Python with CodeX

Learn Python fundamentals through fun projects with the CodeX.

Mission 1 - Welcome

Welcome to the CodeSpace Development Environment!

A virtual world for exploring robotics with code.

We're glad you're here!

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

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

Ready to begin your first *Mission?*

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

Objective 1 - Mission Objectives

Objectives

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

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

The *goals* to complete the Objective are below:

Goal:

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

Solution:

N/A

Objective 2 - Text Editor

Text Editor

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

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

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!
  - Rotate *all the way* around!

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 1 Complete

Welcome to CodeSpace!

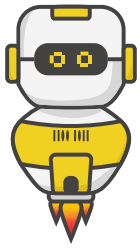
You've completed your first ***Mission.***

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

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

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

Mission 2 - Introducing CodeX



Greetings!

You are at the beginning of an exciting journey. I'll be your guide as you explore the immense world of *Python* with your **CodeX**.

Why learn coding?

- Hey, it's not just for robots anymore!
- Or laptops, mobile phones, and games,...
- Computer *chips* are making lots of things we use every day **smarter**

But... *Everything* computers do has to be coded by humans like YOU

As you complete this *project-based* course, you'll be learning skills that can be used to program *ANY* computer!

Objective 1 - Behold the CodeX

The computer you'll build projects with is called the CodeX.

The Firia Labs ***CodeX*** is a powerful *embedded* computer.

- It has loads of *sensors* and *buttons* for **input**.
- With an **LCD** 🔧**display**, amazing 🔧**Audio**, and tons of 🔧*LEDs* for output.

Even better, the CodeX can **connect** to the world around it.

- Those black connectors along the top edge are electronic terminals you can wire to sensors, motors, lights, and more!
- Plus there's an expansion interface to fit additional circuit-boards.

What projects can *you* imagine using the **CodeX** for?

- Control a light show
- Measure sound and light levels
- Operate a robot
- Generate music and sound effects
- Detect motion and activate an alarm
- Create handheld video games

Goal:

- Click at least one of the 🔧 tools above to learn more about the CodeX.

Tools Found: Display, Audio, LED

Solution:

N/A

Objective 2 **- Static Electricity**

Careful with your CodeX!

A few precautions will keep it safe!

🚨 🚨 Warning!! 🚨 🚨

Static electricity is a charge ⚡ that can build up when you walk across carpet in socks or take off a wool sweater.

- It causes the jolt and spark that happens sometimes when touching something grounded, like a faucet or lightswitch.

Hints**:**

1. Hold your ***CodeX*** by its **edges**, being gentle with the LEDs and other electronic components.
    - They're all exposed on the board as with most other **Maker** computers, so you can *really* get to know them.
2. Keep your CodeX in its case when not in use.
3. It's good practice to touch some grounded metal (desk, doorknob) before handling the CodeX to avoid damaging its sensitive components with static electric discharge.

Goals:

- Close this *Objective panel* to view the 3D scene, and click the *yellow* static electricity lightning bolt at the CPU!
    - Use your mouse to **rotate** the view as needed!
- Click the lightning bolt at the USB connector!
- Click the lightning bolt at the Peripheral Connector!

Solution:

N/A

Quiz 1 **- Static Response**

Question 1: What should you do before handling a CodeX?

✅ Touch some grounded metal

❌ Jumping jacks

❌ Clean it with wet wipes

Objective 3 **- Find the CPU**

Where does the code run?

The Central Processing Unit or 🔧CPU is the brain of the CodeX.

CodeX's CPU is in a *module* with many functions:

1. A *microcontroller* that executes your code.
2. A FLASH filesystem that stores code and data files.
3. Temporary memory (RAM) for a fast-access scratchpad.
4. There's even a built-in Wi-Fi radio!

The CPU also interacts with **all** the other components, lights, display, and peripherals.

- It collects data, issues commands, and pushes display information.

The CPU is an amazing little device!

Can you find the CPU?

Goal:

- Click on the Central Processing Unit (CPU) in the 3D Scene.
    - Hint: You may need to rotate the camera!!

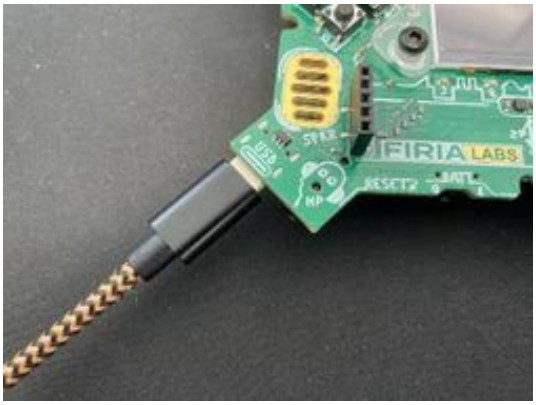
Tools Found: CPU and Peripherals

Solution:

N/A

Objective 4 **- Connect the USB**

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



⚠ Note ⚠

You may see a window pop-up when you plug in the CodeX.

- Feel free to close this window; you won't need it for CodeSpace.

Connecting the 🔧USB cable does two things:

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

Make sure your USB cable is connected now!!

Goal:

- Click on the USB connection port in the 3D Scene.
  - Hint: It is at the bottom of the device!!

Tools Found: USB

Solution:

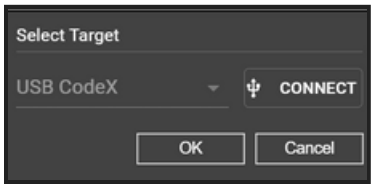
N/A

Objective 5 **- Link to CodeSpace**

Link CodeX to your browser so it can be used with CodeSpace

Connection Steps

1. Make sure the USB cable is connected *both* to your PC and the CodeX.
2. Click the **red** bar **below the code editor** to open the *Select Target* dialog.
   - The *connection bar* looks like this:
     USB CodeX Disconnected - Click to Connect!
   - The bar should look like *this* if your device is already connected:
     USB CodeX Connected



3. In the *Select Target* dialog, click **CONNECT**.
4. The first time your browser connects to a CodeX it will request permission to connect.

- Select **CodeX** from the *device list* and click **Connect**.

Goal:

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

Solution:

N/A

Objective 6 **- Save the Code!**

Time to create a file!

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

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

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

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

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

Your new file should open in your code editor!!

Goal:

- Create a new file named: `Heart1`
    - If this file is already in your file system go ahead and use the *New File...* button anyway!
    - Double check your capitalization!!

Solution:

N/A

Objective 7 **- The CodeTrek**

Check out the CodeTrek!!

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

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

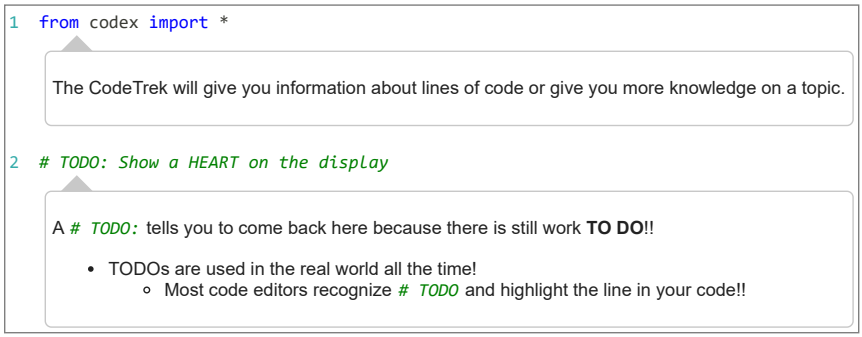
TODOs

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

- It tells you to come back here because there is still work **TO DO**!!
- TODOs are used in the real world all the time!
    - Most code editors recognize `# TODO` and highlight it in your code!!

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

CodeTrek:



Goal:

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

Solution:

N/A

Quiz 2 - Questions TODO

Question 1: What is the CPU's job on the CodeX?

✓ Execute your code

✗ Figure out what you were thinking

✗ Provide +5 Volt power

Question 2: Which of the following is an instruction in a code comment that you need to replace?

✓ `# TODO: fix this`

✗ `# good code below`

✗ `# x should be a float`

Objective 8 - Show Some Heart!

Now it's time for you to run some code!

⚠ Note ⚠

Before you start coding:

- Capitalization matters! Your code is **case sensitive**.
- 🔧Punctuation is important!

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

Time to Type!!

1. Click on the **Code Editor** panel to the left.
2. Remove any sample code that is already there.
3. Type in the code from the **CodeTrek** 🚶
4. Run your code using the RUN ▶ button.

Hints:

- Don't worry about the 🔧colors in the **Code Editor**.
- Use **two** separate lines - be sure to press **ENTER** after the `*`!

CodeTrek:

```
1  from codex import *
2  display.show(pics.HEART)
```

Hint:

- Well, all this *punctuation* has a *purpose*.
  - We are using the **codex module** - pre-loaded code that makes it easier to do things with the CodeX.
  - The `*` means "import **everything**" from that module (it's called a *wildcard*).

Goals:

- Open the CodeTrek 🚶 to see the code.
- RUN ▶ your code to display a **HEART** on the LCD screen!
  - Make sure your code matches the **CodeTrek**!

Tools Found: Punctuation, Syntax Highlighting

Solution:

```
from codex import *
display.show(pics.HEART)
```

***Objective 9* - More Images**

The CodeX has many more pics!

This objective needs a different image.

- It's just a *small* change to the code so you will see a `# TODO` in the change location!!

The CodeTrek keeps you on track.

If you get stuck always refer back to the CodeTrek.

- It can help you get back.

Go ahead and modify your program!!!

- Edit your code and press **Run** to test it out.

You can even try a few other images.

- *Hint*
  - Try `pics.TSHIRT`!!

CodeTrek:

```
from codex import *
# TODO: Show MUSIC on the display
```

The `pics` gallery contains many images:

- Replace the `# TODO` with `display.show(pics.MUSIC)`

Goal:

- RUN ▶ your code to show `MUSIC` on the display!
  - Always check the **CodeTrek**!

Tools Found: CodeX Image Pics

Solution:

```
from codex import *
display.show(pics.MUSIC)
```

Mission 2 Complete

You've completed the first project!

...and you're at the start of a fantastic **adventure**. From this small first project, your journey will take you to greater heights - more projects are ahead to *challenge* and *amaze* you!

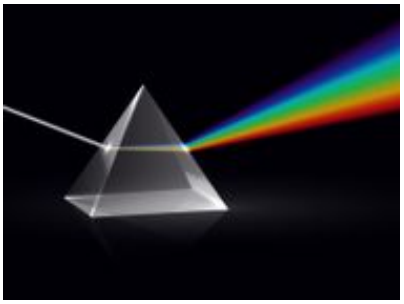
A world of possibilities awaits you...

Mission 3 **- Light Show**

This project explores the CodeX *pixel* LEDs

Have you ever heard of RGB LEDs?

- If not you are in for a real treat!!



The CodeX has a great display, but it also has 10 individual LEDs for you to program!

Four of the LEDs are *Smart* *RGB LEDs* aka Pixel LEDs

Pixel LEDs are powerful Red, Green, Blue (RGB) lights that can be a lot of fun.

Inside each pixel is a set of 3 discrete LED components:

- Red
- Green
- Blue

With just 3 colors you can make the whole spectrum!!!

Project Goals:

- Show a sequence of colors on the Codex pixels
- Vary the speed of color Change

Objective 1 **- Find the Pixels**

RGB pixels

The CodeX has four Red, Green, Blue (RGB) LEDs along the top edge.

- You can set these LEDs to any color under the sun.

The CodeX library gives you a few colors to get started with:

- `BLACK` (this is the same as off!)
- `WHITE`
- `RED`
- `GREEN`
- `BLUE`
- `YELLOW`
- `CYAN`
- `MAGENTA`
- `ORANGE`
- `BROWN`

Can you find the CodeX pixel `0` in the 3d scene?

Goals:

- Create a new file named `Pixels1`
- Click on **RGB** pixel `0` on the CodeX in the 3D scene!

Solution:

N/A

Objective 2 **- Turn on the Red Light**

Start by turning on an RGB pixel!

You can use the following code to set pixel 0 to RED:

```
from codex import *
pixels.set(0, RED)
```

The `pixels.set()` function takes two inputs.

- The first is the *number* of the pixel you want to set
- The second is a *color*

There are some more advanced 🔧pixel LED features you can use later.

It's pretty obvious what the `pixels.set(0, RED)` is telling the CodeX to do.

- But what about the `from codex import *`?

Click to learn more about the 🔧import statement.

CodeTrek:

```
1  from codex import *
2  pixels.set(0, RED)
```

You can `set` pixels 0 - 3 to many different colors.

- Try `BLUE` or `GREEN`

Goal:

- Light up the **CodeX** pixel `0` the color `RED`.
    - Make sure your code matches the CodeTrek!

Tools Found: RGB "pixel" LEDs, import

Solution:

```
1  from codex import *
2  pixels.set(0, RED)
3
```

***Objective 3* - Two in a Row?**

Now display two colors in sequence

⚠ Note ⚠

The code in the objective may NOT do what you expect! *Read Carefully!*

The computer executes your code sequentially

- Starting with **line 1**, the **line 2**, and so on.
- Oh, and... computers are **very fast**.

When you write code, it often doesn't work the way you planned the *first time*. Part of the joy of *coding* is figuring out **why** - and *fixing* it!

Check the CodeTrek for coding hints.

CodeTrek:

```
1  from codex import *
2  pixels.set(0, RED)
```

```
3  # TODO: set pixel 0 to GREEN
```

ALWAYS Update your `# TODO:`s!

Also...

- You might never see the color `RED`:
- `GREEN` will show up so fast that the red is lost to your eye.

Goal:

- Light up pixel `0` `RED` and then change it to `GREEN`.

Solution:

```
1  from codex import *
2  pixels.set(0, RED)
3  pixels.set(0, GREEN)
```

Objective 4 **- What's Going On?**

Why is only the last color showing up?

Hey, at least your program did *something* different! The pixel is now `GREEN`. But the goal is to see **both** images clearly, one at a time...

Notice that your program ENDS very quickly

- It doesn't wait for you to see the *first* color before it shows the *second* one.

CONCEPT: Hardware 🔧Peripherals

- Hardware that's connected to your 🔧CPU can remain active, even **after** your program ends.
- For example, the Pixel LEDs keep shining with the last color you sent them.
- Many computer 🔧peripherals operate *independently* of the computer itself!

Theory: Both colors **are** being displayed, but:

- `RED` is only displayed for a *very* short time (too fast to see)
- `GREEN` color is the **last** thing displayed, so the **LED** keeps showing it even *after* the program ends.

Now, test this theory with a couple more colors.

CodeTrek:

```
1  from codex import *
2  pixels.set(0, RED)
3  pixels.set(0, GREEN)
4  # TODO: Set pixel 0 BLUE
5  # TODO: Set pixel 0 WHITE
```

You can replace these `# TODO`s with code!

- Blue is `BLUE`
- White is `WHITE`

Goal:

- Light up pixel `0` with colors in the following order: `RED`, `GREEN`, `BLUE`, `WHITE`.
    - You must use the codex library color keywords.

Tools Found: CPU and Peripherals

Solution:

```
from codex import *
pixels.set(0, RED)
pixels.set(0, GREEN)
pixels.set(0, BLUE)
pixels.set(0, WHITE)
```

Quiz 1 - Two Images

Question 1: What do you expect the following code to do?

```
from codex import *
display.show(pics.HEART)
display.show(pics.HAPPY)
```

✓ Display each image quickly and end showing the last one

✗ Display only the first image

✗ Show the images for about 1 second each

Objective 5 - Find the Bug

Inside the Mind of the Computer!

