_bot():
18     """Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)"""
19     global left_power, right_power
20
21     # Accelerate slowly if button is pressed
22     if buttons.is_pressed(LEFT):
23         if left_power < SPEED_LIMIT:
24             left_power = left_power + 1
25     elif left_power > 1:
26         # Decelerate if button not pressed
27         left_power = left_power - 2
28
29     # Accelerate slowly if button is pressed
30     if buttons.is_pressed(RIGHT):
31         if right_power < SPEED_LIMIT:
32             right_power = right_power + 1
33     elif right_power > 1:
34         # Decelerate if button not pressed
35         right_power = right_power - 2
36
37     # Apply the power!
38     motors.run(LEFT, left_power)
39     motors.run(RIGHT, right_power)
```

```
40
41 def get_axis(val, offset=0):
42     """Get the given angle of the platform in degrees"""
43
44     # Calculate axis and convert angle to degrees
45     axis = math.asin(val / ONE_G)
46     axis = axis * 180 / math.pi
47
48     # Subtract offset
49     axis = axis + offset
50
51     # Make "looking up" a positive angle
52     axis = -axis
53
54     # Round to the nearest integer
55     axis = round(axis)
56
57     return axis
58
59 def dashboard(val):
60     # Make a bar graph string
61     num_bars = abs(val) / 3
62
63     num_bars = round(num_bars)
64
65     # Truncate at 10 segments
66     num_bars = min(10, num_bars)
67
68     # Use '=' character for the bar graph segments
69     bar_graph = '=' * num_bars
70
71     # Negative on the left, positive on the right!
72     bars_left = bars_right = ''
73     if val < 0:
74         bars_left = bar_graph
75     else:
76         bars_right = bar_graph
77
78     # Use "align" character for LEFT and RIGHT alignment of bars
79     dash = "[-90\xB0 {:>10} {:+3}\xB0 {:<10} +90\xB0]".format(bars_left, val, bars_right)
80
81     # Print on one line with no "newline" at end.
82     print(dash, end='')
83
84 # Blank line, so dashboard displays below the Python prompt
85 print()
86
87 while True:
88     # Drive the CodeBot with BTN0 and BTN1 ('0' and '1' keys)
89     drive_bot()
90
91     # Read the raw accelerometer data
92     x, y, z = accel.read()
93
94     # Get the current pitch and roll angles
95     pitch = get_axis(y, -CODEBOT_SLANT)
96     roll = get_axis(x)
97
98     # Display a beautifully formatted pitch dashboard
99     print("P", end=':')
100    dashboard(pitch)
101    print(" R", end=':')
102    dashboard(roll)
103
104    # Carriage Return - move back to start of same line
105    print("\r", end='')
106
107    # Slow down the display for better readability
108    sleep_ms(50)
109
```

***Objective 7* - Free Solo Climb**

Free Solo Climb!

You've been navigating this mountain by *remote control*. Pretty cool.

- And you have a *sweet* dashboard, showing **pitch** and **roll**.
- With accurate data like that, CodeBot could practically drive itself up the mountain.

So... *empower your 'bot to drive itself!*

Autonomous Robotics Time!

Coding a *remote control* device is okay. But coding an *intelligent* robot that **senses its environment** and **responds to changes** is exciting! A robot like that is ***autonomous!***

Navigating Uphill

Your objective here is to keep CodeBot's nose pointed up the mountain.

- That means you want positive *pitch:* `pitch > 0`
- And if you are pointed *straight uphill* your *roll* will be zero: `roll == 0`

Test drive

Use your previous code to drive on a slope, and watch the **roll** value. **Could you keep pointed uphill based on the `roll` value alone?**

How about this **Algorithm?**

Roll →	Action
Negative	Turn Right
Positive	Turn Left

*Activate **full self driving** mode!*

CodeTrek:

```
1  from botcore import *
2  import math
3  from time import sleep_ms
4
5  # Constants
6  SPEED_LIMIT = 70
7  CODEBOT_SLANT = 20
8  ONE_G = 16384
9
10 # Motor control
11 left_power = 0
12 right_power = 0
13
14 # Enable motors
15 motors.enable(True)
16
17 def drive_bot(pitch, roll):
18     """Drive the CodeBot based on pitch and roll"""
```

AUTONOMY!

That means no more button input in this function. Instead, add 🔧parameters for `pitch` and `roll`, so you can *drive with **data!***

```
19     global left_power, right_power
20
21     left_power_target = right_power_target = SPEED_LIMIT
```

Define "target" power levels.

- Default these to "full speed ahead" !

```
22
23      # Control how hard to turn. Reduce for sharper turns.
24      steer_ratio = 0.7
25
26      # Steer based on 'roll' angle: strive for zero roll!
27      if roll > 0:
28          left_power_target *= steer_ratio
29      elif roll < 0:
30          right_power_target *= steer_ratio
```

Steer based on `roll`

- Zero roll is straight uphill. Non-zero means TURN!
- Use `steer_ratio` to adjust the target speed.

```
31
32      # Adjust power toward target
33      left_power += (left_power_target - left_power) * 0.1
34      right_power += (right_power_target - right_power) * 0.1
```

No Sudden Moves!

Give your **power** levels a nudge toward the *targets.*

- Remember, `drive_bot()` is called constantly from your main 🔧loop.

```
35
36      # Apply the power!
37      motors.run(LEFT, left_power)
38      motors.run(RIGHT, right_power)
39
40  def get_axis(val, offset=0):
41      """Get the given angle of the platform in degrees"""
42
43      # Calculate axis and convert angle to degrees
44      axis = math.asin(val / ONE_G)
45      axis = axis * 180 / math.pi
46
47      # Subtract offset
48      axis = axis + offset
49
50      # Make "looking up" a positive angle
51      axis = -axis
52
53      # Round to the nearest integer
54      axis = round(axis)
55
56      return axis
57
58  def dashboard(val):
59      # Make a bar graph string
60      num_bars = abs(val) / 3
61
62      num_bars = round(num_bars)
63
64      # Truncate at 10 segments
65      num_bars = min(10, num_bars)
66
67      # Use '=' character for the bar graph segments
68      bar_graph = '=' * num_bars
69
70      # Negative on the left, positive on the right!
71      bars_left = bars_right = ''
72      if val < 0:
73          bars_left = bar_graph
74      else:
75          bars_right = bar_graph
76
77      # Use "align" character for LEFT and RIGHT alignment of bars
78      dash = "[-90\xB0 {:>10} {:+3}\xB0 {:<10} +90\xB0]".format(bars_left, val, bars_right)
79
80      # Print on one line with no "newline" at end.
```

```
81      print(dash, end='')
82
83  # Blank line, so dashboard displays below the Python prompt
84  print()
85
86  while True:
87      # Read the raw accelerometer data
88      x, y, z = accel.read()
89
90      # Get the current pitch and roll angles
91      pitch = get_axis(y, -CODEBOT_SLANT)
92      roll = get_axis(x)
93
94      drive_bot()  #TODO: pass-in pitch and roll here
```

Pass `pitch` and `roll` as 🔧arguments to `drive_bot()`

- *A little work to do here...*

```
95
96      # Display a beautifully formatted pitch dashboard
97      print("P", end=':')
98      dashboard(pitch)
99      print(" R", end=':')
100     dashboard(roll)
101
102     # Carriage Return - move back to start of same line
103     print("\r", end='')
104
105     # Slow down the display for better readability
106     sleep_ms(50)
107
```

Goals:

- Climb the mountain *autonomously*
  - Remove the 🔧buttons from your code.
  - No more *remote control* - this 'bot is *off the leash!*
- Reach the first summit
- Go beyond the first summit.
  - Into the *Valley of Reconsideration!*
- Reverse Course - Back to the Summit!
  - Return from the *Valley of Reconsideration*

Tools Found: Buttons, Parameters, Arguments, and Returns, Loops, Keyword and Positional Arguments

Solution:

```
1   from botcore import *
2   import math
3   from time import sleep_ms
4
5   # Constants
6   SPEED_LIMIT = 70
7   CODEBOT_SLANT = 20  # Measured earlier #@1
8   ONE_G = 16384
9
10  # Motor control
11  left_power = 0
12  right_power = 0
13
```

```
14  # Enable motors
15  motors.enable(True)
16  
17  def drive_bot(pitch, roll):
18      """Drive the CodeBot based on pitch and roll"""
19      global left_power, right_power
20  
21      # Strive for pitch > 0 and roll == 0
22      # roll+ --> steer left
23      # roll- --> steer right
24  
25      left_power_target = right_power_target = SPEED_LIMIT
26  
27      steer_ratio = 0.7
28  
29      if roll > 0:
30          left_power_target *= steer_ratio
31      elif roll < 0:
32          right_power_target *= steer_ratio
33  
34      # Adjust power toward target
35      left_power += (left_power_target - left_power) * 0.1
36      right_power += (right_power_target - right_power) * 0.1
37  
38      # Apply the power!
39      motors.run(LEFT, left_power)
40      motors.run(RIGHT, right_power)
41  
42  def get_axis(val, offset=0):
43      """Get the given angle of the platform in degrees"""
44  
45      # Calculate axis and convert angle to degrees
46      axis = math.asin(val / ONE_G)
47      axis = axis * 180 / math.pi
48  
49      # Subtract offset
50      axis = axis + offset
51  
52      # Make "looking up" a positive angle
53      axis = -axis
54  
55      # Round to the nearest integer
56      axis = round(axis)
57  
58      return axis
59  
60  def dashboard(val):
61      # Make a bar graph string
62      num_bars = abs(val) / 3
63  
64      num_bars = round(num_bars)
65  
66      # Truncate at 10 segments
67      num_bars = min(10, num_bars)
68  
69      # Use '=' character for the bar graph segments
70      bar_graph = '=' * num_bars
71  
72      # Negative on the left, positive on the right!
73      bars_left = bars_right = ''
74      if val < 0:
75          bars_left = bar_graph
76      else:
77          bars_right = bar_graph
78  
79      # Use "align" character for LEFT and RIGHT alignment of bars
80      dash = "[-90\xB0 {:>10} {:+3}\xB0 {:<10} +90\xB0]".format(bars_left, val, bars_right)
81  
82      # Print on one line with no "newline" at end.
83      print(dash, end='')
84  
85  # Blank line, so dashboard displays below the Python prompt
86  print()
87  
88  while True:
```

```
89      # Read the raw accelerometer data
90      x, y, z = accel.read()
91
92      # Get the current pitch and roll angles
93      pitch = get_axis(y, -CODEBOT_SLANT)
94      roll = get_axis(x)
95
96      drive_bot(pitch, roll)
97
98      # Display a beautifully formatted pitch dashboard
99      print("P", end=':')
100     dashboard(pitch)
101     print(" R", end=':')
102     dashboard(roll)
103
104     # Carriage Return - move back to start of same line
105     print("\r", end='')
106
107     # Slow down the display for better readability
108     sleep_ms(50)
109
```

***Objective 8* - First Ascent**

Escalation!

Ready to put your climbing code to the test?

- The *climbing holds* are **back**!
- *Just the thing to throw an unsuspecting robot **off course** !*

Give it a Go

Try your code on this mountain course. *It's not gonna be pretty!*

To conquer this new challenge you must add a bit more *intelligence* to your self-driving code.

- Detect when your 'bot has *crashed* into an obstacle, and add some *avoidance* code.
- How to detect a **crash**? Well, if you `pitch` or `roll` more than 45° that's a pretty sure sign things are getting off-kilter!

Keep Trying!

You may not make it on your first few attempts.

CodeTrek:

```
1  from botcore import *
2  import math
3  from time import sleep_ms
4
5  # Constants
6  SPEED_LIMIT = 70
7  CODEBOT_SLANT = 20
8  ONE_G = 16384
9
10 # Motor control
11 left_power = 0
12 right_power = 0
13
14 # Crash detection
15 crash_backoff = 0
```

A 🔧global crash-backoff-countdown.

- When a crash is detected this will be set to countdown while CodeBot recovers.

```
16
17 # Enable motors
18 motors.enable(True)
```

```
19
20 def drive_bot(pitch, roll):
21     """Drive the CodeBot based on pitch and roll"""
22     global left_power, right_power
23
24     left_power_target = right_power_target = SPEED_LIMIT
25
26     # Back-up a bit if crash detected
27     crash = pitch > 45 or abs(roll) > 45
28     if crash:
29         crash_backoff = 20  # Number of loop cycles:
30                             #  increase for longer backoff time.
```

Detect a "Crash"

If the *pitch* or *roll* get crazy, start the "backoff" countdown.

- Remember, this function is called *repeatedly* from your main loop.
- That's why `crash_backoff` has to be global, so it is retained between function calls.

*You **did** remember to add it to the `global` list, right?*

```
31
32     if crash_backoff:
33         # Stomp on the brakes!
34         left_power = -SPEED_LIMIT
35         right_power = -SPEED_LIMIT
36         crash_backoff -= 1
```

When in the "backoff" state:

- Set the motor power variables *directly.*
- Not "nudging" toward a *target* value in this case. *Slam on those brakes!*

Would it be better to *turn* while reversing? *Perhaps...*

```
37
38     steer_ratio = 0.7
39
40     if roll > 0:
41         left_power_target *= steer_ratio
42     elif roll < 0:
43         right_power_target *= steer_ratio
44
45     # Adjust power toward target
46     left_power += (left_power_target - left_power) * 0.1
47     right_power += (right_power_target - right_power) * 0.1
48
49     # Apply the power!
50     motors.run(LEFT, left_power)
51     motors.run(RIGHT, right_power)
52
53 def get_axis(val, offset=0):
54     """Get the given angle of the platform in degrees"""
55
56     # Calculate axis and convert angle to degrees
57     axis = math.asin(val / ONE_G)
58     axis = axis * 180 / math.pi
59
60     # Subtract offset
61     axis = axis + offset
62
63     # Make "looking up" a positive angle
64     axis = -axis
65
66     # Round to the nearest integer
67     axis = round(axis)
68
69     return axis
70
71 def dashboard(val):
72     # Make a bar graph string
73     num_bars = abs(val) / 3
```

```
74
75      num_bars = round(num_bars)
76
77      # Truncate at 10 segments
78      num_bars = min(10, num_bars)
79
80      # Use '=' character for the bar graph segments
81      bar_graph = '=' * num_bars
82
83      # Negative on the left, positive on the right!
84      bars_left = bars_right = ''
85      if val < 0:
86          bars_left = bar_graph
87      else:
88          bars_right = bar_graph
89
90      # Use "align" character for LEFT and RIGHT alignment of bars
91      dash = "[-90\xB0 {:>10} {:+3}\xB0 {:<10} +90\xB0]".format(bars_left, val, bars_right)
92
93      # Print on one line with no "newline" at end.
94      print(dash, end='')
95
96  # Blank line, so dashboard displays below the Python prompt
97  print()
98
99  while True:
100     # Read the raw accelerometer data
101     x, y, z = accel.read()
102
103     # Get the current pitch and roll angles
104     pitch = get_axis(y, -CODEBOT_SLANT)
105     roll = get_axis(x)
106
107     drive_bot(pitch, roll)
108
109     # Display a beautifully formatted pitch dashboard
110     print("P", end=':')
111     dashboard(pitch)
112     print(" R", end=':')
113     dashboard(roll)
114
115     # Carriage Return - move back to start of same line
116     print("\r", end='')
117
118     # Slow down the display for better readability
119     sleep_ms(50)
120
```

Goals:

- Cross Panel 1
- Cross Panel 2
- Cross Panel 3
- Cross Panel 4
- Cross Panel 5
- Cross Panel 6
- Cross Panel 7

Tools Found: Locals and Globals

Solution:

```
1  from botcore import *
2  import math
```

```
 3  from time import sleep_ms
 4
 5  # Constants
 6  SPEED_LIMIT = 70
 7  CODEBOT_SLANT = 20
 8  ONE_G = 16384
 9
10  # Motor control
11  left_power = 0
12  right_power = 0
13
14  # Crash detection
15  crash_backoff = 0  #@1
16
17  # Enable motors
18  motors.enable(True)
19
20  def drive_bot(pitch, roll):
21      """Drive the CodeBot based on pitch and roll"""
22      global left_power, right_power, crash_backoff
23
24      left_power_target = right_power_target = SPEED_LIMIT
25
26      # Back-up a bit if crash detected
27      crash = pitch > 45 or abs(roll) > 45
28      if crash:
29          crash_backoff = 20
30
31      if crash_backoff:
32          # Stomp on the brakes!
33          left_power = -SPEED_LIMIT
34          right_power = -SPEED_LIMIT
35          crash_backoff -= 1
36
37      steer_ratio = 0.7
38
39      if roll > 0:
40          left_power_target *= steer_ratio
41      elif roll < 0:
42          right_power_target *= steer_ratio
43
44      # Adjust power toward target
45      left_power += (left_power_target - left_power) * 0.1
46      right_power += (right_power_target - right_power) * 0.1
47
48      # Apply the power!
49      motors.run(LEFT, left_power)
50      motors.run(RIGHT, right_power)
51
52  def get_axis(val, offset=0):
53      """Get the given angle of the platform in degrees"""
54
55      # Calculate axis and convert angle to degrees
56      axis = math.asin(val / ONE_G)
57      axis = axis * 180 / math.pi
58
59      # Subtract offset
60      axis = axis + offset
61
62      # Make "looking up" a positive angle
63      axis = -axis
64
65      # Round to the nearest integer
66      axis = round(axis)
67
68      return axis
69
70  def dashboard(val):
71      # Make a bar graph string
72      num_bars = abs(val) / 3
73
74      num_bars = round(num_bars)
75
76      # Truncate at 10 segments
77      num_bars = min(10, num_bars)
```

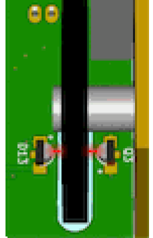
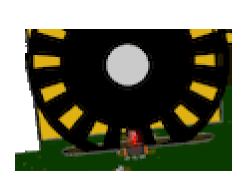
```
78
79      # Use '=' character for the bar graph segments
80      bar_graph = '=' * num_bars
81
82      # Negative on the left, positive on the right!
83      bars_left = bars_right = ''
84      if val < 0:
85          bars_left = bar_graph
86      else:
87          bars_right = bar_graph
88
89      # Use "align" character for LEFT and RIGHT alignment of bars
90      dash = "[-90\xB0 {:>10} {:+3}\xB0 {:<10} +90\xB0]".format(bars_left, val, bars_right)
91
92      # Print on one line with no "newline" at end.
93      print(dash, end='')
94
95  # Blank line, so dashboard displays below the Python prompt
96  print()
97
98  while True:
99      # Read the raw accelerometer data
100     x, y, z = accel.read()
101
102     # Get the current pitch and roll angles
103     pitch = get_axis(y, -CODEBOT_SLANT)
104     roll = get_axis(x)
105
106     drive_bot(pitch, roll)
107
108     # Display a beautifully formatted pitch dashboard
109     print("P", end=':')
110     dashboard(pitch)
111     print(" R", end=':')
112     dashboard(roll)
113
114     # Carriage Return - move back to start of same line
115     print("\r", end='')
116
117     # Slow down the display for better readability
118     sleep_ms(50)
119
```

Mission 13 **- Going the Distance**

Ready to go the distance? Then you'll need to get to know CodeBot's wheel encoders! This mission gives you all the gritty details of those glorious rotating discs.

Objective 1 **- Encoder Check**

Check your Encoders

This is a new sensor, and as usual you will start by testing the basics of how it works. The 🔧wheel encoders have just one key 🔧API function:

```
# Read the selected sensor (LEFT or RIGHT)
# Returns an ADC count 0-4095
val = enc.read(num)
```

Notice this `read()` function 🔧returns an 🔧ADC value.

- This 🔧analog value represents the amount of light shining through the slot in the disc.
- There are 20 slots in the disc, so you should see 20 light/dark transitions as the wheel rotates 360°.

Create a new file!

- Use the File → New File menu to create a new file called "encoder_check.py"

CodeTrek:

```
1  from botcore import *
```

No surprise, the encoders are part of the *botcore* library too!