Computers are fast. Even a small computer like the CodeX can execute *millions* of operations per second!

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

Watch the video below: All the colors *are* being displayed!

It's easy to see ALL the **Pixel colors** when the program goes *slowly, step-by-step.*

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

Find the debug button and prepare for stepping in the next objective!

Goal:

- Enter **DEBUG** mode on the CodeX by pressing the Debug Program button.

Tools Found: Debugging

Solution:

N/A

Objective 6 - Step by Step Colors

Your turn to use the debugger!

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

CONCEPT: *Stepping*

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

Try stepping through your code!

1. Press the **Debug** button to re-load your program and wait at the first line
2. Then use the **Step In** button to execute each line of code
3. The highlighted line executes **after** you click **Step In**
4. Then the *next* line of code is highlighted, waiting and ready to goes
5. Check the CodeX pixel *after* each STEP

CodeTrek:

```
1  from codex import *
2  pixels.set(0, RED)
3  pixels.set(0, GREEN)
4  pixels.set(0, BLUE)
5  pixels.set(0, WHITE)
```

Goal:

- Use the debugger **Step In** button to show the different colors.
    - You will need to hit the debug button again first.
    - You must step at least 5 times!

Tools Found: Debugging

Solution:

```
1  from codex import *
2  from time import sleep
3  pixels.set(0, RED)
4  pixels.set(0, GREEN)
5  pixels.set(0, BLUE)
6  pixels.set(0, WHITE)
```

Objective 7 - Slow it Down

When you *step slowly* through the code, all the colors show up. So you just need a way to delay the computer a little after it shows each color.

```
1 from time import sleep
2 sleep(1) # delay for 1 second
```

Line 2 in the code above will cause the CodeX to delay for 1 second before going to the next line.

- Plenty of time to see a new color displayed!!

Update your code!

Add a line with `sleep(1)` on the *next* line of code **after** each `pixels.set()`.

See the CodeTrek if you need help!

CodeTrek:

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

```
Make sure you add the from time import sleep
    • This lets you call sleep() in your code!!


pixels.set(0, RED)
sleep(1)
pixels.set(0, GREEN)
sleep(1)
pixels.set(0, BLUE)
# TODO: Sleep for 1 second
    Sleep takes a delay time in seconds.

pixels.set(0, WHITE)
```

Goals:

- Import the `sleep` function from the `time` module.
- Use a whole number in the `sleep()` function to delay for that many seconds.

Tools Found: Timing

Solution:

```
from codex import *
from time import sleep

pixels.set(0, RED)
sleep(1)
pixels.set(0, GREEN)
sleep(1)
pixels.set(0, BLUE)
sleep(1)
pixels.set(0, WHITE)
```

Objective 8 **- Name that Number**

Variable Speed?

It would be fun to play with some different delay times. Right now the number `1` appears *three* times in the code, and **all** must be changed to adjust the delay between colors!

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

CONCEPT: *Variables*

A 🔧variable is a *name* to which you 🔧assign some *data*. The *data* could be a number, a color, or any other type of information your program uses.

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

```
delay = 1
```

The line of code above defines a variable `delay` that can now be used anywhere in the program below it, in place of `1`. The best part is, you can now change that value in *one place* in your code!

Now that you're *up to speed* - it's time to...

Update your code!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # TODO: Create a variable named delay
```

Add the `delay` 🔧variable:

```
delay = 1
```

```
5
6  pixels.set(0, RED)
7  sleep(delay)
8  pixels.set(0, GREEN)
9  sleep(delay)
10 pixels.set(0, BLUE)
11 # TODO: Sleep for the delay
```

Use your new `delay` in all your `sleep()` calls!!

```
12 pixels.set(0, WHITE)
```

Goals:

- Use a 🔧variable called `delay` to set your time delay.
- Use the `delay` 🔧variable in the `sleep()` function.

Tools Found: Variables, Assignment

Solution:

```
from codex import *
from time import sleep

delay = 1

pixels.set(0, RED)
sleep(delay)
pixels.set(0, GREEN)
sleep(delay)
pixels.set(0, BLUE)
sleep(delay)
pixels.set(0, WHITE)
```

Quiz 2 - Variable Questions

Question 1: What does `from codex import *` do?

✓ Provides access to built-in codex code

✗ Turns on the codex LEDs

✗ Imports asterisks from the land of codex

Question 2: What does `delay = 1` do?

✓ Assigns the value 1 to a variable named 'delay'

✗ Delays execution for 1 second

❌ Puts the CPU into sleep mode for 1 second

***Objective 9* - Warning Sign**

Light em all!!

Time to light up all the 🔧pixel LEDs!

- And you are going to **create** a flashing warning sign!!

`pixels.set()` can be used to light all 4 pixels.

You can also set a 🔧variable to a color.

- Later in your program you can change the **value** of the `color` variable!

One way to set all pixels to the same color is like this:

```
color = RED
pixels.set(0, color)
pixels.set(1, color)
pixels.set(2, color)
pixels.set(3, color)
```

Finish this project by flashing between `RED` and `YELLOW` on all 4 pixels.

- Make sure to put a 🔧delay between flashes!

Wanna copy some code?

You can use the 🔧Editor Shortcuts to copy and paste lines of code!

CodeTrek:

```
from codex import *
from time import sleep

delay = 1

# TODO: Set a variable called color to RED
```

> You can set a 🔧variable to anything.
>
> - Even a color!!
>
> ```
> color = RED
> ```

```
pixels.set(0, color)
pixels.set(1, color)
pixels.set(2, color)
pixels.set(3, color)

sleep(delay)

# TODO: Change the color variable to YELLOW
```

> **Now change colors**
>
> You can use `YELLOW` to contrast with the `RED`...
>
> - Or check out the 💡 Hints for how to create *custom* colors!

```
pixels.set(0, color)
pixels.set(1, color)
pixels.set(2, color)
pixels.set(3, color)

sleep(delay)
```

```
21
22  color = RED
23  pixels.set(0, color)
24  pixels.set(1, color)
25  pixels.set(2, color)
26  pixels.set(3, color)
27
28  sleep(delay)
29
30  color = YELLOW
31  pixels.set(0, color)
32  pixels.set(1, color)
33  pixels.set(2, color)
34  pixels.set(3, color)
```

This code is getting long...

Soon you will learn how to shorten this up!

Hints:

- **Longing for a Loop ?**

  This objective repeats a block of code multiple times!

  - There is a concept we will cover soon that allows you to repeat things much more neatly - loops.
  - For now just copy and paste the code a few times!

- **Custom Colors ?**

  The standard set of colors you've used so far are actually just constants defined in a Python module.

  You can define your *own* special colors based on red/green/blue combinations. Check out the RGB Colors tool for more on that!

Goals:

- Create a second variable called `color` and set it to the `RED` color from the `codex` library.
- Use the `color` variable as the second argument in the `pixels.set()` function.

Tools Found: RGB "pixel" LEDs, Variables, Timing, Editor Shortcuts, Keyword and Positional Arguments

Solution:

```
1  from codex import *
2  from time import sleep
3
4  delay = 1
5
6  color = RED
7  pixels.set(0, color)
8  pixels.set(1, color)
9  pixels.set(2, color)
10 pixels.set(3, color)
11
12 sleep(delay)
13
14 color = YELLOW
15 pixels.set(0, color)
16 pixels.set(1, color)
17 pixels.set(2, color)
18 pixels.set(3, color)
19
20 sleep(delay)
21
22 color = RED
```

```
23  pixels.set(0, color)
24  pixels.set(1, color)
25  pixels.set(2, color)
26  pixels.set(3, color)
27
28  sleep(delay)
29
30  color = YELLOW
31  pixels.set(0, color)
32  pixels.set(1, color)
33  pixels.set(2, color)
34  pixels.set(3, color)
```

Mission 3 Complete

You can set the pixels to any color you want!

There are so many ways to create fun lighting schemes.

Lights like these are used in so many applications in the real world

- Traffic lights
- Stadium scoreboards
- Concert special effects
- Smart home lighting

Mission 4 - Display Games

This project explores the display

From car dashboards to giant stadium scoreboards, you see LED displays **everywhere**, and most of them are controlled by software.

The CodeX display is small, but with *your code*, it can do a lot!

Project Goals:

- Display and print **text message** *strings*
- Program buttons to determine whether they are pressed
- Make a timed *find a button* **game**

Objective 1 - Back to the Display

You are going to build a game...

But you should practice showing another image on the display first.

Remember how to show an image?

```
display.show(pics.PLANE)
```

*If you are confused click on the **CodeTrek** button!*

CodeTrek:

```
1  from codex import *
2  display.show(pics.PLANE)
```

Goals:

- Create a **New File** named `Display`.
- Show a `PLANE` on the display.
  - Just give `display.show()` one argument, `pics.PLANE` from the CodeX pics library.

Tools Found: Keyword and Positional Arguments, CodeX Image Pics

Solution:

```
1  from codex import *
2  display.show(pics.PLANE)
```

Objective 2 - Text Messages

Not my type...

You've shown lots of images on the display. But can it *also* display **text**?

- Experiment to see if `display.show()` can show `"text"`, and not just image data.

CONCEPT: Data Types

Your code already works with *Data* types:

- `pics.PLANE` - a CodeX image type
- `1` - an integer type

The **type** for text like "Hello" is called string

- `"Hello"` - a **string** type
- *(Strings must be in **quotation marks**)*

Try to make the CodeX display a text **string** message!

CodeTrek:

```
1  from codex import *
2  display.show("Ahoy")
```

You can use `display.show()` to draw a text message on the display!

Goal:

- Use the 🔧string `"Ahoy"` inside the `display.show()` function.

Tools Found: Data Types, CodeX Image Pics, int, str, bool

Solution:

```
1  from codex import *
2  display.show("Ahoy")
```

***Objective 3* - Good With Numbers?**

Fancy a Bit o' Maths?

You might have heard that computers are good at doing mathematics. Time to put that to the test!

Assigning a Calculation

You already know how to define a 🔧variable.

- Previously you assigned a literal `1` to a variable name "delay".
- Now try assigning a *simple calculation* to a variable, and display the result!

Change your program to calculate `num` and `display.show()` the result.

Keep it simple for now:

```
num = 2 + 2
```

...What could go wrong?

⚠ Note ⚠

Unexpected Result Ahead...

- You will see a **TypeError** when you Run this code!!!

Complete this Objective by causing the error - next Objective will fix it!

CodeTrek:

```
1  from codex import *
2  num = 2 + 2
```

`num` is a variable that is set to an 🔧integer

- `2 + 2` is the same as `4`

- So the `num` variable contains `4`

```
3  display.show(num)
```

Use `display.show()` to show the integer.

- This may not work how you expect!!

Hint:

- **What's the Error?**

  When you run the code, an **error message** will appear. That's because `display.show()` only works for certain types of data - like **strings**.

  Your code gave a *number* (an **Integer** or int type) to `display.show()`, producing an error.

  Don't worry, you'll fix this error in the next Objective!

Goal:

- Try to `display.show()` an integer.
  - This will cause a **TypeError**.

Tools Found: Variables, int

Solution:

```
1  from codex import *
2  num = 2 + 2
3  display.show(num)
```

Objective 4 - Converting Types

Fix it Up!

You've discovered `display.show()` doesn't work with integer types.

- But you know it *does* work with strings.
- *Fortunately, **Python** makes it easy to convert back and forth between these types!*

CONCEPT: Data Type Conversions

- `str(n)` Convert 'n' to a String
- `int(s)` Convert 's' to an Integer

Now modify your program:

- **Replace** `display.show(num)` with `display.show( str(num) )`

CodeTrek:

```
1  from codex import *
2  num = 2 + 2
3  # TODO: show num after converting to a string
```

You can use the `str()` function to convert your `num` variable.

```
display.show(str(num))
```

Goal:

- Convert the `num` variable to a string using the `str()` function.

Tools Found: int, str, Data Types

Solution:

```
from codex import *
num = 2 + 2
display.show( str(num) )
```

Objective 5 - Second Show Message

Can you display two messages?

Use the `display.show()` function to show **two** strings.

```
display.show("Hello")
display.show("World")
```

⚠ **Note:** You may not get the result you expect here!

CodeTrek:

```
from codex import *
display.show("Hello")
# TODO: show a second message
```

Use the `display.show()` function to show a second string on the display.

Goal:

- Now use `display.show()` to show a second line of text on the display.
  - You need to call the function twice!

Tools Found: str

Solution:

```
from codex import *
display.show("Hello")
display.show("World")
```

Objective 6 - Printing Text

Oh no! We've seen this problem before.

The `display.show()` function shows one string at a time. It will overwrite any text that was there.

- That's okay, but we don't always want to lose our messages.
- You could use the debugger to see the two messages.
  - Give it a try - are they both printing when you STEP through the code?

BUT *CodeX* has a better way to show text messages

Change your program to use `print` instead of `show`.

```
display.print("Hello")
display.print("World")
```

Try your skills

Play around with some different messages of your choice.

- The computer doesn't care what text you put inside the quotes
  - *Except for some symbols* which require 🔧Escape Sequences

CodeTrek:

```
1  from codex import *
2  display.print("Hello")
3  # TODO: "print" a 2nd text message to the display
```

You need to use `display.print()` here.

- `display.show()` will just overwrite your old 🔧string.
- `display.print()` can show multiple lines of text.

Goal:

- Now use `display.print()` to show two lines of text on the display.
  - You need to call the function twice!

Tools Found: str, Advanced Debugging, Escape Sequences

Solution:

```
1  from codex import *
2  display.print("Hello")
3  display.print("World")
```

Quiz 1 - Typed

Question 1: Which of the following is **NOT** a standard Python type?

✓ `'text'`

✗ `'int'`

✗ `'str'`

Question 2: What will happen if you run this code?

```
from codex import *
display.show(8)
```

✓ The program will error

✗ The display will show an 8

✗ An 8 will be shown below any text already on the screen

Objective 7 - Branching

The final stage of this project is to make a GAME that works like this:

- The display will tell you which button to press.
- You will have **1 second** to press and hold the correct button.
- If you are holding the button within 1 second a pixel will light up `GREEN`!

The first step is to light up a pixel

Here's the plan:

- **If** a specific button was pressed **then**
  - A pixel turns `GREEN`
- **Otherwise,**
  - That pixel turns `RED`

In *Python* code it looks like *(don't type this yet)*

```
if pressed:
    pixels.set(0, GREEN)
else:
    pixels.set(0, RED)
```

Your code will take a **different** 🔧branch depending on the value of `pressed`

CONCEPT: Branching

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

Okay, there's a lot of information to take in above! (...take your time)

Actually just watching the code run will help you understand the `if` statement

- Go ahead - type in the code from the **CodeTrek** and **try stepping through it!**
- Be careful with the 🔧indentation on lines after `if:` and `else:` statements.
- A colon `:` always precedes an **indented** block of code.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  sleep(1)
```

Sleep for `1` second waiting on the user to press and hold a button.

```
5  pressed = True
```

For now just simulate the button is pressed

- Make your 🔧variable equal to the 🔧bool `True`

You can try it out with `False` as well!!

```
6  if pressed:
7      pixels.set(0, GREEN)
```

Set pixel 0 to `GREEN` if the button is pressed at the end of 1 second.

- **Don't forget to 🔧indent your code!!**

```
8  else:
9      pixels.set(0, RED)
```

Set pixel 0 to `RED` if the button is not pressed at the end of 1 second.