```
2
3  # Start rotating slowly
4  motors.run(LEFT, 5)
5  motors.run(RIGHT, ?)
6  motors.enable( ?? )
```

TODO:

- Add code to make the **RIGHT** wheel move backwards, so your bot doesn't run into a wall while you're watching the console.
- Don't forget to `enable()` the motors !

```
7
8  # Loop forever, printing the LEFT encoder value
9  while True:
10     val = enc.read(LEFT)
11     print(val)
12

13
```

Goal:

- **First get your bot moving!**
  - Start the motors rotating slowly so the wheel encoders are turning.
  - Write a while loop that infinitely reads and prints the **LEFT** wheel encoder analog value to the console.

Tools Found: Wheel Encoders, API, Parameters, Arguments, and Returns, Analog to Digital Conversion, Motors, Loops, import

Solution:

```
1  from botcore import *
2
3  motors.run(LEFT, 5)
4  motors.run(RIGHT, -5)
5  motors.enable(True)
6
7  while True:
8      val = enc.read(LEFT)
9      print(val)
10
```

Objective 2 **- No Repetition Repetition**

That's a lot of output data!

Your Python code gets around that while loop pretty quickly.

- It's hard to read the output with so many numbers streaming by!

If you stop your program and scroll the console window up, you will notice *repeated numbers*.

> *Can you guess why* `enc.read()` *would return the same value multiple times?*

Try slowing down the motors...

You will notice even *more* repeated numbers. **Your loop is faster than the wheel encoders.**

Does this repetition matter?

Well, **yes!** This mission isn't just about printing data. You are going to count the slots as they go by, so you can measure distance and rotation. So throwing away those duplicate samples will be a good first step in your *data processing* work.

CodeTrek:

```
from botcore import *

motors.run(LEFT, 5)
motors.run(RIGHT, -5)
motors.enable(True)

# Initialize a variable to hold the previous encoder value
prev = 0
```

Initialize Your state

As your loop runs, you need to remember this one thing:

- What was the `previous` value?

Knowing this, you can filter-out duplicates!

```

while True:
    val = enc.read(LEFT)

    # Print the value only if it has changed
    if val != prev:
        print(val)
```

Ignore the duplicate values!

```

        # Remember the previous value processed.
        prev = val
```

Update your state.

- You now have a new `prev` value!

Hints:

- ## Save your previous value

  You will need another variable to hold the previous value.

  - This `prev` variable can be initialized before your loop begins.

- ### Inside your Loop:

  1. Read the *new* value.
  2. Compare it with the *previous* one.
  3. If they are different, print the *new* value.
  4. Overwrite the *previous* value with the current *new* value before the loop repeats.

Goal:

- ### Discard duplicate values before printing

  Add code to compare the new `enc.read(LEFT)` value with the previous one.

  - If they're different, `print` the new value.
  - Otherwise ignore it! *(throw it in the "bit bucket", eh?)*

Tools Found: Loops, Wheel Encoders, Print Function, State

Solution:

```
1  from botcore import *
2
3  motors.run(LEFT, 5)
4  motors.run(RIGHT, -5)
5  motors.enable(True)
6
7  prev = 0
8  while True:
9      val = enc.read(LEFT)
10     if val != prev:
11         print(val)
12         prev = val
13
```

Objective 3 **- Visualize the Sensor**

Getting Graphical with the Console

Okay I have to admit, those streaming numbers on the console are just a blur to me. It's hard to make out what's actually going on with the sensors.

Wouldn't a *Bar Chart* be nice?

But Hey, it's a *Text* Console, Right?

Yes, *but...* You can make some ASCII graphics!

Making *Bar Chart* style visualizations is really easy with Python strings. You could make the console print numbers like this:

```
4 ****
7 *******
8 ********
8 ********
7 *******
2 **
```

How to make repeated character strings

Say you have a number *N* and you want to make a string of characters that long.

- You might code a loop to make the '****' strings.
- But Python gives you an easier way.
- *Multiply* a string and an int and you get a repeated string!

Try it on the REPL!

You can type something like:

```
>>> 'M' * 10
'MMMMMMMMMM'
```

Try it with different strings, even your name!

Goal:

- **On the REPL**
  - Make a horizontal line of *exactly* 80 equals signs.
  - It should look like '=====...' out to 80 characters.

Tools Found: Character Encoding, str, Print Function, int, undefined

Solution:

N/A

Quiz 1 - String multiplication

Question 1: What shape will the following code print?

(Try on the REPL if you need to!)

```
for i in range(10):
    print("*" * i)
```

✓ Triangle

✗ Circle

✗ Square

✗ Hyperbola

Objective 4 - Graph It

Graph the Wheel Encoder Values

Back to your wheel encoder test program. Your next objective is to graph the encoder values using your new Python string skills.

The values it prints out can range from 0-4095 according to the documentation for these sensors. So you're going to need to *scale down* those values to make a bar graph that fits the console.

- If you divide every value by 100 then the max would be: $\frac{4095}{100} \approx 40$
- Hey, 40 characters as a max width would be great!

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  motors.run(LEFT, 50)
5  motors.run(RIGHT, -50)
6  motors.enable(True)
7
8  # Initialize a variable to hold the previous encoder value
9  prev = 0
10
11 while True:
12     val = enc.read(LEFT)
13
14     # Print the value only if it has changed
15     if val != prev:
16
17         # Scale down the encoder value from 4096 to around 40 max
18         n = val / 100
19         print(val, n * '*')  # Bar graph!
```

Bar Graph!

Two steps here:

1. Scale down `val` to a nice neat variable `n`
2. Use the *multiply* operator to repeat the `'*'` for a nifty bar-graph effect.

```
20
21          # Remember the previous value processed.
22          prev = val
23
```

Hints:

- ## ASCII Bar Graph Time!

  Kicking it old school.

  It's nice that Python lets you multiply a number by a string to make it easy to repeat. Can you multiply by *any* number?

  Hmmmm...

- ## The *goal* here is for you to encounter an ERROR

  Just type the code as shown in the CodeTrek, and you'll get an error when you run it.

  - Complete this Objective by triggering the error message.

Goal:

- **Graph Time?**
  - Try the code exactly as shown in the CodeTrek to print a '*' graph.
  - Just use `n = val / 100` and multiply that by the `'*'` character.
  - *What could go wrong?*

Tools Found: Wheel Encoders, Math Operators, str, list, tuple, float, int, Variables

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  motors.run(LEFT, 50)
5  motors.run(RIGHT, -50)
6  motors.enable(True)
7
8  prev = 0
9  while True:
10     val = enc.read(LEFT)
11     if val != prev:
12         n = val / 100
13         print(val, n * '*')
14         prev = val
15
```

***Objective 5* - Bar Chart**

Fix the Error and Get Charting!

That *TypeError* happens because Python doesn't support multiplying a 🔧string by a 🔧float.

- It *would* be strange to have a string with *fractions of letters*, right?

Say you do `n = val / 100` when val is `315`. What do you *really* want `n` to be?

- In that case you want `n = 3` so you'd have `'***'` in your bar chart.
- You could use Python's 🔧built-in `round()` function to convert $\frac{315}{100}$ to 3. But there is an even easier way, if you only want the 🔧integer portion of division.

Integer Division Operator

Check out the table of 🔧Math Operators in Python. See the one called **"Integer Division"**? It is a double-divide symbol `//`. Play with it on the REPL. Try typing both normal and *integer* divisions:

```
315 / 100
315 // 100
```

So that gives you a nice, simple way to fix the bug!

Goals:

- Test "normal" division on the REPL
  - Enter `315 / 100` and see what you get.
- Test *integer division* on the REPL
  - Enter `315 // 100` and see what you get.
- ## Fix the bug and view your graph

  Use *Integer Division* to fix the **TypeError** in your code, and watch your groovy-graph glide on by!

Tools Found: str, float, Built-In Functions, int, Math Operators, Wheel Encoders, Analog to Digital Conversion

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  motors.run(LEFT, 50)
5  motors.run(RIGHT, -50)
6  motors.enable(True)
7
8  # Initialize a variable to hold the previous encoder value
9  prev = 0
10
11 while True:
12     val = enc.read(LEFT)
13
14     # Print the value only if it has changed
15     if val != prev:
16
17         # Scale down the encoder value from 4096 to around 40 max
18         n = val // 100  # Integer division
19         print(val, n * '*')  # Bar graph!
20
21         # Remember the previous value processed.
22         prev = val
23
```

***Objective 6* - Count Slots**

Count the Slots

Now that you can visualize what's going on with the *encoder sensors* it's time to translate those values into a measurement of how far the wheel has turned.

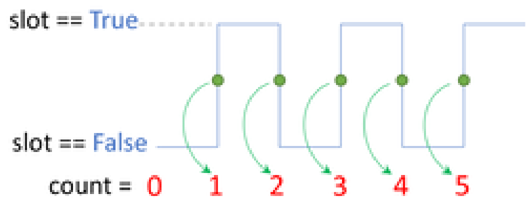
Rather than graphing the 🔧analog value, make a **True**/**False** 🔧boolean decision: **SLOT *or* NOT?**

- Use a 🔧comparison operator with a **threshold** midway between 0 and 4095 to detect the slot.
- Track the previous **True**/**False** state just like you did with `val`
- And count the changes of that **True**/**False** state.

Check Your Pulse!

Take a look at the picture to the right. It's showing how your code would read `True` or `False` as the wheel turns.

- Like a sideways, flat-top version of your *bar chart!*
- The `count` increases at every `False` → `True` or `True` → `False` transition.



Save to a new file!

You are moving on from your basic *encoder check* test-code.

- Use the *File* → *Save As* menu to save your code to a new file called "enc_drive.py"

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  # Choose a threshold value midway through the slot range
5  SLOT_THRESH = 2000
```

This is a constant that you choose.

- Pick a number based on what you observed with your *bar chart.*
- Ideally you'd like the "mountains" and "valleys" to be about equal width.

```
6
7  # Get those motors running SLOWLY for testing
8  motors.run(LEFT, 5)
9  motors.run(RIGHT, -5)
10 motors.enable(True)
11
12 # Initialize a variable to hold the previous encoder value
13 was_slot = False
```

Previous Value of `is_slot`

Change the code here. Instead of a `prev` copy of the encoder `val` you need to save a bool indicating whether a slot was detected on the previous comparison check.

```
14
15 # Initialize a variable to hold the slot count
16 count = 0
```

Argh... More Pieces of State

"State" is just a name for stuff your program has to "remember" while it runs.

- In this case, you have to keep track of the `count`.

```
17
18 while True:
19     # Read the latest value
20     val = enc.read(LEFT)
21
22     # Is it a slot, or not?
23     is_slot = val > SLOT_THRESH
```

Check the *latest* value against the *threshold*...

- The result of your comparison will be a boolean.

```
24
25     # Print the value only if it has changed
26     if is_slot != was_slot:
27
28         # Transitioned to/from a slot!
29         count += 1
30         print('Count is ', count)
```

Update the `count`

Add 1 to the current count, and save it back.

- Use an augmented assignment statement for this.
- Then `print()` the count to the console for *testing!

```
31
32         # Remember the previous value processed.
33         was_slot = is_slot
```

Wait! Before you go...

Your slot status has changed!

- Don't forget to save this new `was_slot` state before continuing.

```
34
```

Goals:

- **Print the Running Count**
  - Modify your code to `print()` the encoder ***count*** to the console.
  - As your bot rotates, I want to see 40 counts every 360° of wheel rotation like so:

```
Count is 1
Count is 2
...
```

- **Use Augmented Assignment to Update** `count`

  Save a little typing, and show me that you've mastered this coding shortcut.

Tools Found: Analog to Digital Conversion, bool, Comparison Operators, Wheel Encoders, Assignment, Constants

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # Choose a threshold value midway through the slot range
5  SLOT_THRESH = 2000
6
7  # Get those motors running SLOWLY for testing
8  motors.run(LEFT, 5)
9  motors.run(RIGHT, -5)
10 motors.enable(True)
11
12 # Initialize a variable to hold the previous encoder value
13 was_slot = False
14
15 # Initialize a variable to hold the slot count
16 count = 0
17
18 while True:
19     val = enc.read(LEFT)
20     is_slot = val > SLOT_THRESH
21
22     # Print the value only if it has changed
23     if is_slot != was_slot:
24
25         # Transitioned to/from a slot!
26         count += 1
```

```
27          print('Count is ', count)
28
29          # Remember the previous value processed.
30          was_slot = is_slot
31
```

Quiz 2 - Integer division and augmented assignment

Question 1: Which **two** of the following are `True`?

✓ `315 // 100 < 315 / 100`

✓ `1 // 2 == 0`

✗ `3 // 4 > 1 / 2`

✗ `round(3 / 4) == 3 // 4`

Question 2: What is printed by the following code?

```
count = 5
count += 1
count /= 2
print(count)
```

✓ 3

✗ 6

✗ 4

✗ 12

Objective 7 - Drive with Precision

Driving the Distance

It's time to put those 🔧wheel encoders to their proper use: *controlling the wheels!*

Now that you can count the slots, it only takes a small change to your code to make your bot drive *only* until a specified number of slots have gone by.

Counting *Slots* or Driving a *Distance?*

Okay, you'll get to converting between **slot-count** and *actual* distance in the next Objective.

- For now, just measure distance in "counts"
- After all *more* **counts** means the bot has traveled *farther*, right?

The **CodeTrek** will guide you in changing your code.

CodeTrek:

```
1  from botcore import *
2
3  # Choose a threshold value midway through the slot range
4  SLOT_THRESH = 2000
5
6  # Move SLOWLY forward!
7  motors.run(LEFT, 10)
8  motors.run(RIGHT, 10)
```

Be sure to drive **forward**

- Both wheels should have the same speed.
- Keep it slow to start with.

```
 9  motors.enable(True)
10
11  def drive_counts(n):
```

Define a new function that you can call to `drive()` with precision!

- You can use the Editor Shortcuts to move your existing block of code inside this funcdtion.
- Select the code and press TAB to indent it properly inside your function.

```
12      """Drive forward for n counts"""
13      # Initialize a variable to hold the previous encoder value
14      was_slot = False
15
16      # Initialize a variable to hold the slot count
17      count = 0
18
19      while count <= n:
```

No more *infinite* *loop !*

- Now your `while` loop should stop after `n` counts.

```
20          val = enc.read(LEFT)
21          is_slot = val > SLOT_THRESH
22
23          # Print the value only if it has changed
24          if is_slot != was_slot:
25
26              # Transitioned to/from a slot!
27              count += 1
28              print('Count is ', count)
29
30              # Remember the previous value processed.
31              was_slot = is_slot
32
33  # Drive for 1 full wheel rotation
34  drive_counts(40)
```

Don't forget to CALL your new function!

The "def" above just *defined* the function so Python knows about it.

- Now you have to ***call*** it to actually run that code.
- Be sure to pass your function the number of `counts` you want to move forward.

```
35
```

Hint:

- Be sure to drive *exactly* 40 counts.
  - That's one revolution of the wheel.
  - After that, your program ends, motor stops, and you're spot-on!

Goal:

- **Drive forward 40 counts**

  That's one full rotation of the wheel, then *STOP!*
  - You can watch the spokes as your bot drives slowly forward, and verify that it goes 360°

Tools Found: Wheel Encoders, Functions, Editor Shortcuts, Loops

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  # Choose a threshold value midway through the slot range
5  SLOT_THRESH = 2000
6
7  # Move SLOWLY forward!
8  motors.run(LEFT, 10)
9  motors.run(RIGHT, 10)
10 motors.enable(True)
11
12 def drive_counts(n):
13     """Drive forward for n counts"""
14     # Initialize a variable to hold the previous encoder value
15     was_slot = False
16
17     # Initialize a variable to hold the slot count
18     count = 0
19
20     while count < n:
21         val = enc.read(LEFT)
22         is_slot = val > SLOT_THRESH
23
24         # Print the value only if it has changed
25         if is_slot != was_slot:
26
27             # Transitioned to/from a slot!
28             count += 1
29             print('Count is ', count)
30
31             # Remember the previous value processed.
32             was_slot = is_slot
33
34 # Drive for 1 full wheel rotation
35 drive_counts(40)
```

Objective 8 **- Sensing Centimeters**

Measure Up!

It's time to convert those "counts" into real measurements.

- Whether it's on a highway or a basketball court, you're going to have to navigate using standard units of distance.
- CodeBot's size makes *centimeters* (cm) a nice unit of measurement.
  - *(Overall length of CodeBot is about 15cm)*

Counts to centimeters

How are you going to convert counts to centimeters?

- The distance around a **circle** is called the *circumference*.
- You might recall the math: $circumference = \pi * diameter$

Oh yeah, π "Pi" - that's like 3.14 or something?

Close, but you can do even better. 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_CM = 6.5
```

```
WHEEL_CIRC = (math.pi * WHEEL_DIA_CM)
```

The code above gives you the accurate *circumference* of CodeBot's wheel, in centimeters.

- Using this knowledge, the **CodeTrek** will guide you as you add the capability to drive a specified distance in *centimeters*, not just *counts!*
- Check the *Hints* for more information on the calculations.

CodeTrek:

```
1  from botcore import *
2  import math
3
4  # Choose a threshold value midway through the slot range
5  SLOT_THRESH = 2000
6  WHEEL_DIA_CM = 6.5
7  WHEEL_CIRC_CM = math.pi * WHEEL_DIA_CM
8  COUNTS_PER_REV = 40
```

Define the Constants

The wheel *diameter* measures **6.5cm**, and doesn't change.

- From that go ahead and calculate the *circumference* - it won't change either!
- And for readability define the number of counts for a full revolution as a constant also.

```
9
10 def drive_counts(n):
```

Your `drive_counts()` function can stay the same.

```
11     """Drive forward for n counts"""
12     # Initialize a variable to hold the previous encoder value
13     was_slot = False
14
15     # Initialize a variable to hold the slot count
16     count = 0
17
18     while count <= n:
19         val = enc.read(LEFT)
20         is_slot = val > SLOT_THRESH
21
22         # Print the value only if it has changed
23         if is_slot != was_slot:
24
25             # Transitioned to/from a slot!
26             count += 1
27             print('Count is ', count)
28
29             # Remember the previous value processed.
30             was_slot = is_slot
31
32 def cm_to_counts(cm):
33     return cm * (COUNTS_PER_REV / WHEEL_CIRC_CM)
```

Convert *centimeters* to *counts*.

- Multiply by the ratio of *counts/cm* based on a full wheel revolution.
- *See the **hints** for more on this calculation.*

```
34
35 def drive_dist(cm, power_lft, power_rt):
36     """Drive forward 'cm' centimeters at specified motor power"""
```

A new function to drive a specified distance in *real* centimeters!

```
37     counts = cm_to_counts(cm)
38
```

```
39      # Start the motors at the given left/right power levels
40      # TODO
```

Add code here to set the power and enable the motors.

```
41
42      drive_counts(counts)
43
44      # Disable the motors to stop moving now
45      # TODO
```

Add code to STOP those motors!

```
46
47  # Drive 100cm forward at 30% power
48  drive_dist(100, 30, 30)
```

Try moving forward for 100cm

- That's 1 meter for CodeBot... *One giant leap for your navigation code!*

Hint:

- ## Converting centimeters to counts
  - You just need to multiply the centimeters by a number...
  - What number?
  - A *ratio*: ***How many counts per centimeter?***

You know the *counts* for a full revolution = 40
You know the *centimeters* for a full revolution = π * 6.5 ≈ 20.4

$$counts\ per\ cm = \left(\frac{40\ counts}{1\ \cancel{rev}}\right) \cdot \left(\frac{1\ \cancel{rev}}{20.4\ cm}\right)$$

So that's about **1.96 counts per cm**

- Multiply the distance by 1.96 to get the number of counts!

Use more exact numbers in your code!

Use π from the math module and calculate the wheel *circumference*.

Goal:

- Drive your codebot to *Checkpoint 1*, and stop.
  - The small black lines on the floor are 10cm apart. The larger ones are 1m apart.
  - Stop your motors as soon as you contact the Checkpoint.

Tools Found: Constants, Math Module, Wheel Encoders, Readability, Motors

Solution:

```
1  from botcore import *
2  import math
3
4  # Choose a threshold value midway through the slot range
5  SLOT_THRESH = 2000
6  WHEEL_DIA_CM = 6.5
```

```
7  WHEEL_CIRC_CM = math.pi * WHEEL_DIA_CM
8  COUNTS_PER_REV = 40
9
10 def drive_counts(n):
11     """Drive forward for n counts"""
12     # Initialize a variable to hold the previous encoder value
13     was_slot = False
14
15     # Initialize a variable to hold the slot count
16     count = 0
17
18     while count <= n:
19         val = enc.read(LEFT)
20         is_slot = val > SLOT_THRESH
21
22         # Print the value only if it has changed
23         if is_slot != was_slot:
24
25             # Transitioned to/from a slot!
26             count += 1
27             print('Count is ', count)
28
29             # Remember the previous value processed.
30             was_slot = is_slot
31
32 def cm_to_counts(cm):
33     return (cm / WHEEL_CIRC_CM) * COUNTS_PER_REV
34
35 def drive_dist(cm, power_lft, power_rt):
36     """Drive forward 'cm' centimeters at specified motor power"""
37     counts = cm_to_counts(cm)
38     motors.run(LEFT, power_lft)
39     motors.run(RIGHT, power_rt)
40     motors.enable(True)
41     drive_counts(counts)
42     motors.enable(False)
43
44 # Drive N cm forward
45 drive_dist(140, 30, 30)
46
```

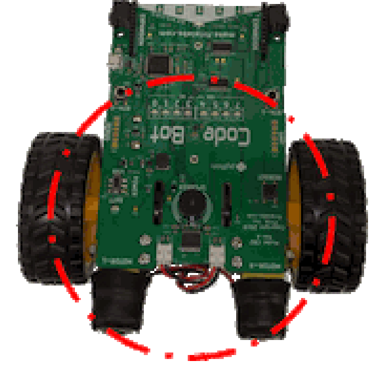
Objective 9 **- Free Throw Rotation**

Rotating with Precision

You know how to drive the wheels a precise distance.

- But how to turn that into a rotation?
- Actually, your `drive_dist()` function is already doing most of the work!

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!

Direction of Rotation: *Clockwise* or *Counter-clockwise?*

Now that you know how to calculate the **distance** required to rotate a given *angle*, what about specifying the **direction**?

- 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	-	+

CodeTrek:

```
1  from botcore import *
2  import math
3
4  # Choose a threshold value midway through the slot range
5  SLOT_THRESH = 2000
6  WHEEL_DIA_CM = 6.5
7  WHEEL_CIRC_CM = math.pi * WHEEL_DIA_CM
8  COUNTS_PER_REV = 40
9  TRACK_WIDTH = 11.7
10 TRACK_CIRC_CM = math.pi * TRACK_WIDTH
```

A couple more constants:

- CodeBot's (track width) in centimenters. This is the diameter of the circle those wheels trace out when your bot rotates.
- Pre-calculate the *circumference* of that circle. You're gonna need it for *rotation!*

```
11
12 def drive_counts(n):
13     """Drive forward for n counts"""
14     # Initialize a variable to hold the previous encoder value
15     was_slot = False
16
17     # Initialize a variable to hold the slot count
18     count = 0
19
20     while count <= n:
21         val = enc.read(LEFT)
22         is_slot = val > SLOT_THRESH
23
24         # Print the value only if it has changed
25         if is_slot != was_slot:
26
27             # Transitioned to/from a slot!
28             count += 1
29             print('Count is ', count)
30
31             # Remember the previous value processed.
32             was_slot = is_slot
33
34 def cm_to_counts(cm):
35     return (cm / WHEEL_CIRC_CM) * COUNTS_PER_REV
36
37 def drive_dist(cm, power_lft, power_rt):
38     """Drive forward 'cm' centimeters at specified motor power"""
39     counts = cm_to_counts(cm)
40     motors.run(LEFT, power_lft)
41     motors.run(RIGHT, power_rt)
42     motors.enable(True)
43     drive_counts(counts)
44     motors.enable(False)
45
46 def rotate_deg(deg, power):
47     """Rotate +/- degrees at specified power"""
```

Define a new function that **rotates** the bot by the given number of *degrees* at the given motor *power* level.

- Since you're rotating, both motors will have *equal* power in *opposite* directions.
- Clockwise is positive (+deg), Counterclockwise is negative (-deg)

```
48     # Get the direction as a signed +1 or -1 factor.
49     direction = deg / abs(deg)
```

You need a "direction" sign to multiply by the motor speeds.

- A good way to get the sign of a number is to divide it by the *absolute value* of itself.

Consider: $\frac{N}{N} = +1$

But: $\frac{N}{|N|} = \pm 1$

```
50
51     # Calculate the distance each wheel must travel.
52     dist = TRACK_CIRC_CM * deg / 360
```

What fraction the *track tircumference* will your wheels travel?

- All the way would be 360°/360°
- Half way would be 180°/360°

You get the idea...

```
53
54     # Move out! Positive direction means clockwise.
55     drive_dist(dist, power * direction, -power * direction)
```

You have your **distance!**

- Now feed it to your trusty `drive_dist()` function.

Notice how the `direction` works? When it's negative it flips the *sign* of each motor power, reversing your rotation.

```
56
57
58 # Turn 90 degrees, then drive forward to the checkpoint
59 # TODO
```

Your Turn

Call some functions!

- *Get thee to a Checkpoint!*

Hint:

- ## Expect Variations in Rotation

  You may have to try a few times to get the rotation angle just as you need it.

  - The 🔧motors have a small and variable delay turning on and off.
  - The 🔧wheel encoders won't always start with the same rotation.

Goal:

- **Center-up behind the line**
  Rotate and Drive to **Checkpoint 1** midway between the *free throw line* (where you start) and the perimeter of the *free throw circle* (radius = 140cm).
  - *Stop your bot exactly **70cm** from the line.*

Tools Found: Motors, Constants, Functions

Solution:

```
1  from botcore import *
2  import math
3
4  # Choose a threshold value midway through the slot range
5  SLOT_THRESH = 2000
6  WHEEL_DIA_CM = 6.5
7  WHEEL_CIRC_CM = math.pi * WHEEL_DIA_CM
8  COUNTS_PER_REV = 40
9  TRACK_WIDTH = 11.7
10 TRACK_CIRC_CM = math.pi * TRACK_WIDTH
11
12 was_slot = False
13
14 def drive_counts(n):
15     """Drive forward for n counts"""
16     # Initialize a variable to hold the previous encoder value
17     # was_slot = False
18     global was_slot
19
20     # Initialize a variable to hold the slot count
21     count = 0
22
23     while count <= n:
24         val = enc.read(LEFT)
25         is_slot = val > SLOT_THRESH
26
27         # Print the value only if it has changed
28         if is_slot != was_slot:
29
30             # Transitioned to/from a slot!
31             count += 1
32             print('Count is ', count)
33
34             # Remember the previous value processed.
35             was_slot = is_slot
36
37 def cm_to_counts(cm):
38     return (cm / WHEEL_CIRC_CM) * COUNTS_PER_REV
39
40 def drive_dist(cm, power_lft, power_rt):
41     """Drive forward 'cm' centimeters at specified motor power"""
42     counts = cm_to_counts(cm)
43     motors.run(LEFT, power_lft)
44     motors.run(RIGHT, power_rt)
45     motors.enable(True)
46     drive_counts(counts)
47     motors.enable(False)
48
49 def rotate_deg(deg, power):
50     """Rotate +/- degrees at specified power"""
51     # Get the direction as a signed +1 or -1 factor.
52     direction = deg / abs(deg)
53
54     # Calculate the distance each wheel must travel.
55     dist = TRACK_CIRC_CM * deg / 360
56
57     # Move out! Positive direction means clockwise.
58     drive_dist(dist, power * direction, -power * direction)
59
60
61 # Drive N cm forward
62 rotate_deg(90, 10)
63 drive_dist(70, 30, 30)
64
```

Objective 10 **- The Need for Speed**

Get your 'bot Up to *Speed!*

Now that you can measure **distance**, the next step is to measure your ***speed.***

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**.
- *(but go with **Centimeters per Second** for this Mission.)*

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 `enc.read(LEFT)` *very rapidly*, every time through the loop to track `count`.
- 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 = t_stop - t_start
print("That took ", t_diff, " milliseconds!")
```

CodeTrek:

```
 1  from botcore import *
 2  import math
 3  from time import ticks_ms
 4
 5  # Choose a threshold value midway through the slot range
 6  SLOT_THRESH = 2000
 7  WHEEL_DIA_CM = 6.5
 8  WHEEL_CIRC_CM = math.pi * WHEEL_DIA_CM
 9  COUNTS_PER_REV = 40
10  TRACK_WIDTH = 11.7
11  TRACK_CIRC_CM = math.pi * TRACK_WIDTH
12  POLL_MS = 100
```

The "polling interval" in milliseconds.

- This controls how often you will calculate and display the speed.
- 10 times per second should work great. That's 100ms.

```
13
14  def drive_counts(n):
15      """Drive forward for n counts"""
16      # Initialize a variable to hold the previous encoder value
17      was_slot = False
18
19      # Initialize a variable to hold the slot count
20      count = 0
21
22      # Initialize variables for polling the speed
23      count_poll = 0  # count at previous poll time
24      t_poll = ticks_ms() + POLL_MS  # Next poll time
```

While you're driving...

- Keep track of the next time you need to "poll" the *speed*.

- You'll also need to calculate how far the bot has traveled since the last poll, so saving the previous count would be good.

```
25
26      while count <= n:
27          val = enc.read(LEFT)
28          is_slot = val > SLOT_THRESH
29
30          # Print the value only if it has changed
31          if is_slot != was_slot:
32
33              # Transitioned to/from a slot!
34              count += 1
35              # print('Count is ', count)
```

Comment-out this line

- De-clutter your console output!

```
36
37              # Remember the previous value processed.
38              was_slot = is_slot
39
40          # Periodically poll speed
41          t_now = ticks_ms()
42          if t_now > t_poll:
43              # Schedule the next poll
44              t_poll = t_now + POLL_MS
```

Is it *POLL* time?

See how the code inside this `if` block runs once every POLL_MS?

- Each time around the `while` loop you check the time...
- When `t_poll` finally arrives, drop into the `if` block!
- Then schedule the next *poll* and do any other *periodic* tasks.

```
45
46              # Calculate speed in cm / sec
47              dist_cm = counts_to_cm(count - count_poll)
48              tm_sec = POLL_MS / 1000
49              speed = # TODO
50              print(f'speed = {speed} cm/s')
```

Calculate and display the **speed** in centimeters per second.

- Remember the equation for *speed?*
- The value printed needs to be an integer.
- The built-in `round()` function will take care of that.
- Use an f-string to format your `print()` output this time!

```
51
52              # Remember count to calculate distance in next poll
53              count_poll = count
54
55
56
57  def cm_to_counts(cm):
58      return (cm / WHEEL_CIRC_CM) * COUNTS_PER_REV
59
60  def counts_to_cm(counts):
61      return counts * WHEEL_CIRC_CM / COUNTS_PER_REV
```

To display the speed, you need to convert counts to centimeters.

- This is the inverse of the `cm_to_counts()` function you wrote above!

```
62
63  def drive_dist(cm, power_lft, power_rt):
```

```
64      """Drive forward 'cm' centimeters at specified motor power"""
65      counts = cm_to_counts(cm)
66      motors.run(LEFT, power_lft)
67      motors.run(RIGHT, power_rt)
68      motors.enable(True)
69      drive_counts(counts)
70      motors.enable(False)
71
72  def rotate_deg(deg, power):
73      """Rotate +/- degrees at specified power"""
74      # Get the direction as a signed +1 or -1 factor.
75      direction = deg / abs(deg)
76
77      # Calculate the distance each wheel must travel.
78      dist = TRACK_CIRC_CM * deg / 360
79
80      # Move out! Positive direction means clockwise.
81      drive_dist(dist, power * direction, -power * direction)
82
83  # Drive forward, then change speeds and go again
84  drive_dist(??)
85  drive_dist(??)
```

A Two-Part Journey

Check the goals and be sure to drive far enough and with enough difference in speed to accomplish your mission!

Hints:

- **Changing Speeds**

  The motor power level determines your speed on flat ground.

  - Drive for some distance at one power level.
  - Increase the power, and drive a bit farther.

- **Printing an Integer Value**

  One way to do this is to convert the number to an integer *before* converting it to a string.

  - Use the `round()` built-in for this.

Goals:

- **Calculate Speed**

  Use the `ticks_ms()` function from the time module.

- **Console *Speedometer!***

  Display your speed.

  - `print()` to the console `speed = N cm/s`
  - The `N` above should be an integer speed value for your current speed in centimeters per second.
  - Check and *print* speed every 100ms

- **Drive at *two* different speeds**

  I want to see two speeds different by at least 10 cm/s in your output.

  - Spend a minimum of 2 seconds at each speed

- **Use an f-string to format your output**

  Mastering this powerful string formatting technique will make it easy for you to create clear and concise output messages.

Tools Found: Time Module, Print Function, int, String Formatting, str, Comments, Built-In Functions, Functions

Solution:

```
from botcore import *
import math
from time import ticks_ms

# Choose a threshold value midway through the slot range
SLOT_THRESH = 2000
WHEEL_DIA_CM = 6.5
WHEEL_CIRC_CM = math.pi * WHEEL_DIA_CM
COUNTS_PER_REV = 40
TRACK_WIDTH = 11.7
TRACK_CIRC_CM = math.pi * TRACK_WIDTH
POLL_MS = 100

was_slot = False

def drive_counts(n):
    """Drive forward for n counts"""
    # Initialize a variable to hold the previous encoder value
    # was_slot = False
    global was_slot

    # Initialize a variable to hold the slot count
    count = 0
    count_poll = 0

    t_poll = ticks_ms() + POLL_MS

    while count <= n:
        val = enc.read(LEFT)
        is_slot = val > SLOT_THRESH

        # Print the value only if it has changed
        if is_slot != was_slot:

            # Transitioned to/from a slot!
            count += 1
            # print('Count is ', count)

            # Remember the previous value processed.
            was_slot = is_slot

        # Periodically poll speed
        t_now = ticks_ms()
        if t_now > t_poll:
            # Calculate speed in cm / sec
            speed = counts_to_cm(count - count_poll) / (POLL_MS / 1000)
            print(f'speed = {round(speed)} cm/s')

            # Remember count, and schedule next poll
            count_poll = count
            t_poll = t_now + POLL_MS


def cm_to_counts(cm):
    return (cm / WHEEL_CIRC_CM) * COUNTS_PER_REV

def counts_to_cm(counts):
    return counts * WHEEL_CIRC_CM / COUNTS_PER_REV

def drive_dist(cm, power_lft, power_rt):
    """Drive forward 'cm' centimeters at specified motor power"""
    counts = cm_to_counts(cm)
    motors.run(LEFT, power_lft)
    motors.run(RIGHT, power_rt)
    motors.enable(True)
    drive_counts(counts)
    motors.enable(False)

```

```
69 def rotate_deg(deg, power):
70     """Rotate +/- degrees at specified power"""
71     # Get the direction as a signed +1 or -1 factor.
72     direction = deg / abs(deg)
73
74     # Calculate the distance each wheel must travel.
75     dist = TRACK_CIRC_CM * deg / 360
76
77     # Move out! Positive direction means clockwise.
78     drive_dist(dist, power * direction, -power * direction)
79
80
81 # Drive forward, then change speeds and go again
82 drive_dist(20, 10, 10)
83 drive_dist(40, 50, 50)
84
```

***Quiz 3* - Speed Round**

Question 1: Which **two** of the following are `True`?

- ✓ `abs(-5) == 5`
- ✓ `abs(-17) / -17 == -1`
- ✗ `abs(5) == -5`
- ✗ `abs(1 / 2) == 0`

Question 2: About how much *time* passes between printing **"Begin"** and **"End"** when the code below is run?