Goals:

- Set a variable named `pressed` equal to the boolean `True`.
- Use an `if` statement in your code followed by an `else` statement.
  - if `pressed` is `True` go **GREEN**
  - ...otherwise go **RED**

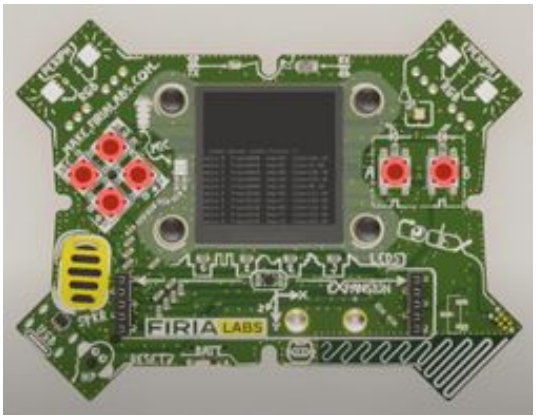
Tools Found: Branching, Indentation, bool, Variables

Solution:

```
from codex import *
from time import sleep

sleep(1)
pressed = True
if pressed:
    pixels.set(0, GREEN)
else:
    pixels.set(0, RED)
```

Objective 8 **- Button Hunting**

The second step is to find the buttons!



Search the 3D scene for the different buttons!

- `BTN_A` is Button A
- `BTN_B` is Button B
- `BTN_L` is the **Left** Button
- `BTN_U` is the **Up** Button
- `BTN_R` is the **Right** Button
- `BTN_D` is the **Down** Button

Click each button and *Watch the Goals turn GREEN*

Keep your eyes on the Goals HUD at lower left of 3D panel!

Goals:

- Click on `BTN_A` (button A) in the 3D Scene.
- Click on `BTN_B` (button B) in the 3D Scene.
- Click on `BTN_L` (button L) in the 3D Scene.
- Click on `BTN_U` (button U) in the 3D Scene.
- Click on `BTN_R` (button R) in the 3D Scene.
- Click on `BTN_D` (button D) in the 3D Scene.

Solution:

N/A

Objective 9 **- Gamer Input**

Gotta grab some *User Input* for your game!

There are a few different ways to access the 🔧CodeX buttons, including:

- `buttons.was_pressed(BTN_A)` True if button has been pressed since last check
- `buttons.is_pressed(BTN_A)` True if button is currently pressed

Now it's time to *button* this thing up!

You will need to check **IF** your button is pressed.

*Use **Button A** for your first test!*

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  display.show("Hold Button A")
```

Tell the game player which button they need to hold before the 1 second 🔧delay starts.

- That gives them the chance to find it!

```
5  sleep(1)
6  pressed = buttons.is_pressed(BTN_A)
```

Set your variable to the value of `buttons.is_pressed(BTN_A)`

- Try out `BTN_A` first but later you will have multiple buttons!

```
7  if pressed:
8      pixels.set(0, GREEN)
9  else:
10     pixels.set(0, RED)
```

Goals:

- **Replace the "hard coded" `pressed` value `True` with a button function:**
  - Use the `buttons.is_pressed()` function with `BTN_A` as the 🔧argument.
- **Set pixel `0` inside your `if` statement.**
  - Your code already does this, I think...

Tools Found: CodeX Buttons, Keyword and Positional Arguments, Timing

Solution:

```
1  from codex import *
2  from time import sleep
3
4  display.show("Hold Button A")
5  sleep(1)
6  pressed = buttons.is_pressed(BTN_A)
7  if pressed:
8      pixels.set(0, GREEN)
9  else:
10     pixels.set(0, RED)
11
```

***Quiz 2* - Buttons**

Question 1: What code will tell me if the **UP** button is currently pressed?

✓ `buttons.is_pressed(BTN_U)`

✗ `buttons.is_pressed(BTN_D)`

✗ `buttons.was_pressed(BTN_U)`

Question 2: What will happen if you run this code?

```
x = False
if x:
    display.show('if')
else:
    display.show('else')
```

✓ `'else'` will print on the display

✗ Your program will error

✗ `'if'` will print on the display

***Objective 10* - For The Win!**

All the pieces are in place

Now, just check a few more buttons and you've got a serious *twitch game!*

You can use whichever buttons you want.

Game Play Sequence

1. Screen prompts the user with *first* button to press... *hurry!*
2. Good press - yay! **GREEN light.**
3. Screen prompts with *second* button... *FAST!*
4. Oops - didn't get there in time. **RED light.**
5. ...repeat for *third* and *fourth* buttons.

*All four LEDs **GREEN** for the **WIN!!***

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  display.show("Hold Button A")
5  sleep(1)
6  pressed = buttons.is_pressed(BTN_A)
7  if pressed:
8      pixels.set(0, GREEN)
9  else:
10     pixels.set(0, RED)
11
12 display.show("Hold Button U")
```

Previously you waited for `BTN_A`.

- Now add a step to wait for *another* button.
- You should do this for all 4 pixels on the CodeX!

Use the 🔧Editor Shortcuts to make this easier!

```
13 sleep(1)
14 pressed = buttons.is_pressed(BTN_U)
```

You don't need to use `BTN_U` but that is a good starting point!

```
15 if pressed:
16     pixels.set(1, GREEN)
```

```
else:
    pixels.set(1, RED)
```

Make sure you are setting pixel `1` this time.

- Each button input should set the next pixel!!
- At the end all 4 pixels should be lit either `GREEN` or `RED`.

```

# TODO: Add a 3rd button

# TODO: Add a 4th button
```

Add a 3rd and 4th button input to your game.

- Don't forget to set a different pixel each time!

Goals:

- Check for 4 different `buttons.is_pressed()`.
  - You need to call the function 4 separate times.
- Set each pixel inside an `if` statement.
  - Be sure to light up all 4 pixels: 0, 1, 2, and 3

Tools Found: Timing, Editor Shortcuts

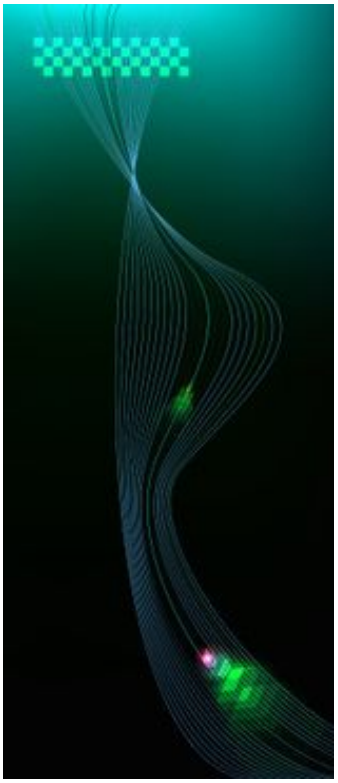
Solution:

```
from codex import *
from time import sleep

display.show("Hold Button A")
sleep(1)
pressed = buttons.is_pressed(BTN_A)
if pressed:
    pixels.set(0, GREEN)
else:
    pixels.set(0, RED)

display.show("Hold Button U")
sleep(1)
pressed = buttons.is_pressed(BTN_U)
if pressed:
    pixels.set(1, GREEN)
else:
    pixels.set(1, RED)

display.show("Hold Button L")
sleep(1)
pressed = buttons.is_pressed(BTN_L)
if pressed:
    pixels.set(2, GREEN)
else:
    pixels.set(2, RED)

display.show("Hold Button B")
sleep(1)
pressed = buttons.is_pressed(BTN_B)
if pressed:
    pixels.set(3, GREEN)
else:
    pixels.set(3, RED)
```

Mission 4 Complete

Python is great for coding games

and you're just getting started!



You'll soon discover a lot more possibilities as you learn more about the **CodeX**, and learn to build more complex software with text-based code.

Real World Applications

Reading buttons and controlling LEDs... Not to mention making split-second timing decisions!

- Ever used a *remote control?*
- How about smart lighting applications for homes and schools?
- And of course, fast-twitch button-press games :-)

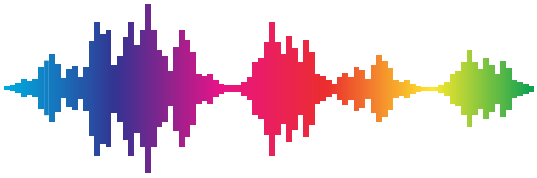
This kind of code is all around you!

Nice work!!

Mission 5 - Micro Musician

Musicians often use computers to help create music

- Drum Machines
- Keyboard synthesizers
- Recording and Mixing with Digital Audio Workstation (DAW) Software
- *...lots more ways - can you think of some?*



This is a short project, just to give you a taste of what the CodeX can do in the area of *electronic music.*

As with the display, your software can take **complete** control over the output of the CodeX, and in *future* projects you will do more customization!

Project Goals:

- Play some of the CodeX's built-in sounds
- Learn about some code formatting to improve your code's 🔧readability

Ready to make some noise?

Objective 1 - Sound Outputs

The CodeX has two places to output sounds

- You can listen to music through the speaker, ***or***
- You can plug headphones into the headphone jack

Speakers and headphones work by converting electrical signals into mechanical *waves*.

Your Python code can control sound output using the 🔧audio functions.

- Play sound files, beep tones, control volume, and more!
- Check out the 🔧audio help in your toolbox for all the details.

Goals:

- Find the CodeX speaker in the 3D view.
- Find the CodeX Headphone Jack in the 3D View.

Tools Found: Audio

Solution:

N/A

Objective 2 - Micro Tunes

Now it's time to write code to play some sounds!

When you do `from codex import *` you get access to the `audio` object.

The 🔧audio object gives you lots of *sound* tools!

- You are going to start by playing an mp3 file.
  - An **mp3** is just an *audio file* in the mp3 **format**.

- Your CodeX has a few sample mp3 files already loaded!

Here is an example:

```
from codex import *
audio.mp3("sounds/welcome")
```

That's pretty simple code!

Using the CodeX 🔧audio library you can play recorded sounds *or* create your own *custom music!*

CodeTrek:

```
1  from codex import *
2  audio.mp3("sounds/welcome")
```

Use the `audio.mp3` function to play a song on the CodeX.

- This is the easiest way to get started with music!!

Goals:

- Create a new file named `"Music1"`.
- Use the CodeX library's `audio.mp3()` function to play a song.

Tools Found: Audio

Solution:

```
1  from codex import *
2  audio.mp3("sounds/welcome")
```

Objective 3 **- Clean Code**

Good code is easy to read (by *humans*, not just computers!)

As your programs get longer, take care to make them *readable*.

- You can add 🔧blank lines anywhere in your code to separate portions without affecting how it works.
- The computer ignores blank lines.

Try adding a **blank line** in between your lines of code.

Update your code to look like this:

```
from codex import *

audio.mp3("sounds/welcome")
```

Try running your code and make sure it works the same as before!

CodeTrek:

```
1  from codex import *
2
```

Add a **blank line** here just to improve 🔧readability.

```
3  audio.mp3("sounds/welcome")
```

Goal:

- Add a blank line to your code!

Tools Found: Blank Lines and Whitespace, Readability

Solution:

```
from codex import *

audio.mp3("sounds/welcome")
```

Objective 4 **- Once More, With Feeling**

You don't want the display to be dark while those inspiring tunes are playing!

- Why don't you show `pics.MUSIC` !!

You may want to spend more time experimenting with the built-in 🔧CodeX sound collection...

- In a *future* lesson, you will learn how to create your own tunes.

Here are some of the songs already loaded in 🔧Codex sound:

- `"africa.mp3"`
- `"techstyle.mp3"`
- `"shire.mp3"`

and of course,

- `"roll.mp3"`

Go ahead ahead try some different songs after you complete this objective!

CodeTrek:

```
from codex import *

display.show(pics.MUSIC)
# TODO: Play the africa.mp3 song
```

Use the `audio.mp3()` function here.

- The `audio` object comes from the `codex` library.

Hint:

- Remember to use the `"sounds/..."` path to load built-in sounds.

```
# Example:
audio.mp3('sounds/roll')
```

Goals:

- Show the `pics.MUSIC` on the CodeX display.
- Play the `"africa.mp3"` song from the CodeX music collection.
  - You can omit the **.mp3** extension on the filename if you prefer.

Tools Found: CodeX Sound Collection

Solution:

```
from codex import *

display.show(pics.MUSIC)
audio.mp3("sounds/africa")
```

***Objective 5* - Comments**

Readability and Comments

As you write code, imagine that someone who has never seen it before will have to read it and figure it out.

- A year from now, you might even pick up your **own** code and say: "what was I thinking!?"

🔧**Readability**: Making code easy to understand for ***humans.***

- Use descriptive variable names
- Use 🔧Comments - notes in the code about what you're doing

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

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

Check the CodeTrek for some comments.

In the following projects, *you can decide* if you want to type in the 🔧comments provided in the lessons, or omit them.

Feel free to *add your own comments,* to help you understand and remember what your code was meant to do!

CodeTrek:

```
from codex import *

# Display the music.
```

This is what a comment looks like.

- Note that per the Python style guide the first letter is capitalized, and it ends with a period.

```
display.show(pics.MUSIC)

# TODO: Add your own comment here.
```

Replace the `# TODO` with your own comment!

```
audio.mp3("sounds/roll")
```

Goal:

- Add 2 comments to your code, and RUN it again!

Tools Found: Readability, Comments

Solution:

```
from codex import *

# Display the music
display.show(pics.MUSIC)

# Your own comment here
audio.mp3("sounds/roll")
```

***Quiz 1* - Readable Quiz**

Question 1: Choose the two places that the CodeX can output sound.

- ✓ The Headphone Port
- ✓ The Speaker
- ✗ The Display
- ✗ The USB Port

Question 2: Which of these is **NOT** a tool to make your code more readable?

- ✓ Variable names without meaning (like `x`)
- ✗ Blank lines in your code
- ✗ Comments that explain your code

***Objective 6* - Portable MP3s**

After your code is *running* on the CodeX.

> **You can go *unplugged*!**

Your projects *don't need a computer attached* after coding.

> What will you create with this **portable power**?

🚨 🚨 Warning!! 🚨 🚨

Be Gentle with Cables!

When you **unplug** a cable, **DO NOT PULL** on the **wire!**

- Hold the **connector** firmly when you *disconnect* and *connect*.

Take it for a Spin

1. Download your code to the CodeX
2. **Disconnect** your CodeX from the USB cable
3. Flip the **BATT** switch to position **1**
4. Wait a few seconds for the program to start...
5. (Did you put batteries in your CodeX?!?!)

The CodeX is an **embedded computer** - meaning you can build it *into* other projects to sense and control stuff in the *physical* world!

Goal:

- Click on the **BATT** switch in the 3D scene!
  - **Hint:** You may need to turn the CodeX around and click on it from behind!

Solution:

N/A

Mission 5 Complete

You can do MUCH more with sounds on the CodeX

In future lessons, you will explore more capabilities AND compose your **own** songs!

Well done! Move ahead to more coding fun...

Mission 6 **- Heartbeat**

In this project you'll give the CodeX a *beating heart.*

Okay, not a *real* heart - that would be a little too messy!

> **But** using the display you can give the CodeX its own *digital* heart, and even make it speed up and slow down just like your own heart does.

Project Goals:

- Code an animated heartbeat, pulsing on the LED display
- Learn how to make your code **LOOP** forever
  - And how to break out of it!
- Make the heartbeat speed *adjustable* using the 🔧CodeX buttons
  - **A** = *slower* and **B** = *faster*

Objective 1 **- Lots of Heart**

Show a heart on the screen!

You might recognize this as the same code as your first project.

> Don't worry, you're going to add a *lot* of new features soon!

CodeTrek:

```
from codex import *

display.show(pics.HEART)
```

Goals:

- Create a **new** file named `Heart2`.
- Show `pics.HEART` on the CodeX display.