```
tm = ticks_ms() + 1000

print("Begin")
while True:
    if ticks_ms() > tm:
        break

print("End")
```

- ✓ 1 second
- ✗ 1 minute
- ✗ 1 millisecond
- ✗ 100 milliseconds

Question 3: Which of the following is output by this code snippet?

```
import math
print(f'{math.pi:.3f}')
```

- ✓ 3.142
- ✗ 3.14
- ✗ 03.14

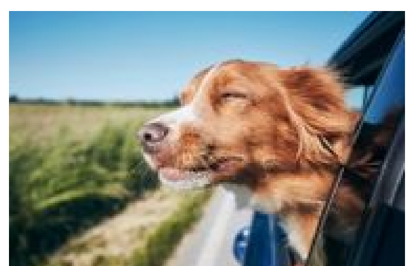
Question 4: Which of the following is output by this code snippet?

```
num = 10
print(f'0b{num:08b}')
```

- ✓ 0b00001010
- ✗ 0b1010
- ✗ 0b00000010
- ✗ 18

Objective 11 - Speed Trap

Cruise Control



You've used the 🔧wheel encoders to measure distance *and* angle of rotation.

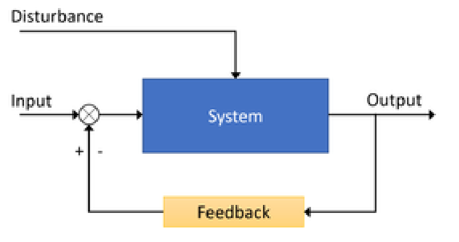
But what if you want your bot to drive at a particular **speed**?

- So far your *speed* has just been based on the 🔧motor *power* level.
- But that's not very accurate! Different surfaces, inclines, and battery power will all affect your *speed* even with a constant motor *power* level.

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?

Systems Engineering

Deep breath... bear with me here! The diagrams below show the drive control for your CodeBot. Take a minute to see how this helps you craft the code for *cruise control*.



Closed-Loop Control System

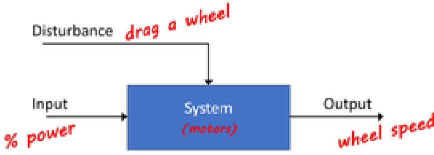
Closed Loop Control automates control of a *System* by sensing the *Output* state and comparing it to the desired state (*Input*). A *Feedback* loop continuously adjusts the *System* to keep the *error* (difference between *Input* and *Output*) close to *zero*.

- **Input** → Desired Speed
- **System** → 🔧Motors
- **Output** → Actual Speed
- **Feedback** → 🔧Wheel Encoders
- **Disturbance** → Friction, terrain, etc.

Your Code is *"Open Loop!"*

Right now you're *displaying* the **speed**, but your code is **not** using it to adjust the **power**.

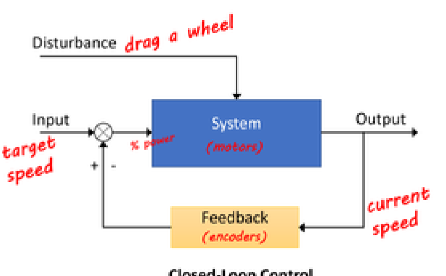
- Any *"Disturbance"* that happens will affect the *Output* (speed).
- And your *Input* in *raw* ***% power*** is only *loosely* related to the *Output* ***speed***.



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:

$$err = (Input - Output) \cdot K_p$$

- *Output* and *Input* are **speeds**.
- K_p is a 🔧constant *you choose* for the amount of *Proportional* feedback.

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`.
- **System** → `power` to the 🔧motors.

  Ex: Code to apply *feedback* to the motors.

```
# Calculate: err = (Input - Output)
err = target_speed - cur_speed

# Apply feedback to System (adjust motor power)
power += err * Kp
motors.run(LEFT, power)
motors.run(RIGHT, power)
```

Consider how the code above works when your 'bot is going **slower** than the desired `target_speed`:

- `target_speed > cur_speed` 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!

- First, use the *File → Save As* menu to save your code to a new file called "enc_speed.py"

CodeTrek:

```
1  from botcore import *
2  import math
3  from time import ticks_ms
4
5  # Choose a threshold value midway through the slot range
6  SLOT_THRESH = 2000
7  WHEEL_DIA_CM = 6.5
8  WHEEL_CIRC_CM = math.pi * WHEEL_DIA_CM
9  COUNTS_PER_REV = 40
10 TRACK_WIDTH = 11.7
11 TRACK_CIRC_CM = math.pi * TRACK_WIDTH
12 POLL_MS = 200
13 Kp = 0.3  # Proportional feedback: Adjust this.
```

Another constant

This controls the amount of proportional feedback.

- **Higher** values will make the motors respond more strongly to speed errors.
  - Too high and your speed will be *janky!*
- **Lower** values *smooth* out the speed changes, but it may take longer to come up to speed.

```
14
15 def drive_speed(dist, target_speed):
```

Replace your `drive_counts()` with this!

Re-write your old function as shown here.

- Check out these new parameters for *real* distance and speed.

```
16     """Drive the given dist (cm) at target_speed (cm/s)"""
17     # Initialize a variable to hold the previous encoder value
18     was_slot = False
19
20     # Initialize a variable to hold the slot count
21     count = 0
22
23     # Initialize variables for polling the speed
24     count_poll = 0  # count at previous poll time
25     t_poll = ticks_ms() + POLL_MS  # Next poll time
26
27     # Initialize a variable to hold the current motor power
28     power = 0
```

This time your code *calculates* the motor power *dynamically!*

- Start with zero power

```
29
30      # Convert distance to "counts"
31      n = cm_to_counts(dist)
```

Convert the given distance into wheel encoder *counts*

```
32
33      motors.enable(True)
```

Your `drive_speed()` function takes full control of the motors.

- Be sure to *enable* and *disable* them.

```
34
35      while count <= n:
36          # Check Encoder
37          val = enc.read(LEFT)
38          is_slot = val > SLOT_THRESH
39          if is_slot != was_slot:
40              # Transitioned to/from a slot!
41              count += 1
42
43              # Remember the previous value processed.
44              was_slot = is_slot
45
46          # Check Speed
47          t_now = ticks_ms()
48          if t_now > t_poll:
49              # Schedule the next poll
50              t_poll = t_now + POLL_MS
51
52              # Calculate speed in cm / sec
53              dist_cm = counts_to_cm(count - count_poll)
54              count_poll = count
55              tm_sec = POLL_MS / 1000
56              cur_speed = dist_cm / tm_sec
```

Calculate the *speed* every POLL.

- No need to `round()` it. More precision is better for controlling the motors, right?

```
57
58              # Feedback Loop! Adjust power in proportion to error.
59              err = target_speed - cur_speed
60              power += err * Kp
61              motors.run(LEFT, power)
62              motors.run(RIGHT, power)
```

Your Control System

Just like those *Systems Engineering* diagrams!

- You're controlling the power based on the speed.
- That's *closed-loop control* dude!

```
63
64              # Log stats to console
65              # TODO: Better number formatting
66              print(f'speed={cur_speed}, power={power}%')
```

These numbers are floats.

- They can get pretty long on the console.
- Maybe use some string formatting to clean these up!?

```
67
```

```
68      motors.enable(False)
69
70
71  def cm_to_counts(cm):
72      return (cm / WHEEL_CIRC_CM) * COUNTS_PER_REV
73
74  def counts_to_cm(counts):
75      return counts * WHEEL_CIRC_CM / COUNTS_PER_REV
76
77  # NOTE: Deleted drive_dist() and rotate_deg() functions!
78
79  # Take a trip!
80  drive_speed(160, 10)
```

Notice some code was removed above. Keep it tidy :-)

Fancy a drive?

Enter the *speed trap* if you dare!

Hint:

- **Formatting your Console Output**

  To print a float with just a single digit after the decimal point, use the `{:.1f}` *format specifier* as described in string formatting.

  Ex:

  ```
  val = 7.654321
  print(f'Number = {val:.1f}')
  ```

  The above prints `Number = 7.7`

Goals:

- Maintain a constant speed of 25 cm/sec through both Checkpoints
- Display your *speed* and *motor power* on the console as you go.
  - Format each float value with just a single digit after the decimal point, so I can read them easily.
  - See the *Hints* if needed for this.

Tools Found: Wheel Encoders, Motors, Constants, float, Functions, Parameters, Arguments, and Returns, Print Function, String Formatting

Solution:

```
1   from botcore import *
2   import math
3   from time import ticks_ms
4
5   # Choose a threshold value midway through the slot range
6   SLOT_THRESH = 2000
7   WHEEL_DIA_CM = 6.5
8   WHEEL_CIRC_CM = math.pi * WHEEL_DIA_CM
9   COUNTS_PER_REV = 40
10  TRACK_WIDTH = 11.7
11  TRACK_CIRC_CM = math.pi * TRACK_WIDTH
12  POLL_MS = 200
13  Kp = 0.4  # Proportional feedback: Adjust this. #@1
14
15  def drive_speed(dist, target_speed):  #@2
16      """Drive the given dist (cm) at target_speed (cm/s)"""
17      # Initialize a variable to hold the previous encoder value
18      was_slot = False
19
```

```
20      # Initialize a variable to hold the slot count
21      count = 0
22
23      # Initialize variables for polling the speed
24      count_poll = 0  # count at previous poll time
25      t_poll = ticks_ms() + POLL_MS  # Next poll time
26
27      # Initialize a variable to hold the current motor power
28      power = 0  #@3
29
30      # Convert distance to "counts"
31      n = cm_to_counts(dist)  #@4
32
33      motors.enable(True)  #@5
34
35      while count <= n:
36          # Check Encoder
37          val = enc.read(LEFT)
38          is_slot = val > SLOT_THRESH
39          if is_slot != was_slot:
40              # Transitioned to/from a slot!
41              count += 1
42
43              # Remember the previous value processed.
44              was_slot = is_slot
45
46          # Check Speed
47          t_now = ticks_ms()
48          if t_now > t_poll:
49              # Schedule the next poll
50              t_poll = t_now + POLL_MS
51
52              # Calculate speed in cm / sec
53              dist_cm = counts_to_cm(count - count_poll)
54              count_poll = count
55              tm_sec = POLL_MS / 1000
56              cur_speed = dist_cm / tm_sec  #@6
57
58              # Feedback Loop! Adjust power in proportion to error.
59              err = target_speed - cur_speed
60              power += err * Kp
61              motors.run(LEFT, power)
62              motors.run(RIGHT, power)  #@7
63
64              # Log stats to console
65              print(f'speed={cur_speed:.1f}, power={power:.1f}%')
66
67      motors.enable(False)
68
69
70  def cm_to_counts(cm):
71      return (cm / WHEEL_CIRC_CM) * COUNTS_PER_REV
72
73  def counts_to_cm(counts):
74      return counts * WHEEL_CIRC_CM / COUNTS_PER_REV
75
76  # NOTE: Deleted drive_dist() and rotate_deg() functions!
77
78  # Take a trip!
79  drive_speed(160, 25)  #@9
```

Objective 12 **- Arc de CodeBot**

A Triumphant Finish!

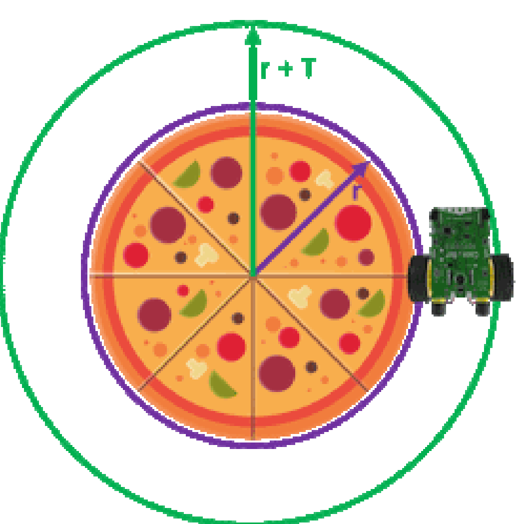
This final Objective will put your Python coding skills *and* your knowledge of the 🔧wheel encoders to the test!

> *You can DO this!*

All Wheel Drive

So far you've gotten a lot done with just the LEFT side encoder.

- But what if you want to drive in a ***circle?***
- Or a curvy line, *or* to drive straight over uneven surfaces where the wheels need *independent control.*



Making an Arc

An "arc" is just a portion of the perimeter of a *circle*.

- Kind of like the crust of a pizza slice!
- A 90° arc would be 1/4 of the pie.
- ...and 360° would go around the whole pizza!

Around the Pizza

Say you wanted to drive CodeBot around the pizza shown above.

- Notice the radius of the two circles (*purple* and *green*) differs by T, CodeBot's **Track Width**.
- The *RIGHT* wheel would need to go farther than the *LEFT* wheel. Right?

The ratio of speeds is the same as the ratio of the perimeters of the *purple* and *green* circles.

$$\frac{DIST_{left}}{DIST_{right}} = \frac{\cancel{2\pi} \cdot r}{\cancel{2\pi} \cdot (r+T)} = \frac{r}{r+T}$$

So if you can drive with different LEFT/RIGHT wheel speeds, you can *precisely* circumnavigate the pizza!

Refactor Your Code!

You need to modify your `drive_speed()` function to accept `speed_left` and `speed_right`.

- That means you'll be checking *both* wheel encoders as you drive.
- And that's *twice* as many *state variables* tracking slot, count, speed, etc.

Relax, the **CodeTrek** will guide you in this refactoring work.

CodeTrek:

```
1  from botcore import *
2  import math
3  from time import ticks_ms
4
5  # Choose a threshold value midway through the slot range
6  SLOT_THRESH = 2000
7  WHEEL_DIA_CM = 6.5
8  WHEEL_CIRC_CM = math.pi * WHEEL_DIA_CM
9  COUNTS_PER_REV = 40
10 TRACK_WIDTH = 11.7
11 TRACK_CIRC_CM = math.pi * TRACK_WIDTH
12 POLL_MS = 200
13 Kp = 0.1
14
15 def drive_speed(dist, speed_left, speed_right):
16     """Drive the given dist (cm) at target speeds (cm/s)"""
```

Refactoring Begins!

Re-work your `drive_speed()` function.

- A control loop for both **LEFT** and **RIGHT** motors.

```
17     # Init global state variables
18     init_drive_state()
```

First, factor-out most of your *state* *variables*.

- The `init_drive_state()` function will handle creating the global variables you need.

- This allows other code you factor-out to access the *state*.

```
19
20      # Set global target_speed
21      target_speed[LEFT] = speed_left
22      target_speed[RIGHT] = speed_right
```

Accessing the global state:

- Now that you've called `init_drive_state()` you can access the variables it created.
- Set the `target_speed` for both encoders.

```
23
24      t_poll = ticks_ms() + POLL_MS
25
26      # Convert distance to "counts"
27      n = cm_to_counts(dist)
28
29      motors.enable(True)
30
31      while count[LEFT] <= n and count[RIGHT] <= n:
32          # Check Encoder
33          update_slot_count(LEFT)
34          update_slot_count(RIGHT)
```

Your `while` loop has 2 parts:

1. Read the encoders and update the slot count.
2. Poll the speed and update the motor power.

Factor-out both of those parts into their own functions!

- The *first* part is `update_slot_count()`

```
35
36          # Periodically poll speed and update motor power
37          t_now = ticks_ms()
38          if t_now > t_poll:
39              t_poll = t_now + POLL_MS
40
41              # Update speed and power
42              update_speed_power(LEFT)
43              update_speed_power(RIGHT)
```

The 2nd part of your `while` loop happens every *POLL interval*.

- Move that code to the `update_speed_power()` function.

```
44
45      motors.enable(False)
46
47  def init_drive_state():
48      """Initialize global state for driving with encoders"""
49      global was_slot, count, count_poll, speed, target_speed, power
```

Your First Factored Function

It creates several lists that track state: [LEFT, RIGHT]

- Remember **botcore** defines `LEFT = 0` and `RIGHT = 1`.
- Initialize all the state you need to manage *distance* and *speed*.