Solution:

```
from codex import *

display.show(pics.HEART)
```

Objective 2 **- Pump It UP**

To make a *Heartbeat* animation, the display needs to alternate between two images:

- The first image you're already showing, `pics.HEART`
- ... then switch to `pics.HEART_SMALL`

> Each **beat** will be a *big / small* cycle, to make the Heart appear to *pulse!*

In this step, you should start with a *single* beat.

⚠ Note ⚠

> Remember from the previous lesson that you need a 🔧delay if you want to see the first heart!
>
> - The computer will not delay for you!!!

CodeTrek:

```
from codex import *
from time import sleep
```

```
You need to import sleep from the time module.
  • This lets you access the sleep() function.

3
4  display.show(pics.HEART)
5  sleep(1)
Add a 🔧delay for 1 second.

6
7  # TODO: Show pics.HEART_SMALL
Use display.show() to show pics.HEART_SMALL.

8  sleep(1)
Add a second 🔧delay at the end.
  • You will need this in the next step!
```

Goals:

- Show the larger `pics.HEART` on the CodeX display.
- Show the smaller `pics.HEART_SMALL` on the CodeX display.
- Add a 🔧delay to your code!

Tools Found: Timing

Solution:

```
1  from codex import *
2  from time import sleep
3
4  display.show(pics.HEART)
5  sleep(1)
6
7  display.show(pics.HEART_SMALL)
8  sleep(1)
```

Objective 3 **- Animated Beats**

Repeat da Beat

Now that you have a *single* **beat**, can you make it repeat?

Go ahead, **change your code** to make the HEART beat several times!

(no need to repeat `from codex import *` - you only need that once!)

Just repeat these lines in your code a few more times:

```
display.show(pics.HEART)
sleep(1)

display.show(pics.HEART_SMALL)
sleep(1)
```

Fingers tired of typing? **Learn about the 🔧Editor Shortcuts!**

Run your program

You can code a few *beats* this way, but the program would get *really* long if you had to **copy/paste** the code to make the heartbeats go on much longer!

...there **must** be a better way!

CodeTrek:

```
from codex import *
from time import sleep

# first beat
display.show(pics.HEART)
sleep(1)

display.show(pics.HEART_SMALL)
sleep(1)

# second beat
# TODO: Repeat the heartbeat
```

Repeat these the HEART beat animation a few times.

Goal:

- Use `display.show()` followed by a `sleep()` at least 4 times in your program.

Tools Found: Editor Shortcuts

Solution:

```
from codex import *
from time import sleep

# first beat
display.show(pics.HEART)
sleep(1)

display.show(pics.HEART_SMALL)
sleep(1)

# second beat
display.show(pics.HEART)
sleep(1)

display.show(pics.HEART_SMALL)
sleep(1)
```

Objective 4 **- Hearts Forever**

A few beats is a good start...

But your Heartbeat animation needs to run forever!

Instead of copying the same code *over and over and over...* there **must** be a way to tell the computer to just **repeat** that code!

YES there is!

It's called a 🔧LOOP.

Open the **CodeTrek**. It shows your original code inside a `while True` loop.

Can you tell what this code might do?

Study the definitions below, then run that code

CONCEPT: While Loops

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

The CodeTrek uses the *literal* value `True` as the condition, so we have an **infinite loop** - one that never ends, because `True` is always... **True**!

Now it's your turn. *To infinity and beyond!*

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Keep displaying beats forever
5  while True:
```

This is the **infinite** loop!

```
6      display.show(pics.HEART)
```

Notice the **indentation** here.

- Use the **TAB** key on your keyboard to **indent** code.
  - That's the same as 4 spaces, but easier

```
7      sleep(1)
8
9      display.show(pics.HEART_SMALL)
10     sleep(1)
```

All the **indented** lines below the `while` loop are **IN** the loop.

- Notice you only need ONE big/small "cycle" in the loop!

Goals:

- Add a `while True` loop to your code.
- Make sure you indent properly for your loop!
  - You must have **4 spaces** or a single **tab** for your indent!!

Tools Found: Loops, Indentation, bool

Solution:

```
from codex import *
from time import sleep

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(1)

    display.show(pics.HEART_SMALL)
    sleep(1)
```

***Objective 5* - Stop It!**

Now that your program doesn't just run straight through and finish, you need a way to **STOP** it...

Click the ⏹ STOP button to exit your running code.

You might have noticed that CodeSpace automatically stopped your code when you moved to the next **Objective**. *But you can stop and start on your own also!*

Feeling the Need for Speed??

Want to make the HEART beat faster? *...or slower?*

- All you have to do is change the `sleep(1)` to a different delay.

Click the **STOP** button so you can edit your code!

- **Now,** play around with a few different values, like `sleep(2)`.
  - ...or `sleep(0.5)`
- **Run** and then **Stop** your program a few times, trying different speeds until you're ready to move on.

CodeTrek:

```
from codex import *
from time import sleep

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(1)

    display.show(pics.HEART_SMALL)
    sleep(1)
```

Goals:

- Use the **infinite** `while True` loop in your program.
- Use the **STOP** Button to stop your running program.

Solution:

```
from codex import *
from time import sleep

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(1)

    display.show(pics.HEART_SMALL)
    sleep(1)
```

***Objective 6* - Heart Break**

Now that you have coded an *infinite loop* and learned how to manually stop it, you're probably wondering: "is there a way to break out of it with **code?**"

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

🔧break

Your new assignment:

- Program one of the 🔧CodeX buttons as a *"kill switch"* to stop the ever-beating heart.

Add an `if` statement inside your `while` loop.

It should check if `BTN_A` was pressed.

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

Be sure to indent the `if` at the same level as the `sleep()`.

- The `if` block will have a *second* level of indentation.

CodeTrek:

```
from codex import *
from time import sleep

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(1)

    display.show(pics.HEART_SMALL)
    sleep(1)

    # If BTN_A pressed exit the loop
    if buttons.was_pressed(BTN_A):
```

Use an `if` statement inside the `while` loop.

- Check if `BTN_A` was pressed!!

```
        break
```

The `break` will exit the `loop`.

- Make sure you indent again after the `if`!

Hint:

- **Check your indentation levels**

  Refer to the CodeTrek...

  # Mind The Gap!

Goals:

- Add an `if` statement with a `break` inside your `while` loop.
- Use the `buttons.was_pressed()` function to check `BTN_A`.

Tools Found: Break and Continue, CodeX Buttons, Indentation

Solution:

```
from codex import *
from time import sleep

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(1)

    display.show(pics.HEART_SMALL)
    sleep(1)

    # If BTN_A pressed exit the loop
    if buttons.was_pressed(BTN_A):
        break
```

***Objective 7* - Explore the Beat**

Now your CodeX is *interactive!*

And your coding skills are growing.

- In the previous Objective you worked with two concepts: **input** and **branching**.
- That enabled your code to do something *different* when a button was pressed.

Review Concepts

Branching with an `if` statement:

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

CodeX Button input:

> The `buttons.was_pressed(BTN_A)` 🔧function will return `True` if **Button A** on the CodeX was pressed since the last time the function was called.

Now *Step Into Your Code!*

> **Experiment** with this code until you *really* understand how it works.
>
> **Stop** the code, then *use* the **Step Over** *button* to run it *one line at a time*.
>
> Press **Button A** while the program is paused on a line, then **Step** and watch what happens next time `buttons.was_pressed(BTN_A)` runs.

- See how the computer follows a different *branch* of the code based on the `if`?

CodeTrek:

```
from codex import *
from time import sleep

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(1)

    display.show(pics.HEART_SMALL)
    sleep(1)

    # If BTN_A pressed exit the loop
    if buttons.was_pressed(BTN_A):
        break
```

Goal:

- Use the debugger **Step Over** button to watch the branching in action!
  - You will need to hit the debug button first.
  - You must step at least 8 times!

Tools Found: Branching, Functions

Solution:

```
from codex import *
from time import sleep

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(1)

    display.show(pics.HEART_SMALL)
```

```
10      sleep(1)
11
12      # If BTN_A pressed exit the loop
13      if buttons.was_pressed(BTN_A):
14          break
```

Quiz 1 - Break-fast Time

Question 1: What happens if you press button 'A' when stepping?

```
while True:
    if buttons.was_pressed(BTN_A):
        break
```

✓ Next `buttons.was_pressed(BTN_A)` will be True

✗ Next `buttons.was_pressed(BTN_A)` will be False

✗ Buttons are ignored when stepping and paused

Question 2: What does the `break` statement do?

✓ Breaks out of a loop.

✗ Crashes the program.

✗ Causes the code to stop.

✗ Jumps over the next line of code.

Objective 8 - Half a Sleep

Are you feeling the excitement?!

Without a doubt, the CodeX's heart should be *racing* by now - **there's a new coder on the scene!**

*...but it's going to take **more** coding on your part to make its heart beat faster.*

What controls the speed in our code so far?

```
sleep(1)
```

The number controlling the speed is `1`

- That's a 1 second 🔧sleep.
- To beat twice as fast, you could cut the delay in half.
  - But how do you do that?

It turns out that the `sleep()` function can take a 🔧float parameter.

CONCEPT: Float Type

You have previously learned about a few different 🔧types.

- You learned about 🔧int, 🔧string, and 🔧bool.

There is another type called 🔧float. A **float** is a *real* number with a decimal point so it can hold a fraction.

This is an example of a float variable: `pi = 3.14159`

Ok so how do I cut the delay in half?

You could use:

```
sleep(0.5)
```

CodeTrek:

```
from codex import *
from time import sleep

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    # TODO: Add a sleep with 0.5

    display.show(pics.HEART_SMALL)
    # TODO: Add a sleep with 0.5

    # If BTN_A pressed exit the loop
    if buttons.was_pressed(BTN_A):
        break
```

Goal:

- Use a `sleep()` with `0.5` as the parameter to speed up your heartbeat.

Tools Found: Timing, float, Data Types, int, str, bool

Solution:

```
from codex import *
from time import sleep

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(0.5)

    display.show(pics.HEART_SMALL)
    sleep(0.5)

    # If BTN_A pressed exit the loop
    if buttons.was_pressed(BTN_A):
        break
```

***Objective 9* - Variable Speed Control**

You learned how to *speed up* the beat.

But you want to be able to change the speed with a button...

> You are going to need a 🔧variable!!

Concept Review: Variables

> To define a 🔧variable:
>
> - choose a name like `delay`
> - assign a value to it like `delay = 1.0`
> - use `delay` in your code just like any other value!

Notice in the code snippet below you *declare a variable* **delay** and use it instead of the literal `1.0`

```
delay = 1.0
sleep(delay)
```

CodeTrek:

```
from codex import *
from time import sleep

# Variable to control beat speed
# TODO: Create your delay variable here
```

Create the `delay` variable:

- `delay = 1.0`

```


# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(delay)
```

Change your `sleep()` to use the new `delay` variable.

```


    display.show(pics.HEART_SMALL)
    # TODO: Use delay in the sleep function
```

Add the second `sleep()` for `delay`.

```


    # If BTN_A pressed exit the loop
    if buttons.was_pressed(BTN_A):
        break
```

Goals:

- Create a variable called `delay` and set it to `1.0`.
- Replace every `sleep(0.5)` in your code with `sleep(delay)`.

Tools Found: Variables

Solution:

```
from codex import *
from time import sleep

# Variable to control beat speed
delay = 1.0

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(delay)

    display.show(pics.HEART_SMALL)
    sleep(delay)

    # If BTN_A pressed exit the loop
    if buttons.was_pressed(BTN_A):
        break
```

Objective 10 **- Brake! Not "break"...**

Your heartbeat speed is easy to change

But only by modifying the **code**

- ...and the CodeX won't *always* be connected to your PC.

- You need to change the heartbeat **while it's running** - *unplugged!*

Can you make the speed change using buttons A and B?

Check out the code snippet below (but *don't* type it in yet!)

```
if buttons.was_pressed(BTN_A):
    delay = delay + 0.2
```

Rather than `break` when `BTN_A` is pressed, the above code:

1. Adds `0.2` to the original `delay` value. ⇒ $1.0 + 0.2 = 1.2$
2. Stores this new value `1.2` back in `delay`

- So every time `BTN_A` is pressed `0.2` seconds is added to `delay`.

Ready to try it?

Your first goal is to *slow* down the heartbeat with **Button-A**.

Test Your Code

Note: The button is only checked *once* per loop, so faster clicks are ignored...

CodeTrek:

```
from codex import *
from time import sleep

# Variable to control beat speed
delay = 1.0

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(delay)

    display.show(pics.HEART_SMALL)
    sleep(delay)

    # If BTN_A pressed go slower
    if buttons.was_pressed(BTN_A):
        # TODO: slow down your beat
```

You can slow the HEART beat by increasing the delay:

```
delay = delay + 0.2
```

Goal:

- Remove the `break` and instead use `delay = delay + 0.2` inside your `if` statement.

Solution:

```
from codex import *
from time import sleep

# Variable to control beat speed
delay = 1.0

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(delay)

    display.show(pics.HEART_SMALL)
    sleep(delay)

    # If BTN_A pressed go slower
```

```
16      if buttons.was_pressed(BTN_A):
17          delay = delay + 0.2
```

Objective 11 **- Variable Speed Heart**

Now *increase* your heart-rate using Button-B

Be sure to *subtract* from `delay` when `BTN_B` is pressed!

Try running your code

Watch the CodeX get *stoked* when you press `BTN_B` a few times, and *calm it back down* with `BTN_A`!

🚨 🚨 Warning!! 🚨 🚨

Whoa! What's up with the 🔧error when the speed gets fast?

When your `delay` variable gets below `0` your program will error!!

- The `sleep()` function can only take positive `+` values.
- Negative `-` values will cause an error.
- Try it out!!

Can you make the heart beat FASTER and SLOWER?

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Variable to control beat speed
5  delay = 1.0
6
7  # Keep displaying beats forever
8  while True:
9      display.show(pics.HEART)
10     sleep(delay)
11
12     display.show(pics.HEART_SMALL)
13     sleep(delay)
14
15     # If BTN_A pressed go slower
16     if buttons.was_pressed(BTN_A):
17         delay = delay + 0.2
18
19     # If BTN_B pressed go faster
20     # TODO: Add a second if for BTN_B
```

Use an `if` statement here to check if `BTN_B` was pressed.

```
21     # TODO: Reduce the delay time
```

Make the heart beat faster with `BTN_B`:

```
delay = delay - 0.2
```

Goals:

- Add a second `if` statement to check if `BTN_B` was pressed.
- Use this code `delay = delay - 0.2` if `BTN_B` was pressed.
- Press `BTN_B` enough to bring your `delay` less than `0.0`.
    - Your program should throw an error.

Tools Found: Exception

Solution:

```
from codex import *
from time import sleep

# Variable to control beat speed
delay = 1.0

# Keep displaying beats forever
while True:
    display.show(pics.HEART)
    sleep(delay)

    display.show(pics.HEART_SMALL)
    sleep(delay)

    # If BTN_A pressed go slower
    if buttons.was_pressed(BTN_A):
        delay = delay + 0.2

    # If BTN_B pressed go faster
    if buttons.was_pressed(BTN_B):
        delay = delay - 0.2
```

Quiz 2 - Heartfelt Recap

Question 1: Why does the heartbeat blink faster when you subtract time?

✓ A smaller delay in each loop cycle makes a faster blink rate.

✗ Negative numbers are always faster than positive ones.

✗ Smaller hearts beat faster than larger ones.

Question 2: Why does your program create an error message when you keep pressing B?