```
50
51      was_slot = [False, False]
52      count = [0, 0]              # Current encoder counts
53      count_poll = [0, 0]
54      speed = [0, 0]              # Current speed
55      target_speed = [0, 0]       # Desired speed
56      power = [0, 0]              # Current motor power levels
57
```

```
58  def update_slot_count(side):
59      """Check Encoder, update global count[] and was_slot[] lists"""
60      val = enc.read(side)
61      is_slot = val > SLOT_THRESH
62      if is_slot != was_slot[side]:
63          # Transitioned to/from a slot!
64          count[side] += 1
```

This code should look familiar!

- Basically it's the same code that was in your `while` loop to update the slot count.
- Now it uses the global state, and handles *both sides*.
- Notice how the augmented assignment saves even more typing here!

```
65
66          # Remember the previous value processed.
67          was_slot[side] = is_slot
68
69  def update_speed_power(side):
70      """This should be called every POLL_MS to update speed[] based on count[] and count_poll[].
71         Also updates power[] based on target_speed[] vs speed[], and sets motor accordingly.
72      """
```

This is factored-out from the POLL section of your `while` loop.

- These *speed* and *error feedback* calculations should look familiar.
- Notice how this function only deals with *one motor* at a time.
- Also, some code has been added to deal with *negative speeds*.
    - So this time, you're *"Reverse-Ready!"*

```
73      # Update speed
74      dist_cm = counts_to_cm(count[side] - count_poll[side])
75      tm_sec = POLL_MS / 1000
76      speed[side] = dist_cm / tm_sec
77      count_poll[side] = count[side]
78
79      speed_sign = target_speed[side] / abs(target_speed[side])
80
81      # Adjust power
82      err = abs(target_speed[side]) - speed[side]
83      power[side] = power[side] + err * Kp
84      motors.run(side, power[side] * speed_sign)
85
86  def cm_to_counts(cm):
87      return (cm / WHEEL_CIRC_CM) * COUNTS_PER_REV
88
89  def counts_to_cm(counts):
90      return counts * WHEEL_CIRC_CM / COUNTS_PER_REV
91
92  def drive_arc(dist, radius, speed):
93      """Drive in a counterclockwise arc with given radius"""
94      ratio = radius / (radius + TRACK_WIDTH)
95      drive_speed(dist, ratio * speed, speed)
```

This function implements the **arc** calculation.

- To keep it simple it only handles *counterclockwise* arcs.
- Later you can adapt it to go clockwise too!

```
96
97
98  FREE_THROW_RADIUS = 145  # cm  (center of circle)
99  PERIMETER_DISTANCE = 2 * math.pi * (FREE_THROW_RADIUS + 10)  # cm
#@10
100
101 drive_arc(PERIMETER_DISTANCE, FREE_THROW_RADIUS, 25)
#@11
```

Goal:

- **The Free Throw Circle is Your Pizza!**

  Make it all the way around the free-throw circle, hitting each Checkpoint

  - The weightlifting team has added some dumbbells to increase the challenge 😏
  - The radius of the free-throw circle is 145cm.

Tools Found: Wheel Encoders, Refactoring, undefined, Variables, Locals and Globals, list, State, Functions, Assignment, Constants

Solution:

```
1  from botcore import *
2  import math
3  from time import ticks_ms
4
5  # Choose a threshold value midway through the slot range
6  SLOT_THRESH = 2000
7  WHEEL_DIA_CM = 6.5
8  WHEEL_CIRC_CM = math.pi * WHEEL_DIA_CM
9  COUNTS_PER_REV = 40
10 TRACK_WIDTH = 11.7
11 TRACK_CIRC_CM = math.pi * TRACK_WIDTH
12 POLL_MS = 200
13 Kp = 0.1
14
15 def drive_speed(dist, speed_left, speed_right):
16     """Drive the given dist (cm) at target speeds (cm/s)"""  #@1
17     # Init global state variables
18     init_drive_state()  #@2
19
20     # Set global target_speed
21     target_speed[LEFT] = speed_left
22     target_speed[RIGHT] = speed_right  #@3
23
24     t_poll = ticks_ms() + POLL_MS
25
26     # Convert distance to "counts"
27     n = cm_to_counts(dist)
28
29     motors.enable(True)
30
31     while count[LEFT] <= n and count[RIGHT] <= n:
32         # Check Encoder
33         update_slot_count(LEFT)
34         update_slot_count(RIGHT)  #@4
35
36         # Periodically poll speed and update motor power
37         t_now = ticks_ms()
38         if t_now > t_poll:
39             t_poll = t_now + POLL_MS
40
41             # Update speed and power
42             update_speed_power(LEFT)
43             update_speed_power(RIGHT)  #@5
44
45     motors.enable(False)
46
47 def init_drive_state():
48     """Initialize global state for driving with encoders"""
49     global was_slot, count, count_poll, speed, target_speed, power  #@6
50
51     was_slot = [False, False]
52     count = [0, 0]            # Current encoder counts
53     count_poll = [0, 0]
54     speed = [0, 0]            # Current speed
55     target_speed = [0, 0]     # Desired speed
56     power = [0, 0]            # Current motor power levels
57
58 def update_slot_count(side):
59     """Check Encoder, update global count[] and was_slot[] lists"""  #@7
```

```
60      val = enc.read(side)
61      is_slot = val > SLOT_THRESH
62      if is_slot != was_slot[side]:
63          # Transitioned to/from a slot!
64          count[side] = count[side] + 1
65
66          # Remember the previous value processed.
67          was_slot[side] = is_slot
68
69  def update_speed_power(side):
70      """This should be called every POLL_MS to update speed[] based on count[] and count_poll[].
71         Also updates power[] based on target_speed[] vs speed[], and sets motor accordingly.
72      """  #@8
73      # Update speed
74      dist_cm = counts_to_cm(count[side] - count_poll[side])
75      tm_sec = POLL_MS / 1000
76      speed[side] = dist_cm / tm_sec
77      count_poll[side] = count[side]
78
79      speed_sign = target_speed[side] / abs(target_speed[side])
80
81      # Adjust power
82      err = abs(target_speed[side]) - speed[side]
83      power[side] = power[side] + err * Kp
84      motors.run(side, power[side] * speed_sign)
85
86  def cm_to_counts(cm):
87      return (cm / WHEEL_CIRC_CM) * COUNTS_PER_REV
88
89  def counts_to_cm(counts):
90      return counts * WHEEL_CIRC_CM / COUNTS_PER_REV
91
92  def drive_arc(dist, radius, speed):
93      """Drive in a counterclockwise arc with given radius"""
94      ratio = radius / (radius + TRACK_WIDTH)
95      drive_speed(dist, ratio * speed, speed)  #@9
96
97  FREE_THROW_RADIUS = 145  # cm  (center of circle)
98  OUTSIDE_DISTANCE = 2 * math.pi * (FREE_THROW_RADIUS + 15)  # cm
99
100 drive_arc(OUTSIDE_DISTANCE, FREE_THROW_RADIUS, 25)  #@10
```

***Mission 14* - Music Box**

Turn the CodeBot into a jukebox and learn about Python's file operations!

***Objective 1* - Tune That Dial**

Play some notes!!

The CodeBot 🔧speaker can play simple audio frequencies.

- These can be strung together to make *music*...

The table below shows the frequencies for some common musical notes.

Note	Frequency (Hz)		Note	Frequency (Hz)		Note	Frequency (Hz)
C_5	523		C_6	1047		C_7	2093
D_5	587		D_6	1175		D_7	2349
E_5	659		E_6	1319		E_7	2637
F_5	698		F_6	1397		F_7	2794
G_5	784		G_6	1568		G_7	3136
A_5	880		A_6	1760		A_7	3520
B_5	988		B_6	1976		B_7	3951

Your bot can play any of these notes. Why don't you try a few!?

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  freqs = {
5      "C": 1047,
6      "D": 1175,
7      "E": 1319,
8      "F": 1397,
9      "G": 1568,
10     "A": 1760
11 }
```

Here is a dictionary of notes that you can use throughout the lesson.

To get the *frequency* of a **G6** you would use:

```
freq = freqs["G"]
```

```
12
13 freq = freqs["G"]
```

Get the frequency of a **G6**.

```
14 spkr.pitch(freq)
```

Play the **G6** on your CodeBot speaker!

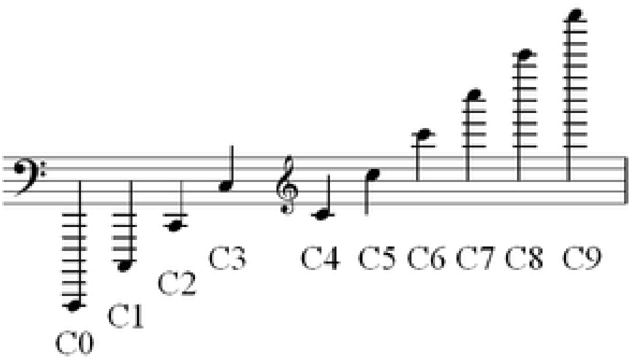
```
15 sleep(1.0)
16 # TODO: Play an E6
17 sleep(1.0)
18 # TODO: Play an A6
19 sleep(1.0)
20 spkr.off()
21
```

Hint:

- ## Scientific Pitch Notation

  Writing musical notes in the form C_5 is called Scientific Pitch Notation.

  The *letter* is the note, and the *number* is the *octave*. So for example, C_4 is **middle C** on the piano.



Goals:

- Play a G_6 (1568 Hz) on the `spkr`
- Play an E_6 (1319 Hz) on the `spkr`

- Play an A_6 (1760 Hz) on the `spkr`

Tools Found: Speaker, dictionary

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  freqs = {
5      "C": 1047,
6      "D": 1175,
7      "E": 1319,
8      "F": 1397,
9      "G": 1568,
10     "A": 1760
11 }
12
13 freq = freqs["G"]
14 spkr.pitch(freq)
15 sleep(1.0)
16 freq = freqs["E"]
17 spkr.pitch(freq)
18 sleep(1.0)
19 freq = freqs["A"]
20 spkr.pitch(freq)
21 sleep(1.0)
22 spkr.off()
23
```

Objective 2 - Twinkle, Twinkle

Time for your first composition!

Here is a list of notes that make up the song **Twinkle, Twinkle, Little Star**

- C_6, C_6, G_6, G_6, A_6, A_6, G_6, F_6, F_6, E_6, E_6, D_6, D_6, C_6

Put your notes in an actual Python 🔧list and use a `for` 🔧loop to play them:

```
notes = ["C", "C", "G", "G", ...]
for note in notes:
    # play note
```

Remember the *Metronome* Mission?

You can use `tempo` and `beat_duration` that you've already learned to control the timing.

Start with these values:

- `tempo = 100`
- `beat_duration = 0.3`

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds per beat
```

Define the `tempo` and `beat_duration` variables.

```
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
```

```
10      "E": 1319,
11      "F": 1397,
12      "G": 1568,
13      "A": 1760
14  }
15
16  notes = ["C", "C", "G"] # TODO: Finish this list
```

Complete this list of notes.

- It should play the song **Twinkle, Twinkle, Little Star**

```
17
18  # Loop through each note in notes
19  # TODO: Insert a for loop
```

Add a `for` loop to get each note from the `notes` list.

```
20      # Lookup the frequency of this note
21      f = # TODO: Get the frequency for the note
```

Lookup the given `note` in your dictionary to find its `freq`uency.

```
22
23      # Play the note for the beat_duration
24      spkr.pitch(f)
25      sleep(beat_duration)
26
27      # Pause for articulation
28      spkr.off()
29      sleep(0.05)
```

Pause to give a little space between notes.

Goal:

- Play the following notes in order:
  - C, C, G, G, A, A, G, F, F, E, E, D, D, C

  Be sure to turn the `spkr.off()` between notes!!

Tools Found: list, Loops, dictionary

Solution:

```
1   from botcore import *
2   from time import sleep
3
4   tempo = 150 # beats per minute
5   beat_duration = 60 / tempo # seconds
6
7   freqs = {
8       "C": 1047,
9       "D": 1175,
10      "E": 1319,
11      "F": 1397,
12      "G": 1568,
13      "A": 1760
14  }
15
16  notes = ["C", "C", "G", "G", "A", "A", "G", "F", "F", "E", "E", "D", "D", "C"]
17
18  # Loop through each note in notes
```

```
19 for note in notes:
20     freq = freqs[note]
21     # play the note for the beat_duration
22     spkr.pitch(freq)
23     sleep(beat_duration)
24
25     # pause for articulation
26     spkr.off()
27     sleep(0.05)
```

***Objective 3* - Jingle Bells**

Time to get in the Holiday Spirit!!

Here are the notes for the song **Jingle Bells**:

- E_6, E_6, E_6, E_6, E_6, E_6, E_6, G_6, C_6, D_6, E_6 F_6, F_6, F_6, F_6, F_6, E_6, E_6, E_6, E_6, D_6, D_6, E_6, D_6, G_6

This time put the notes in a string separated by spaces like this:

- `text = "E E E E E E E G C D"`

Then you can use the Python split function to turn them into a `notes` list.

`split()` turns a string into a list.

- Every element in your list will be a **word** in the string.
- **Words** are separated by **spaces**.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
11     "F": 1397,
12     "G": 1568,
13     "A": 1760
14 }
15
16 text = "E E E E E E" # TODO: Finish this string
```

Add the rest of the notes to **Jingle Bells** here.

- Put a space between each note for `split()`

```
17
18 notes = # TODO: Use the split function here
```

Use the `split()` function on `text` to get a list of notes.

```
19
20 # Loop through each note in notes
21 for note in notes:
22     # Lookup the frequency of this note
23     f = freqs[note]
24
25     # Play the note for the beat_duration
26     spkr.pitch(f)
27     sleep(beat_duration)
28
29     # Pause for articulation
```

```
30      spkr.off()
31      sleep(0.05)
```

Goals:

- Use `text.split()` to create a Python list named `notes`.
- Play the following notes in order:
    - E, E, E, E, E, E, E, G, C, D, E, F, F, F, F, F, E, E, E, E, D, D, E, D, G

  Be sure to turn the `spkr.off()` between notes!!

Tools Found: str, list

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
11     "F": 1397,
12     "G": 1568,
13     "A": 1760
14 }
15
16 text = "E E E E E E E G C D E F F F F F E E E E D D E D G"
17
18 notes = text.split()
19
20 # loop through each note in notes
21 for note in notes:
22     freq = freqs[note]
23     # play the note for the beat_duration
24     spkr.pitch(freq)
25     sleep(beat_duration)
26
27     # pause for articulation
28     spkr.off()
29     sleep(0.05)
```

Objective 4 **- File System**

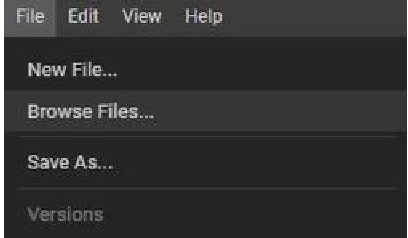
Exploring the File System

Your **jukebox** is going to need *lots* of **songs**.

- Right now, you just have one song that's *baked-into* the Python code!
- **Files** let you store data *outside* of your code.
- You have probably *downloaded* different kinds of *files* before.
    - Music, documents, videos,...

The ***CodeSpace*** File System is accessed by opening the **File** menu above the code editor.

Once you are in the **File** menu, select **Browse Files...** to see a list of all your files.

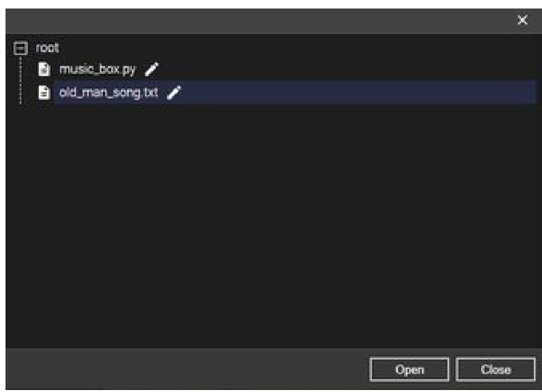


Song Files!

I have added a new file for you: **old_man_song.txt**

Open it up and take a look inside!!

- Select **old_man_song.txt** and click the **Open** button.



But what's in the file?

- It looks like *notes* separated by *spaces...*
- ***Hey, it's another song!!***

Goal:

- Open the **old_man_song.txt** using the **File** menu above the code editor.

Tools Found: File System

Solution:

```
1  # No Code
```

Objective 5 **- This Old Man**

Reading the Song File

Open the File for Reading

Your Python code can *read* from and *write* to files. But first you need to create a **file object** using the `open()` function from Python's 🔧file operations.

- `open` takes two parameters `open(file, mode)`
  - `file` is the filename
  - `mode` is explained below

🔧File Modes

- Files can be opened in different *modes*
- The `open()` function defaults to *read-only* if no mode is given.

- *read-only* mode is the same as `mode = "r"`
    - So the following two lines are equivalent:
        - `f = open("filename")`
        - `f = open("filename", "r")`

When you are done with a file **ALWAYS** close it using the `close()` function.

```
f.close()
```

This will make sure any changes your program made to the file are saved, and will allow other programs on your computer to access the file.

Now open up **old_man_song.txt**, `read()` the song, and ***play** it!

- See file operations and the **CodeTrek** for help on how to read a string from the file.

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
11     "F": 1397,
12     "G": 1568,
13     "A": 1760
14 }
15
16 # There is a file called old_man_song.txt in your system
17 f = # TODO: Open the old_man_song.txt for reading
```

You open a file for **reading** with:

- `f = open("filename", "r")`

```
18
19 # Read all the contents of the file
20 text = # TODO: Read the string from the file
```

To read **all** the contents of a file use:
`contents = f.read()`

```
21
22 f.close()
```

Always `close()` your file when you are done using it!