✓ The `delay` variable goes below zero, and `sleep()` can't handle negative numbers.

✗ Too small a delay creates a time vortex.

✗ There is a "divide by zero" error in the `sleep()` function.

✗ The display can't run that fast.

Mission 6 Complete

Clicking buttons to make the speed faster and slower?

That code's EVERYWHERE!

- Lighting Dimmers
- Game Controllers
- Microwave Ovens
- Vehicle Cruise Controls

Your code could become an excellent Visual Metronome that could be used to set the tempo for a band!

...Imagine what else you might create!

Mission 7 - Personal Billboard

In this project you'll use the CodeX display and buttons to make a *billboard* that shows others how you're feeling, a fun picture, or a short message.

You code will be able to:

- Show images that match your current mood.
- Display **text** messages.
- Select between different images **and** messages while the code is running.

On battery power, you could make the CodeX into a *wearable* electronic **badge** or a **portable sign** for a wall or desk!

Project Goals:

- Program the 🔧CodeX Buttons to select from a series of images to show.
- Make it easy to add lots more images.
- Add the ability to mix **Text** messages with image selection.

Ready to get started?

Objective 1 - Image Selector

As you've seen already, the CodeX has lots of built-in pics like `pics.HAPPY` and `pics.HEART`.

Your first task is to make an image display that lets users select an **emotion** to match their mood by pressing `BTN_L` and `BTN_R` to cycle through their choices.

Can you use the coding ingredients from the last project to do this?

Go ahead an type in the code from the *CodeTrek.*

... remember, typing the `# comments` is optional!

Run your code!

You'll need to press button **L** or **R** to see the first image!

Make sure your display shows both `HAPPY` and `SAD` images.

CodeTrek:

```
1  from codex import *
2
3  # Loop forever - keep reading buttons and showing images!
4  while True:
5      if buttons.was_pressed(BTN_L):
6          display.show(pics.HAPPY)
7
8      # TODO: Show pics.SAD if BTN_R pressed
```

Use the example for `BTN_L` above.

- Use an `if` statement.
- Make sure you indent properly.
- `display.show(pics.SAD)`

Goals:

- Create a **new** file named `Billboard`.
- Use the `buttons.was_pressed()` function.

- Add the code to check if `BTN_R` was pressed.

Solution:

```
from codex import *

# Loop forever - keep reading buttons and showing images!
while True:
    if buttons.was_pressed(BTN_L):
        display.show(pics.HAPPY)

    if buttons.was_pressed(BTN_R):
        display.show(pics.SAD)
```

***Objective 2* - Select More Images**

Give me *more pics!*

To make the buttons scroll through **more than two** choices, you must keep track of the **choice** in your code.

You could use a **number** to track which choice should be displayed, like this:

> A number like this is called an ***index***. *Imagine using your "index finger" to point to each choice.*

How to display an image based on a number choice?

Try using an `if` statement for each image!

```
choice = 0

while True:
    if choice == 0:
        display.show(pics.HAPPY)
    if choice == 1:
        display.show(pics.SAD)
```

Follow this pattern to add TWO more Images so you have FOUR choices in all!

CONCEPT: double equals sign

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

- A "single equal" would mean 🔧assignment.
  - Like assigning `choice = 0` at the top of the program.
- A "double equal" is a 🔧comparison operator, just like `>` and friends.

Your first step is to make `BTN_R` go to the *next* pic `choice + 1`.

See the **CodeTrek** for details. Update your `while` loop!

CodeTrek:

```
from codex import *

choice = 0
```

Create a variable named `choice` and start it at `0`.

```


while True:
    if choice == 0:
        display.show(pics.HAPPY)
```

`pics.HAPPY` will show automatically because `choice` starts at `0`.

- Be sure to use == double equal sign for comparison

```
    if choice == 1:
        display.show(pics.SAD)
    # TODO: Add choice == 2 for HEART
```

Show `pics.HEART` if choice is `2`.

```
    # TODO: Add choice == 3 for ASLEEP
```

Show `pics.ASLEEP` if choice is `3`.

```


    if buttons.was_pressed(BTN_R):
        choice = choice + 1
```

Add `1` to `choice` every time `BTN_R` was pressed.

What will happen when you press it the fourth time?

Goals:

- Create a variable named `choice` that is set to `0` initially.
- Use `if choice == 2:` in your code
- Use `if choice == 3:` in your code

Tools Found: Assignment, Comparison Operators, undefined

Solution:

```
from codex import *

choice = 0

while True:
    if choice == 0:
        display.show(pics.HAPPY)
    if choice == 1:
        display.show(pics.SAD)
    if choice == 2:
        display.show(pics.HEART)
    if choice == 3:
        display.show(pics.ASLEEP)

    if buttons.was_pressed(BTN_R):
        choice = choice + 1
```

Objective 3 **- Scroll Both Directions**

Can you add code to scroll back when `BTN_L` was pressed?

Using what you've already learned, use `if buttons.was_pressed(BTN_L)` to **subtract 1** from choice.

```
if buttons.was_pressed(BTN_L):
    choice = choice - 1
```

Let me show you another awesome capability of the CodeSpace debugger: **Viewing your variables.**

Watch your variables

Try **stepping through the code.** Press button **L** or **R** while the code is waiting somewhere in the while loop, and watch the value of `choice` change as you step through the next pass of the loop.

CodeTrek:

```
from codex import *

choice = 0

while True:
    if choice == 0:
        display.show(pics.HAPPY)
    if choice == 1:
        display.show(pics.SAD)
    if choice == 2:
        display.show(pics.HEART)
    if choice == 3:
        display.show(pics.ASLEEP)

    if buttons.was_pressed(BTN_R):
        choice = choice + 1
    # TODO: if BTN_L subtract -1 from choice
```

You can add a check for `if BTN_L` was pressed here.

If button L was pressed:

```
choice = choice - 1
```

Watch your 🔧indentation!

Goals:

- Add code to scroll back when **BTN_L** is pressed

  You will need `choice = choice - 1`

- Use the debugger **Step In** button to show the different moods.
  - You will need to hit the debug button again first.
  - You must step at least 5 times!
- Open the **Console** and watch the debug variables panel.

  Hint: Use the ☰ button.

Tools Found: Indentation

Solution:

```
from codex import *

choice = 0

while True:
    if choice == 0:
        display.show(pics.HAPPY)
    if choice == 1:
        display.show(pics.SAD)
    if choice == 2:
        display.show(pics.HEART)
    if choice == 3:
        display.show(pics.ASLEEP)

    if buttons.was_pressed(BTN_R):
        choice = choice + 1
```

```
17      if buttons.was_pressed(BTN_L):
18          choice = choice - 1
```

Quiz 1 - Billboard Checkpoint

Question 1: What happens when you press `BTN_L` on the CodeX while paused on a line of code in the debugger?

- ✓ The `buttons.was_pressed(BTN_L)` is `True` on the next step.
- ✗ The button press is lost.
- ✗ The program advances to the next line of code.

Question 2: What does your "scroll both directions" program do when you keep pressing button 'R' after `pics.ASLEEP` is shown?

- ✓ The `choice` variable goes to 4 and keeps counting up.
- ✗ The `choice` variable stops at 3 which is the number of the last image.
- ✗ The `choice` variable starts over at zero.

Question 3: What does the double-equals sign mean in `if choice == 0` ?

- ✓ Compares `choice` to zero, branching when `choice` is zero.
- ✗ Assigns the variable `choice` a value of zero.
- ✗ Selects a choice of either the symbol `==` or `0`.

Objective 4 - Wrap Around

As you've probably noticed, if you keep pressing buttons, `choice` can go *past the ends* of your image list.

That might confuse users!

Can you improve the program, and avoid this problem?

Make the `choice` variable *wrap back around* to the first image (`choice = 0`) and keep advancing from there.

- Write the code to prevent `choice` going above 3.
- **...also** add code to keep it from going *below 0!*

To keep `choice` from going above `3` you can use the **greater than** `>` 🔧comparison operator. This operator checks if one number is **greater than** another!

Here is some wrap around code when adding +1.

```
choice = choice + 1
if choice > 3:
    choice = 0
```

*How would you check for `choice` **less than** `0`?*

CodeTrek:

```
 1  from codex import *
 2
 3  choice = 0
 4
 5  while True:
 6      if choice == 0:
 7          display.show(pics.HAPPY)
 8      if choice == 1:
 9          display.show(pics.SAD)
10      if choice == 2:
```

```
11          display.show(pics.HEART)
12      if choice == 3:
13          display.show(pics.ASLEEP)
14
15      if buttons.was_pressed(BTN_R):
16          choice = choice + 1
17          if choice > 3:
18              choice = 0
```

Start by wrapping around from `3` to `0` if `BTN_R` is pressed again.

```
19
20      if buttons.was_pressed(BTN_L):
21          choice = choice - 1
22          # TODO: Check if less than 0
```

Check `if` the variable `choice` is `<` less than `0`.

```
23              # TODO: Wrap around to 3
```

Wrapping around means going from `0` to `3` when subtracting:

```
choice = 3
```

Goals:

- Use `if choice > 3` when wrapping around on the + 1.
- Use `if choice < 0` when wrapping around on the - 1.

Tools Found: Comparison Operators

Solution:

```
1  from codex import *
2
3  choice = 0
4
5  while True:
6      if choice == 0:
7          display.show(pics.HAPPY)
8      if choice == 1:
9          display.show(pics.SAD)
10     if choice == 2:
11         display.show(pics.HEART)
12     if choice == 3:
13         display.show(pics.ASLEEP)
14
15     if buttons.was_pressed(BTN_R):
16         choice = choice + 1
17         if choice > 3:
18             choice = 0
19
20     if buttons.was_pressed(BTN_L):
21         choice = choice - 1
22         if choice < 0:
23             choice = 3
```

***Objective 5* - Image List**

So many Images to display

...but it takes *so much code* to add more!

It takes *two lines of code* for each image you add, so your program is long and your fingers grow tired...

Wouldn't it be great if you could just make a **list** of Images like this for your code?

```
my_list = [pics.HAPPY, pics.SAD, pics.HEART]
```

Well, in fact Python has a feature just for this purpose!

It's called... wait for it... a 🔧list.

CONCEPT: Lists

In Python you create a 🔧list just as shown above, using **square brackets**.

```
example_list = [item_0, item_1, item_2]
```

The **order** of each item in the list is *important!*

- Items are counted starting with **zero**.
- An item's order in the list is called its **index**.
- You can get any item from a list using its index!

```
first = example_list[0]  # Get item_0
```

To access one of the items in a list, use brackets like:

```
# Assign the first item in the list to my_image
my_image = my_list[0]
```

Lists have other cool features, including the ability to get the number of items:

```
# Get the length of the list
num_choices = len(my_list)
```

Can you think how you might improve your code using a list?

CodeTrek:

```
1  from codex import *
2
3  choice = 0
4
5  my_list = [pics.HAPPY, pics.SAD, pics.HEART, pics.ASLEEP]
```

This is your new 🔧list!

- Make sure the list has exactly **4** items in it!

```
6
7  while True:
8      my_image = my_list[choice]
```

You can choose the image from your 🔧list by its **index**.

`choice` is the same as **index** here!

```
9      display.show(my_image)
```

You got rid of all those `if choice == X:` statements!

- That is **much** simpler!

Now all that's left is showing the image.

```
10
11     if buttons.was_pressed(BTN_R):
12         choice = choice + 1
```

```
        if choice > 3:
            choice = 0

    if buttons.was_pressed(BTN_L):
        choice = choice - 1
        if choice < 0:
            choice = 3
```

Goals:

- Create a 🔧list variable named `my_list` and initialize it with 4 items.
- Access an **index** in `my_list` using the variable `choice`.

Tools Found: list

Solution:

```
from codex import *

choice = 0

my_list = [pics.HAPPY, pics.SAD, pics.HEART, pics.ASLEEP]

while True:
    my_image = my_list[choice]
    display.show(my_image)

    if buttons.was_pressed(BTN_R):
        choice = choice + 1
        if choice > 3:
            choice = 0

    if buttons.was_pressed(BTN_L):
        choice = choice - 1
        if choice < 0:
            choice = 3
```

***Objective 6* - No Magic Numbers!**

Use built-in *list* features to enhance your code!

Notice how the **length** of your list is *baked into* your code?

The **"magic number"** of **3** for the *length* of your list is making your code harder to read and maintain.

- If you add Images to the list, you have to modify the rest of the code.
- People reading your code might not know *why* the numbers **3** and **0** are being used to compare and assign `choice`.

Make your code more *readable* and *maintainable*:

- Define a variable for `LAST_INDEX` to replace `3`. (Why is this variable 🔧upper case?)
- Use the built-in `len(my_list)` to automatically get the **length** of the list

Take a look at this code:

```
LAST_INDEX = len(my_list) - 1
```

Why do you need to subtract 1?

- Because the indexes start at 0 so the **index** of the last item is 1 less than the **length**!

Now you can add some *more* Images to your list.

- Lists make it so easy!
- Check the 🔧pic gallery for more Images!

CodeTrek:

```
from codex import *

choice = 0

my_list = [
    pics.HAPPY,
    pics.SAD,
    pics.HEART,
    pics.ASLEEP,
    pics.SURPRISED
]
```

When the list gets long you can define it on multiple lines!

- Add another image to your 🔧list.

```

# Define the last index
LAST_INDEX = len(my_list) - 1
```

Add the `LAST_INDEX` variable!

- Remember to subtract 1!!!

```

while True:
    my_image = my_list[choice]
    display.show(my_image)

    if buttons.was_pressed(BTN_R):
        choice = choice + 1
        if choice > LAST_INDEX:
```

Instead of the **magic** `3` you can use `LAST_INDEX`!

```
            choice = 0

    if buttons.was_pressed(BTN_L):
        choice = choice - 1
        if choice < 0:
            choice = LAST_INDEX
```

Don't forget to use `LAST_INDEX` here too!

Goals:

- Add a fifth image to `my_list`!
- Create a `LAST_INDEX` variable and default it to `len(my_list) - 1`.

Tools Found: Constants, CodeX Image Pics, list

Solution:

```
from codex import *

choice = 0

my_list = [
    pics.HAPPY,
    pics.SAD,
    pics.HEART,
    pics.ASLEEP,
    pics.SURPRISED
```

```
]

# Define the last index
LAST_INDEX = len(my_list) - 1

while True:
    my_image = my_list[choice]
    display.show(my_image)

    if buttons.was_pressed(BTN_R):
        choice = choice + 1
        if choice > LAST_INDEX:
            choice = 0

    if buttons.was_pressed(BTN_L):
        choice = choice - 1
        if choice < 0:
            choice = LAST_INDEX
```

Quiz 2 - List Len

Question 1: Why do you have to **subtract 1** from `len(my_list)` to get `LAST_INDEX`?

✓ List indexes start at 0 so the index of the last item is `len(my_list) - 1`.

✗ Because the program needs to count up as well as down in the list.

✗ List indexes are negative numbers so `len(my_list) - 1` is required.

Objective 7 - Text Time!

Images are expressive

...but *TEXT* can say much more!

You can define any message you want by putting it in quotes:

```
my_message = "Hi there!"
```

The computer doesn't care what's between the quotation marks - it's just a 🔧*string of characters.*

Changing your program to display text messages is very simple:

- `display.show()` doesn't just accept Images - it can also handle **string** 🔧types.

Modify your program to add a *personalized message* string to the list.

Example:

```
my_list = [
    "Ahoy",
    pics.HAPPY,
    pics.SAD,
    pics.HEART,
    pics.ASLEEP,
    pics.SURPRISED
]
```

Test your program and make sure it shows Images and text!!

CodeTrek:

```
from codex import *

choice = 0

my_list = [
    # TODO: Add a personalized message here
```

Add a personalized 🔧string inside your list like `"Smile!"`.

```
    pics.HAPPY,
    pics.SAD,
    pics.HEART,
    pics.ASLEEP,
    pics.SURPRISED
]

# Define the last index
LAST_INDEX = len(my_list) - 1

while True:
    my_image = my_list[choice]
    display.show(my_image)

    if buttons.was_pressed(BTN_R):
        choice = choice + 1
        if choice > LAST_INDEX:
            choice = 0

    if buttons.was_pressed(BTN_L):
        choice = choice - 1
        if choice < 0:
            choice = LAST_INDEX
```

Goal:

- Add a 🔧string message to `my_list`.

Tools Found: str, Data Types

Solution:

```
from codex import *

choice = 0

my_list = [
    "Smile!",
    pics.HAPPY,
    pics.SAD,
    pics.HEART,
    pics.ASLEEP,
    pics.SURPRISED
]

# Define the last index
LAST_INDEX = len(my_list) - 1

while True:
    my_image = my_list[choice]
    display.show(my_image)

    if buttons.was_pressed(BTN_R):
        choice = choice + 1
        if choice > LAST_INDEX:
            choice = 0

    if buttons.was_pressed(BTN_L):
        choice = choice - 1
        if choice < 0:
            choice = LAST_INDEX
```

Objective 8 **- Green With Envy**

Color My World

What if you're neither `HAPPY` nor `SAD`?

...and text just isn't describing you?

Sometimes you just need a *color.*

Maybe you are `GREEN` with envy!

Wouldn't it be cool to fill the display with a color?

Add the color `GREEN` to `my_list` and try to show it!

🚨 🚨 Warning!! 🚨 🚨

`GREEN` may not work properly in `display.show()`!!

`display.show()` only works with Bitmap and **str** 🔧types.

CodeTrek:

```
1  from codex import *
2
3  choice = 0
4
5  my_list = [
6      # TODO: Add the color GREEN
       Add the color GREEN to my_list.
       • You want it to show up on the pixels!
7      "Smile!",
8      pics.HAPPY,
9      pics.SAD,
10     pics.HEART,
11     pics.ASLEEP,
12     pics.SURPRISED
13 ]
14
15 # Define the last index
16 LAST_INDEX = len(my_list) - 1
17
18 while True:
19     my_image = my_list[choice]
20     display.show(my_image)
21
22     if buttons.was_pressed(BTN_R):
23         choice = choice + 1
24         if choice > LAST_INDEX:
25             choice = 0
26
27     if buttons.was_pressed(BTN_L):
28         choice = choice - 1
29         if choice < 0:
30             choice = LAST_INDEX
```

Goal:

- Try to `display.show()` the color `GREEN`.
  - Beware... you may get an 🔧error!

Tools Found: Data Types, Exception

Solution:

```
1  from codex import *
2
```

```
choice = 0

my_list = [
    GREEN,
    "Smile!",
    pics.HAPPY,
    pics.SAD,
    pics.HEART,
    pics.ASLEEP,
    pics.SURPRISED
]

# Define the last index
LAST_INDEX = len(my_list) - 1

while True:
    my_image = my_list[choice]
    display.show(my_image)

    if buttons.was_pressed(BTN_R):
        choice = choice + 1
        if choice > LAST_INDEX:
            choice = 0

    if buttons.was_pressed(BTN_L):
        choice = choice - 1
        if choice < 0:
            choice = LAST_INDEX
```

***Objective 9* - Fill 'er Up**

What's in a Color?

Colors in the `codex` library are actually tuples!

- A **tuple** is like a list that can't be changed.
- CodeX color tuples have three integer values: `(red, green, blue)`

  `GREEN` is defined as `(0, 255, 0)`

You can fill the whole screen with a color using this new display function:

```
# Fill the display with given color
display.fill(COLOR)
```

Fixing Your BUG!

You've already seen a similar error using `display.show()`

Remember, to show an int number you had to convert it to a string with `str()`.

- But for your **Billboard** you need to detect a tuple and treat it as a *color.*
- *So, what can you do?*

You need to check the type of the item *before* showing it!

CONCEPT: type checking

The built-in function `type()` is used to read the *type* of object a variable refers to.

- Each data type has a name such as:
  - **str** for strings
  - **int** for integers
  - **tuple** for tuples
- The `type(object)` function returns the the object's type.

Finally, you will need an `if` and `else` statement.

`else` is a 🔧branching statement that comes after an `if`.

- It means if the `if` condition was `False` then run this block!

CodeTrek:

```
1  from codex import *
2
3  choice = 0
4
5  my_list = [
6      GREEN,
7      "Ahoy",
8      pics.HAPPY,
9      pics.SAD,
10     pics.HEART,
11     pics.ASLEEP,
12     pics.SURPRISED
13 ]
14
15 # Define the last index
16 LAST_INDEX = len(my_list) - 1
17
18 while True:
19     my_image = my_list[choice]
20
21     # If the type is a color
22     if type(my_image) == tuple:
```

Check whether the 🔧list item is a color.

- Colors in `codex` are just 🔧tuples!

```
23         display.fill(my_image)
```

`display.fill()` will fill the display with a color!

```
# Fill the screen blue
display.fill(BLUE)
```

```
24     else:
```

An `else:` statement must follow an `if condition:`.

- It also needs to be indented at the same level!

```
25         # TODO: Use display.show()
```

Show Images or strings with `display.show()`

- Use TAB to *move the code you already have* beneath this `else`!!

```
26
27     if buttons.was_pressed(BTN_R):
28         choice = choice + 1
29         if choice > LAST_INDEX:
30             choice = 0
31
32     if buttons.was_pressed(BTN_L):
33         choice = choice - 1
34         if choice < 0:
35             choice = LAST_INDEX
```

Goals:

- Use the `type()` function to check for a color (tuple).
- Use the `display.fill()` function if the item is a color!

Tools Found: tuple, list, int, Display, str, Data Types, Branching

Solution:

```
from codex import *

choice = 0

my_list = [
    "Ahoy",
    pics.HAPPY,
    GREEN,
    pics.SAD,
    pics.HEART,
    pics.ASLEEP,
    pics.SURPRISED
]

# Define the last index
LAST_INDEX = len(my_list) - 1

while True:
    my_image = my_list[choice]

    # If the type is a color
    if type(my_image) == tuple:
        display.fill(my_image)
    else:
        display.show(my_image)

    if buttons.was_pressed(BTN_R):
        choice = choice + 1
        if choice > LAST_INDEX:
            choice = 0

    if buttons.was_pressed(BTN_L):
        choice = choice - 1
        if choice < 0:
            choice = LAST_INDEX
```

Mission 7 Complete

Congratulations! There was a LOT to learn in this project!

Just a few of the new *Tools* you used:

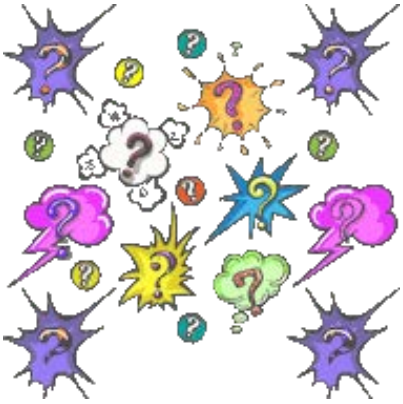
- Managing lists of information.
- Handling the "list index overflow/wrap" case.
- Inspecting different data types.
- Viewing variables in the **Debug Panel.**

And you have built real-world code!

- A *scrolling menu system* like the one you built is found in a lot of devices and applications, from medical equipment to toys.

Mission 8 - Answer Bot

In this project you will create a random answer generator.



Instead of selecting messages yourself, like in the previous project, this time you'll have the computer decide for you!

You'll never have to waste your time on all those unimportant, random decisions of life again.

→ Just press the button and let your **Answer Bot** decide :-)

Project Goals:

- Program the CodeX to choose and display a random number when a button is pressed.
- Change the program to display a random text message from a list of possible answers.

Ready to get started?

Objective 1 - Display a Number

To begin, you will pick a fixed number and show it on the display.

⚠ Heads up! ⚠

The code in the CodeTrek has an error which you will correct later.

Can you spot the error before you run it?

You *may* remember seeing this in a previous project!

Go ahead and run your code!

CodeTrek:

```
from codex import *

number = 1
display.show(number)
```

Goals:

- Create a **new** file named **Answer_Bot**.
- Run the code from the CodeTrek to cause an error.

Tools Found: Exception

Solution:

```
from codex import *

number = 1
display.show(number)
```

Objective 2 - Fix it Up

It's time to fix the code and make the number show up on the display!

When you encountered this error before, the fix was to convert the integer type into a string type.

- The built-in str function is made for that!
- You could certainly fix the error with `str(number)`...

But you also now know a different text-message function: `display.print()`

- I heard a rumor that it does the `str()` conversion *automatically!*

Give it a try with `display.print(number)` instead

Try some different numbers just for fun!

CodeTrek:

```
from codex import *

number = 1
# TODO: Show number on the display
```

Use the `display.print()` function.

Goal:

- Change your code to use `display.print(number)`

Tools Found: int, str

Solution:

```
from codex import *

number = 1
display.print(number)
```

Objective 3 **- Randomize!**

It's time to get RANDOM in here!

You'll be using the **random** Python module, so look for a new 🔧import statement.

CONCEPT: *random*

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

```
import random
# Get random number from 0 to 9
x = random.randrange(10)
```

- **Notice:** Just like 🔧lists count from **0**, so does `randrange(N)`
- So a 🔧range of **10** numbers gives values `0` through `9`.

Modify your code to display a **random number** rather than your fixed number from the previous step.

You should see a *random* number when you run the program.

Run the program a few times to make sure you see different numbers.

Sometimes you may see the same number repeat, but that's all part of the randomness!

CodeTrek:

```
1  from codex import *
2  import random
3
4  number = # TODO: Random number 0 to 9
```

To get a random number from 0 to 9 use:

```
random.randrange(10)
```

```
5  display.print(number, scale=3)
```

Try a different `scale` if you like!

- The `display.print()` function has a keyword argument that lets you change the size of your text.

Goals:

- Use the `randrange()` function with `10` as the argument.
  - This will give you an int from `0` to `9`!
- **Embiggen It!**

  The `display.print()` function has another awesome feature.
  - You can *scale up* the size of the text like so:

    ```
    display.print(number, scale=3)
    ```

  Go ahead and try bigger text!

Tools Found: import, Random Numbers, list, Ranges, int, Keyword and Positional Arguments

Solution:

```
from codex import *
import random

number = random.randrange(10)
display.print(number, scale=3)
```

Objective 4 **- Mix Things Up**

Random Quick Mix!

Re-starting the program every time you want a new random number is *too slow!*

You can fix that!

Modify your code to randomly select a number from 0 to 9 each time **Button A** is pressed.

Make sure to indent your code inside a loop, checking for button presses.

- Refresh your memory on CodeX Buttons and loops if needed!

Run your code and...

Press **Button A** a few times.

A random number should show up each time.

CodeTrek:

```
1  from codex import *
2  import random
```

```
3
4  # TODO: Add a while True loop
```

Add a `while True:` loop here to make the program keep requesting random numbers.

```
5      # TODO: If BTN_A was pressed
```

Check if `BTN_A` was pressed.

- `if buttons.was_pressed(BTN_A):`

```
6          number = random.randrange(10)
7          display.print(number, scale=3)
```

Goals:

- Use a `while True` loop in your code.
- Check `if` **button A** was pressed.

Tools Found: Indentation, CodeX Buttons, Loops

Solution:

```
1  from codex import *
2  import random
3
4  while True:
5      if buttons.was_pressed(BTN_A):
6          number = random.randrange(10)
7          display.print(number, scale=3)
8
```

Objective 5 **- Robot Opinion**

Time to give the Robot an *opinion!*

Now that you have made an *amazing* **random number display**, it's time to make the CodeX answer a question like: "What is your favorite food?" ...and display a *random* answer.

This is a perfect place for a 🔧list!

- Use a *random number* as an **index** into a *list* of **Answers.**

Personalize Me!

This is *your* **Answer Bot**, so you can make it answer a *different* question:

- Favorite sports team
- Best dance moves
- Magic 8 ball answers...
- *You decide!*

Now, get your list of *Answers* ready and code this thing!

Here is an example:

```
# What's for lunch?
answers = [
    "Pizza",
    "Burger",
    "Salad"
]
```

Run your code and press Button A a few times to get some answers.

CodeTrek:

```
from codex import *
import random

answers = [
    # TODO: Fill in with your own answers
```

Add your own answers here.

- Each string must be separated by a comma ,

Here is an example:

```
answers = [
    "Pizza",
    "Burger",
    "Salad"
]
```

```
]

while True:
    if buttons.was_pressed(BTN_A):
        count = # TODO: count is the number of answers
```

The `count` is the `len()` of the `answers` list.

- `len(answers)`

```
        index = random.randrange(count)
```

You need a random number from 0 to the number of `answers` minus 1.

```
        display.print(answers[index])
```

Print the random answer from the `answers` list.

- Your `index` is the random number!!

Goals:

- Create a variable named `answers` that is defined as a list.
- Use the `len()` function to get the length of the `answers` list.

Tools Found: list, str

Solution:

```
from codex import *
import random

answers = [
    "Pizza ",
    "Burger",
    "Salad "
]

while True:
    if buttons.was_pressed(BTN_A):
        count = len(answers)
```

```
13          index = random.randrange(count)
14          display.print(answers[index])
```

Quiz 1 - Get Some Answers

Question 1: Why does `range(10)` only go up to **9**??

✓ Ranges start at 0 (zero), and there are 10 values from 0-9.

✗ The `range(10)` also includes -1 so the max is 9.

✗ The variable itself consumes one integer so there are only 9 values in `range(10)`.

Question 2: What is the `count` variable doing for you in this program?

```
answers = ["0", "1", "2"]

while True:
    if buttons.was_pressed(BTN_A):
        count = len(answers)
        index = random.randrange(count)
```

✓ The `count` variable stores len(answers) to give to the 'randrange' function.

✗ The `count` variable automatically scans the list and counts the number of items.

✗ The `count` is a beloved character in educational television.

Objective 6 - Wait for Answer

Let's fancy up the *Answer Bot*

Flashy colors while you wait!

- Make the `pixels` constantly cycle through 🔧random colors.
- But now you need a 🔧list of **colors**.
- **Good news** - you already have that list!*

Importing `from codex import *` gives you access to a lot of cool CodeX features, including the `colors` module.

- That's where RED, GREEN, BLUE, and all the other *predefined* color 🔧constants come from.
- **But also it gives you:** `COLORS_BY_NAME` **and** `COLOR_LIST`**.**

```
# Built into the 'colors' module
COLOR_LIST = [
    BLACK, BROWN, RED, ORANGE, YELLOW,
    GREEN, BLUE, PURPLE, GRAY, WHITE,
    CYAN, MAGENTA, PINK, LIGHT_GRAY, DARK_GREEN,
    DARK_BLUE,
]
```

You already know how to pick a random item from a list, *right?*

If you're unsure, let the **CodeTrek** be your guide!

CodeTrek:

```
1  from codex import *
2  import random
3  from time import sleep
```

Import `sleep` so that you can add some delays.

```
4
5  answers = [
6      "Pizza ",
7      "Burger",
8      "Salad "
9  ]
10
11 while True:
12     # Pick a random color from COLOR_LIST
13     index = random.randrange( len(COLOR_LIST) )
14     color = COLOR_LIST[index]
```

The `len()` function tells you how many colors are in COLOR_LIST.