```
23
24 notes = text.split()
25
26 # Loop through each note in notes
27 for note in notes:
28     # Lookup the frequency of this note
29     f = freqs[note]
30
31     # Play the note for the beat_duration
32     spkr.pitch(f)
33     sleep(beat_duration)
34
35     # Pause for articulation
36     spkr.off()
37     sleep(0.05)
```

Goals:

- Read the data from a file named **old_man_song.txt**.
- Play the notes from the **old_man_song.txt**.

  Be sure to turn the `spkr.off()` between notes!!

Tools Found: Files, str

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
11     "F": 1397,
12     "G": 1568,
13     "A": 1760
14 }
15
16 # there is a file called old_man_song.txt in your system
17 f = open("old_man_song.txt", "r")
18
19 # read all the contents of the file
20 text = f.read()
21
22 f.close()
23
24 notes = text.split()
25
26 # loop through each note in notes
27 for note in notes:
28     freq = freqs[note]
29     # play the note for the beat_duration
30     spkr.pitch(freq)
31     sleep(beat_duration)
32
33     # pause for articulation
34     spkr.off()
35     sleep(0.05)
```

Objective 6 - Frere Jacques

Writing to a File!

You can `open()` *files* in different modes.

- Check out the file operations tool for a complete list.
- With the *"write"* modes you can also create *new* files.

Once you have an open *file object* you can write a string to it like this:

```
# Create a new file and put a special message in it.
f = open("my_file.txt", "w")
f.write("Hi, welcome to my file!")
f.close()
```

IMPORTANT NOTE: After you write to a file you must at some point **flush** it to guarantee that any *"buffered"* data is saved to the filesystem.

There are two ways to **flush** a file:

1. Close the file: `f.close()`, *or*
2. Flush the file: `f.flush()` *(when you need to keep it open for more writing...)*

This Objective's Goals

You will be *creating* a file, *writing* a song to it, *reading* the data back from the file, and finally *playing* the song on the **speaker**. *Whew!*

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
11     "F": 1397,
12     "G": 1568,
13     "A": 1760
14 }
15
16 data = "C D E C C D E C E F G E F G"
17
18 # Open a new file in write mode
19 # TODO: Open my_song.txt for writing
```

Open a file as **write only** with `f = open("filename", "w")`

```
20 # TODO: Write the data to my_song.txt
```

Write a string to the file `f.write("my string")`

```
21 f.close()
```

You **MUST** call `f.close()` or `f.flush()` to push the file contents to the file system.

It is always good practice to close your files!!!

```
22
23 # Open your file in read mode
24 f = open("my_song.txt", "r")
25 text = f.read()
26 f.close()
27
28 notes = text.split()
29
30 # Loop through each note in notes
31 for note in notes:
32     # Lookup the frequency of this note
33     freq = freqs[note]
34
35     # Play the note for the beat_duration
36     spkr.pitch(freq)
37     sleep(beat_duration)
38
39     # Pause for articulation
40     spkr.off()
41     sleep(0.05)
```

Hint:

- If your file is not writing:
  - Make sure you call `f.flush()` or `f.close()`

- This pushes the data to the file system!!

Goals:

- Write this string to a new file named **my_song.txt**: `data = "C D E C C D E C E F G E F G"`
- Open **my_song.txt** again and read the data back out.
- Play the notes from **my_song.txt**.

  Be sure to turn the `spkr.off()` between notes!!

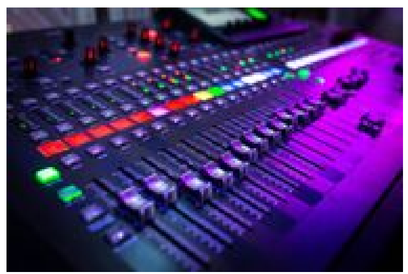
Tools Found: Files, str

Solution:

```
from botcore import *
from time import sleep

tempo = 150 # beats per minute
beat_duration = 60 / tempo # seconds

freqs = {
    "C": 1047,
    "D": 1175,
    "E": 1319,
    "F": 1397,
    "G": 1568,
    "A": 1760
}

data = "C D E C C D E C E F G E F G"

# open a new file in write mode
f = open("my_song.txt", "w")
f.write(data)
f.close()

# open your file in read mode
f = open("my_song.txt", "r")
text = f.read()
f.close()

notes = text.split()

# loop through each note in notes
for note in notes:
    freq = freqs[note]
    # play the note for the beat_duration
    spkr.pitch(freq)
    sleep(beat_duration)

    # pause for articulation
    spkr.off()
    sleep(0.05)
```

Objective 7 - Little Lamb

Add a Little *Rhythm!*



So far your tunes have been *melodic,* but the beat... face it, that's a little monotonous.

- It's time to spice up the timing.

Here are the notes for the song **Mary Had a Little Lamb**:

- G_6, F_6, E_6, F_6, G_6, G_6, G_6, F_6, F_6, F_6, G_6 A_6, A_6
- But some of those notes need to be played longer than others!

- How can you easily link a note with its duration?

Introducing... Multidimensional Lists!

Keeping your songs in the format `['C', 'D', 'E', 'C', 'C'...]` is an acceptable solution when you just have a sequence of notes.

- But now each note requires a *second* piece of data, a *duration!*

A multidimensional list, also known as a "matrix", is a perfect way to group related data. For example:

```
song = [[note, beats], [note, beats]...]
```

Check it out - lists inside a list!

A Lot to Unpack

You may already have used Python's unpacking feature to assign elements of a list or tuple to variable names in one go:

```
# Get the first note and beats in the song
note, beats = song[0]
```

But did you know you can do that as part of a `for loop`? See the loop tool for details on using a `target_list` in your `for loop`.

This could come in handy for playing your song:

```
for note, beat in song:
    # This is gonna rock...
```

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
11     "F": 1397,
12     "G": 1568,
13     "A": 1760
14 }
15
16 song = [
17     ["E", 1],
18     ["D", 1],
19     ["C", 1],
20     ["D", 1],
```

Not Just `notes` Anymore!

Your `song` is now a list of *lists!*

- Each element of the song is *itself* a list of `[note, beats]`

```
21     ["E", 1],
22     ["E", 1],
23     ["E", 2],
24     ["D", 1],
25     ["D", 1],
26     ["D", 2],
27     ["E", 1],
28     ["G", 1],
29     ["G", 2]
30 ]
31
32 # Loop through each [note,beats] element of song
33 for ??? in song:  # TODO: target list
34     # Lookup the frequency of this note
```

```
35      f = freqs[note]
```

Fix This

You need to use the `target_list` form of the `for` loop.

- Unpack each `song` element into `note, beats`.

```
36
37      # Play the note for the beat_duration * number of beats
38      spkr.pitch(f)
39      sleep(beats * beat_duration)
```

Sleep while the note is playing

Turn the speaker on, sleep for the length of the beats,
then turn the speaker off.

- Multiply `beats * beat_duration` to calculate how many seconds to `sleep()`.

```
40
41      # Pause for articulation
42      spkr.off()
43      sleep(0.05)
44
```

Goals:

- Unpack `note` and `beat` from your song using the *target list* in a `for` loop.
- Play the following notes in order:
  - G, F, E, F, G, G, G, F, F, F, G, A, A

Tools Found: list, Assignment, tuple, Loops, Iterable

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
11     "F": 1397,
12     "G": 1568,
13     "A": 1760
14 }
15
16 song = [
17     ["E", 1],
18     ["D", 1],
19     ["C", 1],
20     ["D", 1],
21     ["E", 1],
22     ["E", 1],
23     ["E", 2],
24     ["D", 1],
25     ["D", 1],
26     ["D", 2],
27     ["E", 1],
28     ["G", 1],
29     ["G", 2]
```

```
30  ]
31
32  # Loop through each [note,beats] element of song
33  for note, beats in song:
34      freq = freqs[note]
35      # play the note for the beat_duration * number of beats
36      spkr.pitch(freq)
37      sleep(beats * beat_duration)
38
39      # pause for articulation
40      spkr.off()
41      sleep(0.05)
42
```

***Objective 8* - Black Sheep**

An INTeresting problem...

You calculated your note duration by multiplying

`beat_duration` X `beats`

both of which are integers.

What if `beats` was a string ?

Okay, bear with me here. In the next Objective you are going to be reading all this from a **file**, and it's gonna come in as strings all the way. So, might as well deal with it now, right?

Introducing the int() function!

`int(x)` takes a value `x` and converts it to an integer!

The value can be a:

- string
- float
- ... even a bool

A Melody from the *String Section*

For this objective you'll be coding another song, using *only* strings!

Here's the song... *Feel free to **copy and paste** it into your code!*

```
black_sheep = [
    ["C", "2"],
    ["C", "2"],
    ["G", "2"],
    ["G", "2"],
    ["A", "1"],
    ["A", "1"],
    ["A", "1"],
    ["A", "1"],
    ["G", "4"],
    ["F", "2"],
    ["F", "2"],
    ["E", "2"],
    ["E", "2"],
    ["D", "2"],
    ["D", "2"],
    ["C", "4"]
]
```

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
```

```
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
11     "F": 1397,
12     "G": 1568,
13     "A": 1760
14 }
15
16 black_sheep = [
17     ["C", "2"],
18     ["C", "2"],
```

Stringified Beats!

This is a new song... can you guess the song from its name?

- Notice how the *beats* are now strings.

```
19     ["G", "2"],
20     ["G", "2"],
21     ["A", "1"],
22     ["A", "1"],
23     ["A", "1"],
24     ["A", "1"],
25     ["G", "4"],
26     ["F", "2"],
27     ["F", "2"],
28     ["E", "2"],
29     ["E", "2"],
30     ["D", "2"],
31     ["D", "2"],
32     ["C", "4"]
33 ]
34
35 def play_song(song):
36     # Loop through each [note, str_beats] in the song
37     for note, str_beats in song:
```

Another *Funky* Function

A bit of refactoring here. Notice this function's parameter is the *song* to play.

- Just use your keyboard or mouse to select the whole `for` loop and press *TAB* to indent it so it fits right inside the loop. *(see editor shortcuts for more)*

```
38         # Convert string beats to integer
39         beats = int(str_beats)
```

Integer Conversion

Convert that string to an integer so you can multiply it with `beat_duration` later.

```
40
41         # Lookup the frequency of this note
42         f = freqs[note]
43
44         # Play the note for the beat_duration
45         spkr.pitch(f)
46         sleep(beats * beat_duration)
47
48         # Pause for articulation
49         spkr.off()
50         sleep(0.05)
51
52 play_song(black_sheep)
```

Don't forget to ***call*** your new function, so the song actually plays!

Goals:

- Create an 🔧integer variable named `beats` from a 🔧string using the `int` function.
  - You will need to change your 🔧unpacking variable name to something different from *"beats"*.
  - Then do the `int()` conversion inside the `for` 🔧loop.
- Create a 🔧function named `play_song(song)` and move your whole `for` loop inside it.
  - Use the 🔧Editor Shortcuts to select the entire loop and 🔧indent it beneath the function `def`.
  - ***Call*** your new function at the *bottom* of your program, passing it the `song` as an 🔧argument.
- Play the following notes in order:
  - C, C, G, G, A, A, A, A, G, F, F, E, E, D, D, C

  With the following beats:
  - 2, 2, 2, 2, 1, 1, 1, 1, 4, 2, 2, 2, 2, 2, 2, 4

Tools Found: int, str, float, bool, Assignment, Loops, Functions, Editor Shortcuts, Indentation, Keyword and Positional Arguments, Refactoring, Parameters, Arguments, and Returns

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
11     "F": 1397,
12     "G": 1568,
13     "A": 1760
14 }
15
16 black_sheep = [
17     ["C", "2"],
18     ["C", "2"],
19     ["G", "2"],
20     ["G", "2"],
21     ["A", "1"],
22     ["A", "1"],
23     ["A", "1"],
24     ["A", "1"],
25     ["G", "4"],
26     ["F", "2"],
27     ["F", "2"],
28     ["E", "2"],
29     ["E", "2"],
30     ["D", "2"],
31     ["D", "2"],
32     ["C", "4"]
33 ]
34
35 def play_song(song):
36     # Loop through each note in notes
37     for note, str_beats in song:
38         # Convert string beats to integer
39         beats = int(str_beats)
```

```
40
41          # Lookup the frequency of this note
42          f = freqs[note]
43
44          # Play the note for the beat_duration
45          spkr.pitch(f)
46          sleep(beats * beat_duration)
47
48          # Pause for articulation
49          spkr.off()
50          sleep(0.05)
51
52  play_song(black_sheep)
```

Objective 9 - Rain, Rain

Another Song File!

I've placed **rain_rain_song.csv** in your filesystem!

It's saved in the format:

```
G,2
E,2
G,1
```

On each line, the note and number of beats are separated by a comma `','`.

- You will need to do some string operations to convert this into a multidimensional list!

Introducing `readlines()`

If you take a look at the file operations tool, you'll see the familiar `f.read()` function which returns a string. But *also* there is `f.readlines()`.

- This returns a list of *strings*, one for each line in the file!

`f.readlines()` would turn the above *file string* into this:

```
["G,2", "E,2", "G,1"]
```

Make Mine *Multidimensional*

The list above is *close*, but not exactly what you need for your `play_song(song)` function.

- You need a "list of lists", not a "list of strings".
- How to convert `"G,2"` into `["G", "2"]` ?

Check out the `split()` function of strings

Example:

```
line = "G,2"  # Line from the file
note_beat = line.split(",")  # Separate by comma
print(note_beat) # Prints: ["G", "2"]
```

You've got all the tools you need, time to code!

CodeTrek:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
```

```
11      "F": 1397,
12      "G": 1568,
13      "A": 1760
14  }
15
16  # Open the file and read in the lines
17  f = open("rain_rain_song.csv", "r")
18  file_lines = f.readlines()
19  f.close()
```

Open the File

Just like before...

- But this time use `readlines()` to get a list of strings, one for each line in the file.

```
20
21  # Build a multidimensional list from the file_lines
22  song = []
23  for line in file_lines:
24      # Make a list [note, beat] from each line in file
25      note_beat = line.split(",")
26      song.append(note_beat)
```

Building the Matrix

Start with an empty list to hold your `song`. Then loop over each **line** of the file and:

1. Split the comma-separated **line** string into a list: `[note, str_beat]`
2. Append this new list to the `song`.

```
27
28  def play_song(song):
29      # Loop through each [note, str_beats] in song
30      for note, str_beats in song:
31          # Convert string beats to integer
32          beats = int(str_beats)
33
34          # Lookup the frequency of this note
35          f = freqs[note]
36
37          # Play the note for the beat_duration
38          spkr.pitch(f)
39          sleep(beats * beat_duration)
40
41          # Pause for articulation
42          spkr.off()
43          sleep(0.05)
44
45  play_song(song)
```

Second Verse, *Same as the First!*

The rest of your code is the same.

Nice!

Goals:

- Read the data from rain_rain_song.csv with `readlines()`.
- Split each line from the file using the `split(",")` function.
- Play the notes and beats from the file rain_rain_song.csv.

Tools Found: str, list, Files, Loops

Solution:

```
from botcore import *
from time import sleep

tempo = 150 # beats per minute
beat_duration = 60 / tempo # seconds

freqs = {
    "C": 1047,
    "D": 1175,
    "E": 1319,
    "F": 1397,
    "G": 1568,
    "A": 1760
}

# Open the file and read in the lines
f = open("rain_rain_song.csv", "r")
file_lines = f.readlines()
f.close()

# Build a multidimensional list from the file_lines
song = []
for line in file_lines:
    # Make a list [note, beat] from each line in file
    note_beat = line.split(",")
    song.append(note_beat)

def play_song(song):
    # Loop through each [note, str_beats] in song
    for note, str_beats in song:
        # Convert string beats to integer
        beats = int(str_beats)

        # Lookup the frequency of this note
        f = freqs[note]

        # Play the note for the beat_duration
        spkr.pitch(f)
        sleep(beats * beat_duration)

        # Pause for articulation
        spkr.off()
        sleep(0.05)

play_song(song)
```

Objective 10 **- Jukebox**

Bringing it all together

I've placed 3 more files in your filesystem!

```
song_files = [
    "jingle_bells_song.csv",
    "twinkle_twinkle_song.csv",
    "rain_rain_song.csv",
    "black_sheep_song.csv"
]
```

You've already coded a function to play each song individually.

- Now it's time to make a ***jukebox!***

Jukebox Operation

Play one of the four songs when button **BTN-0** on CodeBot is pressed.

- Next time it's pressed, play the *next song* in the list.

That's it. Get coding!

CodeTrek:

```
from botcore import *
from time import sleep

tempo = 150 # beats per minute
beat_duration = 60 / tempo # seconds

freqs = {
    "C": 1047,
    "D": 1175,
    "E": 1319,
    "F": 1397,
    "G": 1568,
    "A": 1760
}

song_files = [
    "jingle_bells_song.csv",
    "twinkle_twinkle_song.csv",
    "rain_rain_song.csv",
    "black_sheep_song.csv"
]
```

The Song List

Copy this from the Objective instructions.

- These files have been added to your *filesystem*

```

def decode_song_file(name):
    # Open the file and read in the lines
    f = open(name, "r")
    file_lines = f.readlines()
```

Package Your Code

You've already written code to *decode a song file.* You just need to move it inside a function.

Make a function that:

- Takes a *file name* as a parameter.
- Returns the **song** as a *multidimensional list.*

```
    f.close()

    # Build a multidimensional list from the file_lines
    song = []
```

```
31      for line in file_lines:
32          # Make a list [note, beat] from each line in file
33          note_beat = line.split(",")
34          song.append(note_beat)
35
36      return song
37
38  # Build a list of songs decoded from files
39  song_list = []
40  for filename in song_files:
41      s = decode_song_file(filename)
42      song_list.append(s)
```

Song Catalog

Iterate through the `song_files` and use your *decode* function to convert them all to *songs* you can play.