- So that's the range of 🔧random numbers you need!

```
15
16     pixels.set(0, color)
17     # TODO: Set pixel 1 to color
18     # TODO: Set pixel 2 to color
19     # TODO: Set pixel 3 to color
```

Set all 4 pixels to the randomly chosen color.

```
20
21     if buttons.was_pressed(BTN_A):
22         count = len(answers)
23         index = random.randrange(count)
24         display.print(answers[index])
25
26     sleep(0.1)
```

This delay is important because it prevents the `pixels` color's from blending together.

- This pause allows your eyes to see the color.

Goals:

- Create a variable named `color` and assign to it a random color from `COLOR_LIST`
- Add a 🔧delay so your `pixels` can display each `color` long enough for your eyes to see it.

Tools Found: Random Numbers, list, Constants, Timing

Solution:

```
1  from codex import *
2  import random
3  from time import sleep
4
5  answers = [
6      "Pizza",
7      "Burger",
8      "Salad"
9  ]
10
11 while True:
12     # Pick a random color from COLOR_LIST
13     index = random.randrange( len(COLOR_LIST) )
14     color = COLOR_LIST[index]
15
16     pixels.set(0, color)
17     pixels.set(1, color)
18     pixels.set(2, color)
19     pixels.set(3, color)
```

```
20
21      if buttons.was_pressed(BTN_A):
22          count = len(answers)
23          index = random.randrange(count)
24          display.print(answers[index])
25
26      sleep(0.1)
```

***Objective 7* - Choices, Choices**

Choices, Choices...

There are often *many* ways to achieve your goals when coding!

- Your "Answer Bot" is working great, but you can improve it.
- *And* make your code simpler and more 🔧readable at the same time!

Improvement: *Individual Random Pixels*

Make each *pixel* flash a different random color.

- You already know a way to do this... just repeat the code you already have, and choose 4 different colors each time.
- But a new Python 🔧random feature will make it even easier!

The `random.choice()` function simplifies what your code is already doing

- Behind the scenes it does *exactly* what your code was doing to pick an item from a list.

```
# Choose a random color from the list
color = random.choice(COLOR_LIST)
```

CodeTrek:

```
1  from codex import *
2  import random
3  from time import sleep
4
5  answers = [
6      "Pizza",
7      "Burger",
8      "Salad"
9  ]
10
11 while True:
12     pixels.set(0, random.choice(COLOR_LIST))
13     pixels.set(1, random.choice(COLOR_LIST))
14     pixels.set(2, random.choice(COLOR_LIST))
15     pixels.set(3, random.choice(COLOR_LIST))
```

Use `random.choice()` to choose a random color for each *pixel*.

```
16
17     if buttons.was_pressed(BTN_A):
18         display.print(random.choice(answers))
```

Simplify choosing from your `answers` list too!

```
19
20     sleep(0.1)
```

Goals:

- Use `random.choice()` to set each pixel to its own 🔧random item from `COLOR_LIST`
- Simplify your code that chooses from the `answers` list.
    - Using `random.choice(...)` would be a good *choice!*

Tools Found: Readability, Random Numbers

Solution:

```
from codex import *
import random
from time import sleep

answers = [
    "Pizza",
    "Burger",
    "Salad"
]

while True:
    pixels.set(0, random.choice(COLOR_LIST))
    pixels.set(1, random.choice(COLOR_LIST))
    pixels.set(2, random.choice(COLOR_LIST))
    pixels.set(3, random.choice(COLOR_LIST))

    if buttons.was_pressed(BTN_A):
        display.print(random.choice(answers))

    sleep(0.1)
```

Mission 8 Complete

But seriously, the *fundamentals* of this code are really important to a **lot** of applications!

🔧Random number code is crucial for:

- Secure password encryption
- Real-world simulator trainers
- Scientific statistical sampling
- Artifical Intelligence (AI) tools

Mission 9 - Game Spinner

In this project, you'll make a Game Spinner that can:

- **Choose** the next person to tell a story in a group of friends
- **Navigate** every turn in a Crazy Compass Game
- **Decide** which pizza slice to eat next
- Provide an element for a game **you create!**

Your game spinner will show a spinning arrow on the CodeX display when you press Button A or B, and then slow down and stop in one of 8 random directions.

Project Goals:

- Display an *Arrow* in a random direction
- Detect button A **or** B to trigger the *Arrow spin*
- **Animate** an Arrow *spinning around*
- Make the Arrow *gradually slow* rather than stopping abruptly

Ready to get started?

Objective 1 - Random Arrow

The CodeX has a set of "Compass Arrow" 🔧pics that are perfect for your spinner.

There's even a built-in list with **ALL** the **ARROWS**

```
# This list is ALREADY provided!
# (DO NOT TYPE THIS IN!)
pics.ALL_ARROWS = [
    pics.ARROW_N,
    pics.ARROW_NE,
    pics.ARROW_E,
    pics.ARROW_SE,
    pics.ARROW_S,
    pics.ARROW_SW,
    pics.ARROW_W,
    pics.ARROW_NW,
]
```

Displaying an ARROW from the 🔧list is simple.

There are 8 arrows, so the list *index* goes from 0 to 7:

```
display.show(pics.ALL_ARROWS[num])
```

Your Game Spinner needs to land on a *random* direction

Use the 🔧random module to choose which ARROW to display. The following sets **num** to a random number:

```
num = random.randrange(8)
```

- You could also use `random.choice()` *...it's your choice!*

CodeTrek:

```
1  from codex import *
2  import random
3
4  num = random.randrange(8)
```

Instead of using the magic number `8` here you could create a variable!

```
ARROWS_LEN = len(pics.ALL_ARROWS)
num = random.randrange(ARROWS_LEN)
```

```
5  display.show(pics.ALL_ARROWS[num])
```

Show the random arrow from the `pics.ALL_ARROWS` list.

Goals:

- Create a **new** file named `Game_Spinner`.
- Show a random arrow from the `pics.ALL_ARROWS` list.

Tools Found: CodeX Image Pics, list, Random Numbers

Solution:

```
1  from codex import *
2  import random
3
4  num = random.randrange(8)
5  display.show(pics. ALL_ARROWS[num])
```

Quiz 1 - Which Arrows

Question 1: What are the possible values of `num`?

```
import random
num = random.randrange(8)
```

✓ 0,1,2,3,4,5,6,7

✗ 1,2,3,4,5,6,7,8

✗ -3,-2,-1,0,1,2,3,4

Question 2: Which image is displayed by display.show() below?

```
pics.ALL_ARROWS = [
    pics.ARROW_N,
    pics.ARROW_NE,
    pics.ARROW_E,
    pics.ARROW_SE,
    pics.ARROW_S,
    pics.ARROW_SW,
    pics.ARROW_W,
    pics.ARROW_NW
]

display.show(pics.ALL_ARROWS[3])
```

✓ pics.ARROW_SE

✗ pics.ARROW_SW

✗ pics.ARROW_S

✗ pics.ARROW_E

Objective 2 - Click to Flick

Flick!

The classic Game Spinner has a metal arrow that you finger-flick to spin.

- Make your **CodeX** spin whenever Button **A** or **B** is pressed.

The program should run forever, so wrap your code in an **infinite loop**, checking *both* BTN_A and BTN_B.

There are two new concepts that will help with this task!

CONCEPT: *Instantaneous Button Polling*

The function `buttons.is_pressed(BTN_A)` returns `True` if button **A** is **currently** held down.

*Your code can quickly check the **state** of the 🔧CodeX Buttons with this function.*

CONCEPT: *Logical Operators*

You've seen how 🔧branching and 🔧loops control the flow of your program with `True` / `False` *decisions:*

- Functions like `buttons.is_pressed(BTN_A)` that *return* `True` or `False`.
- 🔧Comparison operations like `x > 51` (which are also `True` or `False`)

But what if you have *multiple items* to compare - like **two buttons**, testing if *either one **or** the other* is `True`?

- That's where 🔧logical operators: `and`, `or`, and `not` come into play.
- *Be sure to check out the **Toolbox** help for this topic!*

---

*It's time to **apply** these new concepts to make your spinner respond to either button!*

CodeTrek:

```
from codex import *
import random

while True:
```

Add in an infinite loop!

```
    if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
```

Check if **either** button **IS** currently pressed.

`or` lets you do either:

```
if X or Y:
```

```
        num = random.randrange(8)
        display.show(pics.ALL_ARROWS[num])
```

Double check your indentation!

Goals:

- **Keep running *forever!***

  Add an infinite `while True` loop.

- **Check for A or B to "spin"**

  Use an `or` operator inside an `if` statement.

  - On button press, CHOOSE!

Tools Found: CodeX Buttons, Branching, Loops, Comparison Operators, Logical Operators

Solution:

```
from codex import *
import random

while True:
    if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
        num = random.randrange(8)
        display.show(pics.ALL_ARROWS[num])
```

Objective 3 **- Fun Functions**

Your program does the job, but you can **improve** it with...

Realism!

Realistic spinning action would be awesome!

Next step will be to add **animation** - to make the arrow spin around before it lands on the random choice!

The shape of your program will be: *(example - don't type this in)*

```
while True:
    if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
        # animate arrow spinning around
        # ...

        # show random arrow
        # ...
```

Yikes! It may take a few lines of code to do the animation, so the above code could get messy. Too bad there aren't built-in functions like `spin_animation()` and `show_random_arrow()`.

Alas, those functions are not built-in!

But you *can* make your **own** 🔧functions!

Making new functions is like creating your *own* language!

CONCEPT: 🔧*Functions*

Here is an example function.

```
def show_random_arrow():
    num = random.randrange(8)
    display.show(pics.ALL_ARROWS[num])
```

The keyword `def` means "define function". After the def statement runs, the named function can be *called* just like a built-in function!

- Dividing code into logical functions can make it much more 🔧readable.

⚠ **Note:** 🔧Functions must be **defined** *before* they are used, so make sure to put the `def` **above** your `while` loop!

CodeTrek:

```
from codex import *
import random

def show_random_arrow():
```

Define your new function here.

- Remember `def` is used to create a new function.

```
    num = random.randrange(8)
    display.show(pics.ALL_ARROWS[num])
```

Move all code to show the random arrow inside the function!

```

while True:
    if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
        show_random_arrow()
```

Call your first function!!

- The function can be called **AFTER** it is defined!

Goals:

- Define a function named `show_random_arrow()`.
- Call your `show_random_arrow()` function!

Tools Found: Functions, Readability, Divide and Conquer

Solution:

```
from codex import *
import random

def show_random_arrow():
    num = random.randrange(8)
    display.show(pics.ALL_ARROWS[num])

while True:
    if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
        show_random_arrow()
```

***Objective 4* - Animation**

Animation

...is achieved with a rapid sequence of images.

- You've already seen that the CodeX is quite good at displaying a list of images quickly.

    Sometimes even too quickly! You have to slow it down to see all the images clearly.

To animate a **spinning arrow** you just need to cycle through all 8 positions (0-7) with a short delay between each.

```
display.show(pics.ALL_ARROWS[0])
sleep(0.1)
display.show(pics.ALL_ARROWS[1])
sleep(0.1)
# ...Wait! There has to be a better way.
```

While the above would work, there **is** a better way. A 🔧loop!

But **not** an *infinite* loop. You only need to repeat **8** times.

Study the code in the `spin_animation()` function in the CodeTrek.

Do you see how the variable `index` starts at 0 and counts up each time the loop repeats?

Experiment with your animation!

- What would happen if you changed `sleep(0.1)` to a *smaller* value?
- Could the arrow be made to spin the other direction?

CodeTrek:

```
1  from codex import *
2  # TODO: import sleep function
```

Remember it's: `from time import sleep`

```
3  import random
4
5  def spin_animation():
```

Define a new `spin_animation()` function!

```
6      index = 0
```

Create a variable called `index`.

- This variable will count up inside the `while` loop.
- It will also be used to access the **arrow image** from the list.

```
7      while index < 8:
```

The `while` loop will run 8 times.

- Each time the `index` variable will count up one.
- When `index` reaches 8 the loop will stop.

```
8          display.show(pics.ALL_ARROWS[index])
9          sleep(0.1)
10         index = index + 1
```

`index` should increase at the **END** of the loop!!!!

If `index` gets increased before selecting an image you will end up with an error.

`pics.ALL_IMAGES[8]` would error because `8` is **OUT OF RANGE**!

```
11
12 def show_random_arrow():
13     num = random.randrange(8)
14     display.show(pics.ALL_ARROWS[num])
15
16 while True:
17     if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
18         # TODO: Call the animation function
```

Make sure you call your new animation function!

- To *call* a function you must use its *name* followed by an argument list in parentheses. Even if it has NO arguments, like `spin_animation()`.

```
19         show_random_arrow()
```

Goals:

- Define a new `spin_animation()` function with no parameters.
- Call the `spin_animation()` function.
- Create a `while` loop that is **NOT** *infinite*.
  - The `while` statement must check an `index` variable.

Tools Found: Loops, Exception, Keyword and Positional Arguments

Solution:

```
from codex import *
from time import sleep
import random

def spin_animation():
    index = 0
    while index < 8:
        display.show(pics.ALL_ARROWS[index])
        sleep(0.1)
        index = index + 1

def show_random_arrow():
    num = random.randrange(8)
    display.show(pics.ALL_ARROWS[num])

while True:
    if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
        spin_animation()
        show_random_arrow()
```

Quiz 2 - Indented?

Question 1: Why is the `if` statement below **indented** beneath the `while`?

```
while True:
    if buttons.is_pressed(BTN_A):
        display.show(pics.ALL_ARROWS[0])
```

✓ So that it runs completely inside the loop.

✗ Because `if` statements have to be indented.

✗ So that the arrow is only displayed when a button is pressed.

Question 2: Which 🔧condition **stops** the loop in this code?

```
index = 0
while index < 8:
    index = index + 1
```

✓ The loop stops when `index` reaches 8.

✗ An infinite loop never stops.

✗ The loop stops when `index` reaches 0.

✗ The statement `index = index + 1` ends the loop.

Question 3: What is show_random_arrow in the code below?

```
def show_random_arrow():
    num = random.randrange(8)
    display.show(pics.ALL_ARROWS[num])
```

✓ A Function

✗ A String

✗ A Party

❌ A Loop

***Objective 5* - Style Points - Physics Part 1**

Your *animation* is nice...

But it needs more *realism*! A real *Game Spinner* starts out **fast** and **slows down** gradually before it **stops**.

> You *could* measure the weight and friction properties of a real spinner and code an exact 🔧simulation...

For now just use a *rough approximation* of real-world physics - slowing the spin animation *gradually*.

> Your first step is to make the animation spin **longer**.

CONCEPT: *Parameters and Arguments*

🔧Functions in Python can be defined with a list of 🔧parameters.

- When you call a function, you can supply values for those parameters.
- For example when you call `display.show("hello")` you are providing the value `"hello"` to the function.
  - Values you pass when calling a function are called 🔧arguments.
- Functions are *always* defined and called with **parentheses**, even if there are no parameters.

Change your `spin_animation()` function to define a `count` parameter like this:

```
def spin_animation(count):
    index = 0
    while index < count:
        # ...show img and delay
```

Instead of always `8` the caller will supply `count` which can be any number of *loops* you'd like.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3  import random
4
5  def spin_animation(count):
```

Add a `count` parameter to your function.

```
6      index = 0
7      while index < count:
```

Compare `index` against `count` in your `while` loop.