- When this loop finishes you will have decoded all the files into *songs* and collected them in `song_list`.

```
43
44  def play_song(song):
45      # Loop through each [note, str_beats] in song
46      for note, str_beats in song:
47          # Convert string beats to integer
48          beats = int(str_beats)
49
50          # Lookup the frequency of this note
51          f = freqs[note]
52
53          # Play the note for the beat_duration
54          spkr.pitch(f)
55          sleep(beats * beat_duration)
56
57          # Pause for articulation
58          spkr.off()
59          sleep(0.05)
60
61
62  # Start with first song in list
63  i_song = 0
64
65  while True:
66      if buttons.was_pressed(0):
67          # Play the next song!
68          selected_song = song_list[i_song]
69          play_song(selected_song)
```

Main Loop - *Musical Buttons!*

Until you press a button this loop just spins forever checking if one *was pressed*.

- Press BTN0 to play the next song.
- Notice `i_song` starts with *index 0*, the first song in the list.
    - Coders often use 'i' in a variable name to mean *"index"*.

```
70
71          # Advance to next song index
72          i_song = i_song + 1
73
74          # Wrap around at end of list
75          if i_song == len(song_list):
76              i_song = 0
```

Classic wrap-around code.

- Just like *PacMan!*

```
77
```

Goals:

- Play the next song when button BTN-0 was pressed.
- Listen to `'jingle_bells_song.csv'`
- Listen to `'twinkle_twinkle_song.csv'`
- Listen to `'rain_rain_song.csv'`
- Listen to `'black_sheep_song.csv'`

Tools Found: Functions, Buttons, Parameters, Arguments, and Returns, Iterable, list, Variables

Solution:

```
1  from botcore import *
2  from time import sleep
3
4  tempo = 150 # beats per minute
5  beat_duration = 60 / tempo # seconds
6
7  freqs = {
8      "C": 1047,
9      "D": 1175,
10     "E": 1319,
11     "F": 1397,
12     "G": 1568,
13     "A": 1760
14 }
15
16 song_files = [
17     "jingle_bells_song.csv",
18     "twinkle_twinkle_song.csv",
19     "rain_rain_song.csv",
20     "black_sheep_song.csv"
21 ]  #@1
22
23 def decode_song_file(name):
24     # Open the file and read in the lines
25     f = open(name, "r")
26     file_lines = f.readlines()  #@2
27     f.close()
28
29     # Build a multidimensional list from the file_lines
30     song = []
31     for line in file_lines:
32         # Make a list [note, beat] from each line in file
33         note_beat = line.split(",")
34         song.append(note_beat)
35
36     return song
37
38 # Build a list of songs decoded from files
39 song_list = []
40 for filename in song_files:
41     s = decode_song_file(filename)
42     song_list.append(s)  #@3
43
44 def play_song(song):
45     # Loop through each [note, str_beats] in song
46     for note, str_beats in song:
47         # Convert string beats to integer  #@4
48         beats = int(str_beats)
49
50         # Lookup the frequency of this note
51         f = freqs[note]
52
53         # Play the note for the beat_duration
54         spkr.pitch(f)
55         sleep(beats * beat_duration)
56
```

```
57          # Pause for articulation
58          spkr.off()
59          sleep(0.05)
60
61
62  # Start with first song in list
63  i_song = 0  #@5
64
65  while True:
66      if buttons.was_pressed(0):
67          # Play the next song!
68          selected_song = song_list[i_song]
69          play_song(selected_song)  #@6
70
71          # Advance to next song index
72          i_song = i_song + 1
73
74          # Wrap around at end of list
75          if i_song == len(song_list):
76              i_song = 0  #@7
77
```

***Mission 15* - Cyber Storm**

Help protect an email server by using file operations!

***Objective 1* - .eml**

If you receive an email about tinned meat, don't open it!

It's spam...

I've placed an email in your filesystem named **'Antivirus.eml'**.

- **'.eml'** is a standard file extension for emails.

Email files can be read as though they are plain text.

- The file contains two primary sections (header and body) and many different parts.

What are the different parts of an email?

We all know how emails work, but coding often requires you to *explicitly* define the parts of a system.

Humor me in the name of security!

An email consists of:

- "From" - the sender
- "To" - the recipient
- "Date" - the date the email was sent
- "Subject" - the brief or title
- "Body" - the content

Time to print the contents of the 'Antivirus.eml' using 🔧File Operations!

Create a new file!

- Use the File → New File menu to create a new file called "email_scan.py"

CodeTrek:

```
1  email_file = 'Antivirus.eml'
2
```

```
3  f = # TODO: Open the email in read mode!
```

Use the `open(filename, mode)` function!

- Filename is a string of the file's name
- For reading use the `"r"` mode

The `open` function returns a file object.

- `f = open("Antivirus.eml", "r")`

```
4  file_contents = f.read()
```

Once you have an open file you can read its contents.

- `f.read()` will read the entire file at once.

```
5  print(file_contents)
6  f.close()
```

Always close your file after you are done using it.

- This will free up resources and *flush* the file.
  - Your computer will thank you!

Goals:

- Open the `"Antivirus.eml"` file in **read** mode.
- Print the entire contents of `"Antivirus.eml"` to the console.

Tools Found: Files, Print Function, str

Solution:

```
1  email_file = 'Antivirus.eml'
2
3  f = open(email_file, "r")
4  file_contents = f.read()
5  print(file_contents)
6  f.close()
```

Objective 2 - With or Without You

The 'with' Statement!

In the first objective you opened a file like this:

```
f = open(file_name, "r")
file_contents = f.read()
f.close()
```

`f.close()` is called because it releases resources back to the computer.

- That seems like it would be easy to forget!!

This is where the `with` statement comes in.

- It will close the file for you automatically!

This is awesome for many reasons, two of which being:

1. You don't have to call `f.close()`.
2. The resources get freed up even if there is an error in your code.

Here's what the above code would look like utilizing `with`!

```
with open(file_name, "r") as f:
    file_contents = f.read()
```

Easy as that!

Also, this time why don't you use the `readlines()` function instead of `read()`?

`readlines()` reads the entire file but...

- It returns a list of strings instead of just a single string.
- This will be useful for breaking down the email.

CodeTrek:

```
1  email_file = "Antivirus.eml"
2
3  # TODO: Open the email_file using the 'with' statement!
```

Remember, you can use `with open(email_file, "r") as f:` instead of `f = open(email_file, "r")`!

```
4      file_contents = # TODO: Read the file contents using readlines()
```

The file object can be accessed inside the `with` block.

- This time use `f.readlines()` instead of `f.read()`

```
5      print(file_contents)
```

Goals:

- Open the `"Antivirus.eml"` email file using the `with` statement.
- Use the `readlines()` function to read the entire file into a list.
- Print the contents of the email file, in the format returned from `readlines()`, to the console.

Tools Found: Exception, list, str, Print Function

Solution:

```
1  email_file = "Antivirus.eml"
2
3  with  open (email_file, "r") as f:
4      file_contents = f.readlines()
5      print(file_contents)
6
```

Objective 3 **- Newline**

Line by line

When you've got a small file it's fine to read all of the content at once.

- If you're working with a large file, it may be more convenient to read the data out line by line!

Fortunately, `open()` allows for this because it returns a file as an iterable.

- This let's you step through each line in the file using a for loop.

```
for line in f:
    # do something with line
```

Check out File Operations for an example!

Let's talk escape sequences!

In the previous objective, the "From" line looked like this:

```
'From: Anti Virus <antivirus@firialabs.com>\r\n'
```

What are those *strange* things, `'\r'` and `'\n'`?

- They are called escape sequences!!
- Each one of those (backslash included) inserts a single "special character".

Here are a few you are likely to run across:

Char	Name	Description
`'\n'`	New Line	Unsuprisingly, it creates a new line in a text file!
`'\r'`	Carriage Return	Set the cursor to the beginning of the line
`'\t'`	Tab	A tab that shows spaces as set by your text editor

Since you're working on isolating components of an email, these line endings need to be removed.

Introducing strip()!

`'string'.strip(chars)` is a function that removes characters from the **beginning** and **end** of a string!

- If `chars` is not given, the function just removes whitespace.

Whitespaces are more than just spaces. They include:

- Newlines `'\n'`
- Carriage Returns `'\r'`
- Tabs `'\t'`
- Spaces `' '`

CodeTrek:

```
1  email_file = 'Antivirus.eml'
2
3  with open(email_file) as f:
4      email = ""
```

An empty string at the start.

- You will be *appending* the contents of each line to this string.
- But first be sure to `strip()` the whitespace!

```
5      # TODO: Iterate over file with a for loop
```

You can treat `f` just like a list here!

```
for line in f:
    # Do something with the line!
```

```
6          email = email + line.strip()
```

`line.strip()` will remove **all** whitespace characters from the beginning and end of a file.

Append the line to the `email` string after you have stripped it!

- Remember the `+` operator means *append* in string-land.
- Also known as *concatenation!*

```
7      print(email)
```

Print the cleaned-up email contents.

- Each line in the email should have been stripped of whitespace.

Goals:

- Iterate over the email file to read its lines one by one using a `for` loop.
- 1. `strip()` the whitespace from each line in the file.
  2. Combine all the stripped lines into a single string variable.
  3. Print the variable to the console.

Tools Found: Iterable, Loops, Files, Escape Sequences, str, Print Function, list

Solution:

```
1  email_file = 'Antivirus.eml'
2
3  with open(email_file) as f:
4      email = ''
5      for line in f:
6          email = email + line.strip()
7      print(email)
```

***Objective 4* - Email Isolate**

Where does the chicken check his email?

His inboks...

Wouldn't it be cool if you could access the email's date by typing `email['date']`?

- Apply some previous concepts and organize the email in a dictionary!

First, you'll need to somehow *isolate* the 'Date' line from the other lines.

Introducing *startswith!*

Prepare to be shocked...

- `s.startswith(prefix)` returns `True` if the string `s` starts with `prefix`!

Since a stripped 'Date' looks like `'Date: Fri, 14 Aug 1987 09:10:17 -0800'` you can identify it using `startswith('Date: ')` like so:

```
is_date_line = 'Date: Fri, 14 Aug 1987 09:10:17 -0800'.startswith('Date: ')
print(is_date_line) # True
```

Ah Ha!

After identifying the line...

Theres one last bit of formatting before adding the actual **date** to a dictionary.

- You'll need to get rid of the `'Date: '` part of the string! A good way to do that is "slicing".
- A string can be sliced using the notation `s[start:stop]`.
  - Just like ranges, string slicing *begins* at the `start` and *ends 1 character before* the `stop`.
  - It will just return the rest of the string if `stop` is missing.

CodeTrek:

```
1  email_file = 'Crypto.eml'
2
3  with open(email_file) as f:
4      email = {}
```

Start with a new empty dictionary.

```
5       for line in f:
6           clean_line = line.strip()
```

Strip whitespace from the **beginning** and **end** of the line.

```
7           if line.startswith("Date: "):
```

Identify the 'Date' line using `startswith`.

```
8               email['date'] = clean_line[6:]
```

`clean_line` looks like `'Date: Sat, 03 Jan 2009 19:06:00 -0100'`.

You want the **value** to be the *contents* of the **Date** line.

- Gotta remove the **"Date: "** prefix.
- `clean_line[6:]` returns a string with the first 6 characters removed!

```
9           elif # TODO: Catch the 'From' line with startswith
```

How can you tell whether a line starts with "From: "?

That's right!

`line.startswith("From: ")`!

```
10              email['from'] = # TODO: slice the clean_line to isolate the data
```

Now you need to cut `"From: "` out of the string!

`"From: "` is 6 characters long!

You can remove 6 characters from the start of a string like this:

- `email['from'] = clean_line[6:]`

```
11          elif line.startswith("To: "):
12              email['to'] = clean_line[4:]
13          elif line.startswith("Subject: "):
14              email['subject'] = clean_line[9:]
15      print(email)
```

Hint:

- Your dictionary should look like this:

```
email = {
  "subject": "Crypto",
  "to": "Codee <codee@firialabs.com>",
  "from": "Crypto Telemarketer <cryptomarketing@firialabs.com>",
  "date": "Sat, 03 Jan 2009 19:06:00 -0100"
}
```

Goals:

- Add a `'from'` **key** to a dict named `email`.

```
email['from'] = "contents of the FROM line..."
```

  - The **value** should be the 'From' line in `'Crypto.eml'`.

  - Strip off whitespace line endings and `'From: '` before assigning the value.

- 1. Add the `'date'`, `'to'` and `'subject'` **key:value** pairs to the dict.
  2. Print the entire dict to the console.

Tools Found: dictionary, str, Ranges

Solution:

```
1  email_file = 'Crypto.eml'
2
3  with open(email_file) as f:
4      email = {}
5      for line in f:
6          clean_line = line.strip()
7          if line.startswith("Date: "):
8              print(clean_line)
9              email['date'] = clean_line[6:]
10         elif line.startswith("From: "):
11             email['from'] = clean_line[6:]
12         elif line.startswith("To: "):
13             email['to'] = clean_line[4:]
14         elif line.startswith("Subject: "):
15             email['subject'] = clean_line[9:]
16     print(email)
```

Quiz 1 - More File Ops

Question 1: What is `x` in the code below?

```
with open(my_file, 'r') as f:
    for x in f:
        print(x)
```

✓ A Line in the File

✗ A Character in the File

✗ Every Number in the File

Question 2: What is this character in Python `'\n'`?

✓ New Line

✗ Tab

✗ Carriage Return

✗ Backspace

Question 3: Why would you use `with` to open a file?

✓ It will close it for you.

✗ It is opened with super speed.

✗ It merges it with a second file.

Question 4: What is this character in Python `'\t'`?

✓ Tab

✗ New Line

❌ Carriage Return

❌ Backspace

***Objective 5* - Body Isolate**

Something's missing...

You still need the body of the email.

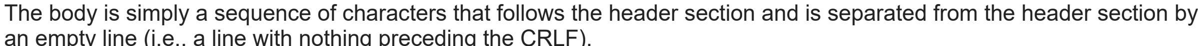
The body is unique from the other components:

- It can be multiple lines!
- It doesn't start with a `'Body: '` prefix.

Oh no!! That breaks your system!

So, how can you find it?

The "Internet Message Format", which was standardized by RFC 5322 (look it up if you'd like!) says this:



> The body is simply a sequence of characters that follows the header section and is separated from the header section by an empty line (i.e., a line with nothing preceding the CRLF).

Simply put, the email will have a line that **only** contains `'\r\n'` (aka **CRLF**).

- Everything after that line is the **body**!

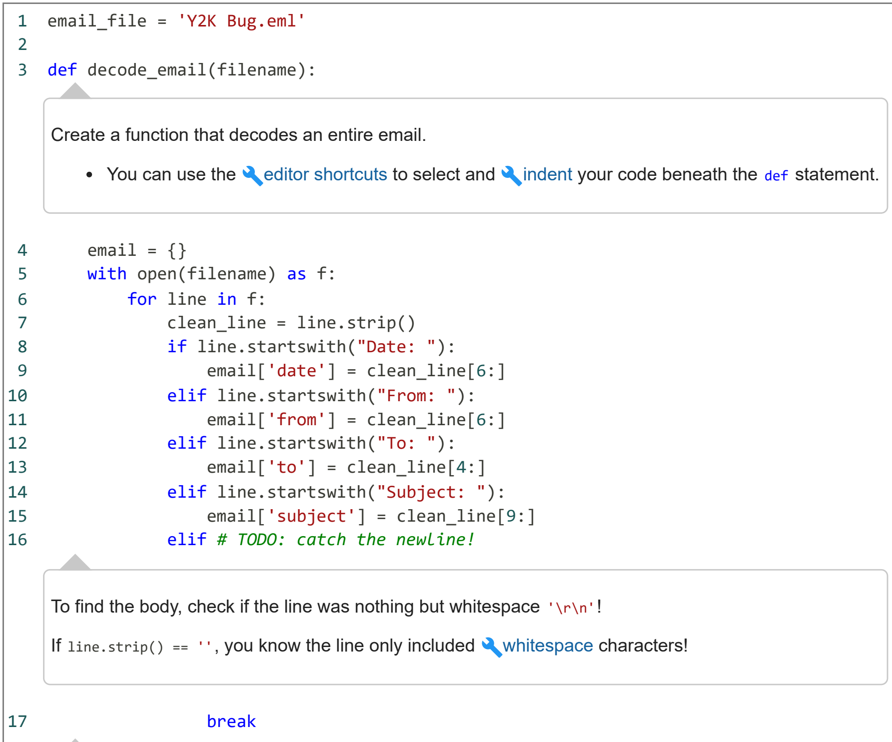
Isolating the body is easier than it first appears!

- The file object "keeps track" of which lines you've already read as you 🔧iterate.

If you call `read()` after you've iterated over a few lines, you'll get the rest of the file!!

Try to put it all together!

CodeTrek:



```
1  email_file = 'Y2K Bug.eml'
2
3  def decode_email(filename):
```

Create a function that decodes an entire email.

- You can use the 🔧editor shortcuts to select and 🔧indent your code beneath the `def` statement.

```
4      email = {}
5      with open(filename) as f:
6          for line in f:
7              clean_line = line.strip()
8              if line.startswith("Date: "):
9                  email['date'] = clean_line[6:]
10             elif line.startswith("From: "):
11                 email['from'] = clean_line[6:]
12             elif line.startswith("To: "):
13                 email['to'] = clean_line[4:]
14             elif line.startswith("Subject: "):
15                 email['subject'] = clean_line[9:]
16             elif # TODO: catch the newline!
```

To find the body, check if the line was nothing but whitespace `'\r\n'`!

If `line.strip() == ''`, you know the line only included 🔧whitespace characters!

```
17                 break
```

If the line is just whitespace, then `break` the for loop so you can read the remaining lines as the body!

```
18
19          email['body'] = # TODO: read the body from the file!
```

`f.read()` will give you the rest of the email!

- You've already iterated through the date, from, to, and subject in the header.

Simply assign the rest to the `'body'` key like `email['body'] = f.read()`!

```
20      return email
21
22  eml = decode_email(email_file)
23
24  print(eml)
```

Hint:

- Make sure you don't include the **empty** line after the header in your `email 'body'`.

Goal:

- 1. Assign a `'body'` key:value pair with the value of the `"Y2K Bug.eml"` body to the `email` dict.
  2. Print the entire dictionary to the console.

Tools Found: Iterable, dictionary, Editor Shortcuts, Indentation, undefined, Loops

Solution:

```
1   email_file = 'Y2K Bug.eml'
2
3   def decode_email(filename):
4       email = {}
5       with open(filename) as f:
6           for line in f:
7               clean_line = line.strip()
8               if line.startswith("Date: "):
9                   email['date'] = clean_line[6:]
10              elif line.startswith("From: "):
11                  email['from'] = clean_line[6:]
12              elif line.startswith("To: "):
13                  email['to'] = clean_line[4:]
14              elif line.startswith("Subject: "):
15                  email['subject'] = clean_line[9:]
16              elif line.strip() == '': # explain email spec for this
17                  break
18          email['body'] = f.read() # teach that reading continues from where it left off last
19      return email
20
21  eml = decode_email(email_file)
22
23  print(eml)
24
```

Objective 6 - Are You In or Not?

Word Slayer

You've got the email translated to a dict.

- Time to work on the security!

You'll need to write a function that can identify undesirable language and replace it with a notice of removal.

If only we could search a string for specific words...

Introducing the *in* and *not in* keywords!

The `in` keyword has two purposes:

1. Iterating through a `for` loop.
2. Checking if a value exists in a sequence.

You can use `in` to check if a word is **in** a string!!

WAIT FOR IT...

- `not in` is the opposite of `in`!

But how do you *replace* a string?

Introducing *replace()*

`s.replace(old, new)` replaces `old` with `new` in string `s`.

Pretty straightforward right?

CodeTrek:

```
1  email_file = 'Creeper Virus.eml'
2
3  def decode_email(filename):
4      email = {}
5      with open(filename) as f:
6          for line in f:
7              clean_line = line.strip()
8              if line.startswith("Date: "):
9                  email['date'] = clean_line[6:]
10             elif line.startswith("From: "):
11                 email['from'] = clean_line[6:]
12             elif line.startswith("To: "):
13                 email['to'] = clean_line[4:]
14             elif line.startswith("Subject: "):
15                 email['subject'] = clean_line[9:]
16             elif line.strip() == '':
17                 break
18         email['body'] = f.read()
19     return email
20
21 def scan_email(email):
```

Add a new function to scan the email dict for *viruses*!

```
22     if # TODO: if you found the string 'virus' in the body
```

Use the `in` keyword here!

To catch a `'substring'` in a `'string'`, simply check `if 'substring' in 'string'`.

- That would look like `if 'virus' in email['body']:`!

```
23         print("Found a virus. Removing it.")
24         email['body'] = # TODO: replace the word 'virus' with 'REMOVED' in the body!
```

Use `'string'.replace(old, new)` to swap the word `'virus'` with `'REMOVED'`

- In this case, your `'string'` is `email['body']`

```
25         return True
26     else:
27         print('No virus detected')
28         return False
29
```

```
30 eml = decode_email(email_file)
31 virus_found = scan_email(eml)
32 print(eml['body'])
```

🔧Print the new `'body'` to the console to witness your amazing security!

Goals:

- Replace the word `'virus'` with `'REMOVED'` in the `email['body']` using the `replace()` function!
    - You will be reading the `'Creeper Virus.eml'` file.
- 🔧Print the new email **body** to the console.

Tools Found: dictionary, Loops, str, Print Function, Functions

Solution:

```
1  email_file = 'Creeper Virus.eml'
2
3  def decode_email(filename):
4      email = {}
5      with open(filename) as f:
6          for line in f:
7              clean_line = line.strip()
8              if line.startswith("Date: "):
9                  email['date'] = clean_line[6:]
10             elif line.startswith("From: "):
11                 email['from'] = clean_line[6:]
12             elif line.startswith("To: "):
13                 email['to'] = clean_line[4:]
14             elif line.startswith("Subject: "):
15                 email['subject'] = clean_line[9:]
16             elif line.strip() == '':
17                 break
18         email['body'] = f.read()
19     return email
20
21 def scan_email(email):
22     if 'virus' in email['body']:
23         print("Found a virus. Removing it.")
24         email['body'] = email['body'].replace('virus', 'REMOVED')
25         return True
26     else:
27         print('No virus detected')
28         return False
29
30 eml = decode_email(email_file)
31 virus_found = scan_email(eml)
32 print(eml['body'])
```

Objective 7 - Blocklist

Flag the "Bad Actors"

After you've identified an email as containing a virus:

- Keep note of the sender's address so you can block their emails in the future!

A list of disallowed senders is called a **"blocklist"**!

You can create a file called `'blocklist.csv'` that will **persist** through objectives!

How do I create a file?

You will use the same `open(filename, mode)` function with a different **mode**!

There are two `mode`'s that will create a file if one doesn't exist.

1. `'w'` or 'write' overwrites if the file already exists and creates a new one!
2. `'a'` or 'append' writes to the end of a file if it exists, adding to the previous content!

Interested in more modes? Check out File Operations!

Notify the User!

Print a message the first time you create a *blocklist*.

- That means you need to check if one already exists...

  But how do I check if a file exists?

You ask extremely pertinent questions...

Introducing *os.path.exists()*

`os.path.exists(filepath)` returns whether `filepath` exists on the file system!

CodeTrek:

```
1  import os
```

Make sure to `import` the `os` module.

- This will let you use the `os.path.exists()` function.

```
2
3  email_file = 'Creeper Virus.eml'
4
5  def decode_email(filename):
6      email = {}
7      with open(filename) as f:
8          for line in f:
9              clean_line = line.strip()
10             if line.startswith("Date: "):
11                 email['date'] = clean_line[6:]
12             elif line.startswith("From: "):
13                 email['from'] = clean_line[6:]
14             elif line.startswith("To: "):
15                 email['to'] = clean_line[4:]
16             elif line.startswith("Subject: "):
17                 email['subject'] = clean_line[9:]
18             elif line.strip() == '':
19                 break
20         email['body'] = f.read()
21     return email
22
23 def scan_email(email):
24     if 'virus' in email['body']:
25         print("Found a virus. Removing it")
26         email['body'] = email['body'].replace('virus', 'REMOVED')
27         return True
28     else:
29         print('No virus detected')
30         return False
31
32 eml = decode_email(email_file)
33 virus_found = scan_email(eml)
34
35 # If a virus was found, add sender to the blocklist!
36 if virus_found:
37     # Alert user if this is the first time creating blocklist
38     # TODO: if not os.path.exists...
```

Check to see if the `'blocklist.csv'` file does **NOT** exist.

- `if not os.path.exists('blocklist.csv'):`

```
39          print("Creating blocklist!")
40
41      # TODO: Open 'blocklist.csv' in append mode!
```

Can you guess how to open the file in append `mode`?

- ***Yep!***

It's the same as before except the mode is `'a'`:

- `with open('blocklist.csv', 'a') as f:`

```
42          bl_entry = eml['from'] + ','
```

The format of a '.csv' or **Comma Separated Values** is data separated by commas.

- You can append a comma to the `bl_entry` string using the `+` operator!

```
bl_entry = eml['from'] + ','
```

This writes the email address `+` a `','`

```
43          f.write(bl_entry)
```

Now write to the **blocklist** file.

- `f.write(bl_entry)`

Goals:

- Print "Creating a blocklist" the first time you create `'blocklist.csv'`
  - Check if the `'blocklist.csv'` file exists with `os.path.exists()`.
- Add the "bad actor" `'from'` email address to the *blocklist*.
  - Append the dict `'from'` value followed by a comma `','` to `'blocklist.csv'` if a virus is found in `'Creeper Virus.eml'`.

Tools Found: Files, dictionary, str

Solution:

```
1  import os
2
3  email_file = 'Creeper Virus.eml'
4
5  def decode_email(filename):
6      email = {}
7      with open(filename) as f:
8          for line in f:
9              clean_line = line.strip()
10             if line.startswith("Date: "):
11                 email['date'] = clean_line[6:]
12             elif line.startswith("From: "):
13                 email['from'] = clean_line[6:]
14             elif line.startswith("To: "):
15                 email['to'] = clean_line[4:]
16             elif line.startswith("Subject: "):
17                 email['subject'] = clean_line[9:]
18             elif line.strip() == '':
19                 break
```

```
20          email['body'] = f.read()
21      return email
22
23  def scan_email(email):
24      if 'virus' in email['body']:
25          print("Found a virus. Removing it")
26          email['body'] = email['body'].replace('virus', 'REMOVED')
27          return True
28      else:
29          print('No virus detected')
30          return False
31
32  eml = decode_email(email_file)
33  virus_found = scan_email(eml)
34
35  if virus_found:
36      # Alert user if this is the first time creating blocklist
37      if not os.path.exists('blocklist.csv'):
38          print("Creating blocklist!")
39
40      with open('blocklist.csv', 'a') as f:
41          bl_entry = eml['from'] + ','
42          f.write(bl_entry)
```

Objective 8 **- Threat Removal**

Put that blocklist to use!

That running list of nefarious emailers you've developed is about to come in handy.

You can prevent a virus from even making it to the inbox by deleting any emails received from blocklisted addresses.

Introducing os.remove()!

If the sender's email is on the blocklist...

- Delete the file by calling `os.remove(filepath)`!

Reminder: The emails on the blocklist are separated by a `','`

- You can call `file_contents.split(',')` to get an array of addresses!
- See the 🔧string tool for more details on `split()`

CodeTrek:

```
1   import os
2
3   email_file = 'Creeper Virus.eml'
4
5   def decode_email(filename):
6       email = {}
7       with open(filename) as f:
8           for line in f:
9               clean_line = line.strip()
10              if line.startswith("Date: "):
11                  email['date'] = clean_line[6:]
12              elif line.startswith("From: "):
13                  email['from'] = clean_line[6:]
14              elif line.startswith("To: "):
15                  email['to'] = clean_line[4:]
16              elif line.startswith("Subject: "):
17                  email['subject'] = clean_line[9:]
18              elif line.strip() == '':
19                  break
20          email['body'] = f.read()
21      return email
22
23  def scan_email(email):
24      if 'virus' in email['body']:
25          email['body'] = email['body'].replace('virus', 'REMOVED')
```

```
26          return True
27      else:
28          return False
29
30  def spam_filter(sender, filename):
```

Create a new function for filtering out emails from bad actors!

```
31      with open('blocklist.csv', 'r') as f:
32          data = f.read()
33          blocklist = # TODO: separate the adresses using split!
```

Read in the whole `'blocklist.csv'` and then create a list of blocked senders.

The `'.csv'` separator is `','`. Use it in the `'string'.split()` function to create your list!

- `blocklist = data.split(',')`

```
34          if sender in blocklist:
```

Check if the email's `'from'` is in the list of blocked senders.

- Hey, you can use `in` on lists too!

```
35              # TODO: delete the file!
```

Use `os.remove(filename)` to delete a bad email.

- Be careful though! You'll get an error if the file doesn't exist!

```
36
37  eml = decode_email(email_file)
38  virus_found = scan_email(eml)
39
40  if virus_found:
41      # Alert user if this is the first time creating blocklist
42      if not os.path.exists('blocklist.csv'):
43          print("Creating blocklist!")
44
45      with open('blocklist.csv', 'a') as f:
46          bl_entry = eml['from'] + ','
47          f.write(bl_entry)
48
49  spam_filter(eml['from'], email_file)
```

Don't forget to call your new `spam_filter` function.

Goals:

- Read in the entire `'blocklist.csv'` file.
- Delete the `'Creeper Virus.eml'` from the file system if its `'from'` value is on the **blocklist**!

Tools Found: str, list

Solution:

```
1  import os
2
3  email_file = 'Creeper Virus.eml'
4
5  def decode_email(filename):
```

```
6      email = {}
7      with open(filename) as f:
8          for line in f:
9              clean_line = line.strip()
10             if line.startswith("Date: "):
11                 email['date'] = clean_line[6:]
12             elif line.startswith("From: "):
13                 email['from'] = clean_line[6:]
14             elif line.startswith("To: "):
15                 email['to'] = clean_line[4:]
16             elif line.startswith("Subject: "):
17                 email['subject'] = clean_line[9:]
18             elif line.strip() == '':
19                 break
20         email['body'] = f.read()
21     return email
22
23 def scan_email(email):
24     if 'virus' in email['body']:
25         email['body'] = email['body'].replace('virus', 'REMOVED')
26         return True
27     else:
28         return False
29
30 def spam_filter(sender, filename):
31     with open('blocklist.csv', 'r') as f:
32         data = f.read()
33         blocklist = data.split(',')
34         if sender in blocklist:
35             os.remove(filename) # teach delete file
36
37 eml = decode_email(email_file)
38 virus_found = scan_email(eml)
39
40 if virus_found:
41     # Alert user if this is the first time creating blocklist
42     if not os.path.exists('blocklist.csv'):
43         print("Creating blocklist!")
44
45     with open('blocklist.csv', 'a') as f:
46         bl_entry = eml['from'] + ','
47         f.write(bl_entry)
48
49 spam_filter(eml['from'], email_file)
```

Quiz 2 - File Modes

Question 1: Which of these returns `True`?

✓ `'us' in 'virus'`

✗ `'i' in 'team'`

✗ `'ate' in 'threat'`

Question 2: What does the mode `'a'` mean in `open(my_file, 'a')`?

✓ Append

✗ Write

✗ Read

✗ Exclusive Creation

Question 3: Which of these opens a file for Read only?

✓ `open(my_file, 'r')`

❌ `open(my_file, 'w')`

❌ `open(my_file, 'a')`

Question 4: What is the correct code to delete a file?

✅ `os.remove(my_file)`

❌ `delete(my_file)`

❌ `ERROR: Invalid Code Block!! f = open(my_file, 'r') f.remove()

ERROR: Invalid Code Block!!

❌ `os.path.delete(my_file)`

Question 5: What does `break` do in the following code?

```
for i in range(10):
    break
print('Done!')
```

✅ Exits the Loop

❌ Ends Your Program

❌ Breaks the Internet

Objective 9 **- Complete Scan**

Testing out the system!

Yahoo!! Your security system is finally ready to test!

I've supplied you with a brand new `'blocklist.csv'`.

- It contains the sender information of some bad actors!

I also dropped a few more *emails* on your filesystem

This includes all the emails we've come across so far, plus a couple extras:

(Click the copy *button and paste this into your code.)*

```
email_files = [
    'Creeper Virus.eml',
    'Antivirus.eml',
    'Y2K Bug.eml',
    'Crypto.eml',
    'Firework Celebration.eml'
]
```

If your security program works properly:

- All the viruses will be deleted
- The safe emails will remain!

Give it a shot!

CodeTrek:

```
1  import os
2
3  email_files = [
```

Your email file list

Paste this in from the Objective description.

```
4      'Creeper Virus.eml',
5      'Antivirus.eml',
6      'Y2K Bug.eml',
7      'Crypto.eml',
8      'Firework Celebration.eml'
9  ]
10
11 def decode_email(filename):
12     email = {}
13     with open(filename) as f:
14         for line in f:
15             clean_line = line.strip()
16             if line.startswith("Date: "):
17                 email['date'] = clean_line[6:]
18             elif line.startswith("From: "):
19                 email['from'] = clean_line[6:]
20             elif line.startswith("To: "):
21                 email['to'] = clean_line[4:]
22             elif line.startswith("Subject: "):
23                 email['subject'] = clean_line[9:]
24             elif line.strip() == '':
25                 break
26         email['body'] = f.read()
27     return email
28
29 def scan_email(email):
30     if 'virus' in email['body']:
31         email['body'] = email['body'].replace('virus', 'REMOVED')
32         return True
33     else:
34         return False
35
36 def spam_filter(sender, filename):
37     with open('blocklist.csv', 'r') as f:
```

This time I've supplied you with `'blocklist.csv'`.

- An accumulation of nefarious emailers from the previous objectives!

```
38         data = f.read()
39         blocklist = data.split(',')
40         if sender in blocklist:
41             os.remove(filename)
42
43 #----- New code -----
44
```

New code goes here

Delete the `if virus_found:` block of code.

- This program totally relies on the *blocklist*.
- Your `spam_filter()` and stuff will move inside a loop below.

Now you just need to loop across the `email_files` list...

```
45 # TODO: iterate over email_files!
```

Iterate over the list of `email_files` and send each through the decoder, scanner, and filter!

```
for email_file in email_files:
    # TODO
```

```
46     eml = decode_email(email_file)
```

```
47     virus_found = scan_email(eml)
48     spam_filter(eml['from'], email_file)
```

Goal:

- Delete all `email_files` sent from any address in `'blocklist.csv'`!
    - Be sure to keep the good ones!

Tools Found: Iterable

Solution:

```
1  import os
2  
3  email_files = [
4      'Creeper Virus.eml',
5      'Antivirus.eml',
6      'Y2K Bug.eml',
7      'Crypto.eml',
8      'Firework Celebration.eml'
9  ]
10 
11 def decode_email(filename):
12     email = {}
13     with open(filename) as f:
14         for line in f:
15             clean_line = line.strip()
16             if line.startswith("Date: "):
17                 email['date'] = clean_line[6:]
18             elif line.startswith("From: "):
19                 email['from'] = clean_line[6:]
20             elif line.startswith("To: "):
21                 email['to'] = clean_line[4:]
22             elif line.startswith("Subject: "):
23                 email['subject'] = clean_line[9:]
24             elif line.strip() == '':
25                 break
26         email['body'] = f.read()
27     return email
28 
29 def scan_email(email):
30     if 'virus' in email['body']:
31         email['body'] = email['body'].replace('virus', 'REMOVED')
32         return True
33     else:
34         return False
35 
36 def spam_filter(sender, filename):
37     with open('blocklist.csv', 'r') as f:
38         data = f.read()
39         blocklist = data.split(',')
40         if sender in blocklist:
41             os.remove(filename)
42 
43 for email_file in email_files:
44     eml = decode_email(email_file)
45     virus_found = scan_email(eml)
46     spam_filter(eml['from'], email_file)
```