```
8          display.show(pics.ALL_ARROWS[index])
9          sleep(0.1)
10         index = index + 1
11
12 def show_random_arrow():
13     num = random.randrange(8)
14     display.show(pics.ALL_ARROWS[num])
15
16 while True:
17     if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
18         spin_animation(8)
```

```
    Call spin_animation() with 8 for the count!!


        show_random_arrow()
```

Goals:

- Make your `spin_animation()` function take one parameter named `count`.
- Compare `count` in a `while` loop statement.
- Call `spin_animation()` with an argument of `8`.

Tools Found: Computer Simulations, Functions, Parameters, Arguments, and Returns, Keyword and Positional Arguments

Solution:

```
from codex import *
from time import sleep
import random

def spin_animation(count):
    index = 0
    while index < count:
        display.show(pics.ALL_ARROWS[index])
        sleep(0.1)
        index = index + 1

def show_random_arrow():
    num = random.randrange(8)
    display.show(pics.ALL_ARROWS[num])

while True:
    if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
        spin_animation(8)
        show_random_arrow()
```

Objective 6 - Unruly Index

Time to increase the number of spins

Right now the Arrow goes through 8 positions. *Can you make it keep going around?*

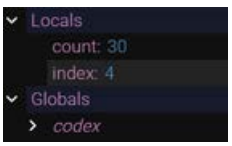
- Go ahead and bump up the number of spins to `30` in your `spin_animation()` function call!

⚠ **WARNING:** You will get an error when you run this.

Debug the Code

Step into the code, and open the **console** window to inspect your **variables**. Now that you have your own functions, here are a few more hints:

- The **Step In** button will enter your function, but if you want to skip over it you can press the **Step Over** button.
- Variables defined *inside* your function (and parameters like `count`) are local variables.
  - You'll find them separately listed in the debug console, as shown here.
- You will need to hold the button down on the CodeX when you **STEP** on the `is_pressed()` call!



What value does the `index` variable have when the error occurs?

CodeTrek:

```
from codex import *
from time import sleep
import random

def spin_animation(count):
```

```
    index = 0
    while index < count:
        display.show(pics.ALL_ARROWS[index])
        sleep(0.1)
        index = index + 1

def show_random_arrow():
    num = random.randrange(8)
    display.show(pics.ALL_ARROWS[num])

while True:
    if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
        # TODO: Call spin_animation with 30 count
```

Call the `spin_animation()` function again.

- This time set the value of the `count` parameter to `30`!

```
        show_random_arrow()
```

Hint:

- You will need to hold a button down on the CodeX when you press the STEP IN button.

Goals:

- Call `spin_animation()` with a parameter `count` of `30`!
- Press a CodeX button and let the program error!
- Use the debugger **Step In** button to step into the spin animation.
  - You will need to hit the debug button again first.
  - You must step at least 20 times!

Tools Found: Exception, Functions, Locals and Globals, Print Function

Solution:

```
from codex import *
from time import sleep
import random

def spin_animation(count):
    index = 0
    while index < count:
        display.show(pics.ALL_ARROWS[index])
        sleep(0.1)
        index = index + 1

def show_random_arrow():
    num = random.randrange(8)
    display.show(pics.ALL_ARROWS[num])

while True:
    if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
        spin_animation(30)
        show_random_arrow()
```

Objective 7 - Tame the Unruly Index

Have you found the error?

The list `pics.ALL_ARROWS` has just 8 elements, indexed 0 through 7.

When your `index` variable reaches **8**, it is *past the end* of the list!

How can you keep `index` in the range 0 - 7?

Here's an idea:

- Use *another variable* called `loops` to count the *total* number of repeats.
- Keep using `index` too, but *reset* it back to **0** when it reaches **8**.

NOTE: Beware the difference between `=` 🔧assignment and `==` 🔧comparison operations!

Try running your code, and make sure the Arrow spins more than one complete cycle!

- **Step** through the code also - **watch `index` and `loops` variables change...**

  Is your code running properly now?

CodeTrek:

```
1  from codex import *
2  from time import sleep
3  import random
4
5  def spin_animation(count):
6      index = 0
7      loops = 0
```

Create a new `loops` variable that will keep track of the **total** loops run.

- `index` will still keep track of the **index**!

```
8      while loops < count:
```

Compare against `loops` instead of `index` in the `while` statement.

```
9          loops = loops + 1
```

Increment `loops`.

```
10         display.show(pics.ALL_ARROWS[index])
11         sleep(0.1)
12         index = index + 1
13         if index == 8:
14             index = 0
```

If `index` goes **OUT OF RANGE** set it back to `0`.

```
15
16 def show_random_arrow():
17     num = random.randrange(8)
18     display.show(pics.ALL_ARROWS[num])
19
20 while True:
21     if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
22         spin_animation(30)
23         show_random_arrow()
```

Goals:

- Compare `loops` in a `while` statement.
- Check `index` for equality `==` with `8` in an `if` statement.

Tools Found: list, Assignment, undefined

Solution:

```
from codex import *
from time import sleep
import random

def spin_animation(count):
    index = 0
    loops = 0
    while loops < count:
        loops = loops + 1
        display.show(pics.ALL_ARROWS[index])
        sleep(0.1)
        index = index + 1
        if index == 8:
            index = 0

def show_random_arrow():
    num = random.randrange(8)
    display.show(pics.ALL_ARROWS[num])

while True:
    if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
        spin_animation(30)
        show_random_arrow()
```

Objective 8 **- Style Points - Physics Part 2**

Spin Down

Now that you can make the animation spin as long as you like, it's time to make the arrow gradually **slow down**.

> **What controls the speed of your animation now?**

The `sleep(0.1)` needs to change, to create *longer* delays, while the loop repeats.

- Right now your code uses the value `0.1`
- Change it to use a 🔧variable!

Add a variable called `delay` to your `spin_animation()` function.

- Start the delay at `0.05` or 50 milliseconds
- Add `0.005` (5 milliseconds) to `delay` after every `sleep(delay)`
- Increase the number of spins so you can easily see the effect!

Debug the Code

There's a *lot* going on in this program! Step through and **watch the variables** as each animation loop runs.

> Make sure you understand what's happening with the *functions* and *variables* in your program!

Can you see the Arrow slooow dooown gradually?

CodeTrek:

```
from codex import *
from time import sleep
import random

def spin_animation(count):
    delay = 0.05
```

Create a `delay` variable that starts at 50 milliseconds.

```
    index = 0
    loops = 0
    while loops < count:
        loops = loops + 1
```

```
11          display.show(pics.ALL_ARROWS[index])
12          # TODO: sleep for the delay
```

> 🔧Sleep for the `delay`.

```
13          delay = delay + 0.005
```

> Add 5 milliseconds to the `delay` every loop.
>
> This will slooooow your animation down!!

```
14          index = index + 1
15          if index == 8:
16              index = 0
17
18  def show_random_arrow():
19      num = random.randrange(8)
20      display.show(pics.ALL_ARROWS[num])
21
22  while True:
23      if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
24          spin_animation(30)
25          show_random_arrow()
```

Goals:

- Create a new `delay` variable and default it to 50 milliseconds.
- 🔧Sleep for the length of `delay` instead of a hard-coded value.

Tools Found: Variables, Timing

Solution:

```
1   from codex import *
2   from time import sleep
3   import random
4
5   def spin_animation(count):
6       delay = 0.05
7       index = 0
8       loops = 0
9       while loops < count:
10          loops = loops + 1
11          display.show(pics.ALL_ARROWS[index])
12          sleep(delay)
13          delay = delay + 0.005
14          index = index + 1
15          if index == 8:
16              index = 0
17
18  def show_random_arrow():
19      num = random.randrange(8)
20      display.show(pics.ALL_ARROWS[num])
21
22  while True:
23      if buttons.is_pressed(BTN_A) or buttons.is_pressed(BTN_B):
24          spin_animation(30)
25          show_random_arrow()
```

Mission 9 Complete

Take your **Game Spinner** for a Spin!

Besides being a useful tool for random selection, this project gave **you** some great tools for making **much** more powerful programs!

Breaking your program down into 🔧functions allows you to do really complex tasks in software, while keeping your code 🔧readable.

🔧Divide and Conquer!

Fast button inputs, animation, and simulation! **That's how you code:**

- Video games
- Flight Simulators
- Virtual Reality

Excellent work!! Ready for more?

Mission 10 - Reaction Tester

How fast is your reaction time?

In this project, you will make a device to measure your reaction time!

Create a device that measures the time between:

- Bright 🔧Pixel LEDs lighting up, and...
- A 🔧CodeX Button being pressed.

After the measurement is complete, this time will be shown on the display until a button is pressed to restart the game.

Who has the fastest reaction time?

With a little coding, you're about to find out!

Project Goals:

- Give the player a 3-2-1 countdown.
- Wait a *random* delay, so they can't "guess" the timing.
- Turn all the pixels GREEN (for go).
- Measure the time until a button press occurs.
- Show the reaction time on the display.
- Wait for a button press, then restart the game.

Ready to get started?

Objective 1 - Milliseconds

For your first step, light up those 🔧pixel LEDs! *And...*

Make it *Unpredictable!*

You wouldn't want the user to just *time* their reaction.

- That's cheating!
- So you need to add a little 🔧random delay to keep 'em guessing!
- After that, it's LIGHTS ON and see how fast they react.

Let's say you want to wait between `1.000` seconds and `5.000` seconds.

- With a 0.001 second resolution. *That's 1 millisecond!*
  - ...try and guess *MY* random time? *No way!*
- How can you accomplish that?

You can use the familiar `random.randrange()` function.

- But... you will need to make a few adjustments.

Random Arguments?

The `randrange()` function can take 🔧arguments just like the 🔧range function.

- If you call it like this: `randrange(1, 5)` you will get a random 🔧integer between `1` and `4`.

*That's good... but you want **millisecond** resolution!*

Use `randrange(1000, 5000)` to get *milliseconds* of delay.

- Then you can divide by `1000` for seconds! *Perfect!*

CodeTrek:

```
1  from codex import *
2  from time import sleep
3  import random
```

```
4
5  # TODO: Display the Countdown...
6
7  # All pixels off, so they're ready to go ON when needed...
8  pixels.set([BLACK, BLACK, BLACK, BLACK])
```

Setting all the pixels to `BLACK` turns them all off.

- `BLACK` is the same as RGB (0, 0, 0)

```
9
10 # Get 1000 to 5000 millisecond delay.
11 ms = random.randrange(1000, 5000)
```

`random.randrange(start, stop)` produces integers.

- You want a 🔧float seconds between 1.0 and 5.0
- Use `randrange()` to get milliseconds and convert to seconds!

```
12
13 # Convert milliseconds to seconds.
14 delay_time = ms / 1000
```

Convert milliseconds to seconds:

Just divide by `1000`!!

```
15 sleep(delay_time)
16
17 # Lights ON: Time to REACT!
18 pixels.set([GREEN, GREEN, GREEN, GREEN])
```

Turn all the pixels `GREEN` after the delay.

Goals:

- Create a **new** file named `Reaction_Time`.
- Get a random millisecond value from `1000` to `5000` with `randrange()`.
- Divide by `1000` to get seconds!
- Light ALL the 🔧pixel LEDs GREEN
  - Use a 🔧list of colors to do this in *one line!*

Tools Found: RGB "pixel" LEDs, Random Numbers, Keyword and Positional Arguments, Ranges, int, list, float

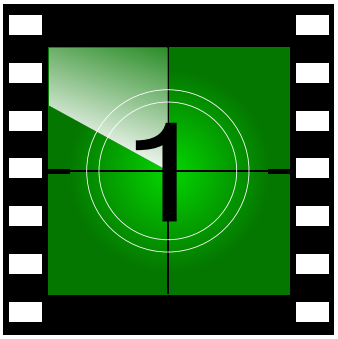
Solution:

```
1  from codex import *
2  from time import sleep
3  import random
4
5  # TODO: Display the Countdown...
6
7  # All pixels off, so they're ready to go ON when needed...
8  pixels.set([BLACK, BLACK, BLACK, BLACK])
9
10 # Get 1000 to 5000 millisecond delay.
11 ms = random.randrange(1000, 5000)
12
13 # Convert milliseconds to seconds.
14 delay_time = ms / 1000
15 sleep(delay_time)
```

```
16
17  # Lights ON: Time to REACT!
18  pixels.set([GREEN, GREEN, GREEN, GREEN])
```

***Objective 2* - The Countdown**

Countdown!



The game should start by giving the player time to prepare:

- Print `"3"` then `"2"` then `"1"` to the *display*

Then the *pixels* and *LCD* should go dark.

- And then a random 🔧delay before the pixels turn on `GREEN`!

Here are a couple of display functions you can use:

1. `display.clear()` - this function clears the display.
2. `display.print()` - displays text 🔧strings like `display.show()` but *scrolls* rather than overwriting text!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3  import random
4
5  # Clear screen.
6  display.clear()
```

`display.clear()` turns the screen black and removes all text.

- This will come in handy later when you make the program repeat!

```
7
8  # All pixels off.
9  pixels.set([BLACK, BLACK, BLACK, BLACK])
10
11 display.print(3, scale=6)
12 sleep(1)
13 display.print(2, scale=6)
14 sleep(1)
15 display.print(1, scale=6)
16 sleep(1)
17 display.clear()
```

`display.print()` doesn't clear the screen like `display.show()` does.

- This countdown will stay on the screen until you clear it.
- Use the `scale` 🔧keyword argument to make the text *BIG!*
- Delay a bit between counts with `sleep()`
- And don't forget to **clear** the display after the countdown.

```
18
19 # Get 1000 to 5000 millisecond delay.
20 ms = random.randrange(1000, 5000)
21
22 # Convert milliseconds to seconds.
23 delay_time = ms / 1000
24 sleep(delay_time)
25
26 pixels.set([GREEN, GREEN, GREEN, GREEN])
```

Goals:

- Clear the display when your program first starts.
- Print the "3"..."2"..."1" *countdown*

Tools Found: Timing, str, Keyword and Positional Arguments

Solution:

```
from codex import *
from time import sleep
import random

# Clear screen.
display.clear()

# All pixels off.
pixels.set([BLACK, BLACK, BLACK, BLACK])

display.print(3, scale=6)
sleep(1)
display.print(2, scale=6)
sleep(1)
display.print(1, scale=6)
sleep(1)
display.clear()

# Get 1000 to 5000 millisecond delay.
ms = random.randrange(1000, 5000)

# Convert milliseconds to seconds.
delay_time = ms / 1000
sleep(delay_time)

pixels.set([GREEN, GREEN, GREEN, GREEN])
```

Objective 3 **- The Fourth Dimension**

You need to measure the time between when the pixels turn ON and when a button is pressed.

CONCEPT: *Computer Clocks*

Computers rely on electronic **clock** circuits. Each **tick** of the CodeX's speedy internal **clock** moves it through your code *one step at a time.* It's really the *heartbeat* of the computer!

What else does the **clock** drive?

- Time delays in `sleep()` functions.
- Scheduled activities within the 🔧CPU.
- ...everything *timing* related on the computer.

From the moment you turn ON the CodeX, the clock is *always running.*

Here's a function in the 🔧time module on CodeX that returns the value of a **counter**.

- That **counter** ticks up 1 every millisecond.
- The starting point is *arbitrary* so it doesn't really have much meaning *except* for measuring time differences.

```
start_time = time.ticks_ms()
```

CONCEPT: *`import` vs `from`*

Until now you have always referenced the `sleep()` function by 🔧importing it from the time module:

```
from time import sleep
```

For this Objective, try a different form of 🔧import, like this:

```
import time
start_time = time.ticks_ms()
time.sleep(1.0)
```

Reaction time is the *time difference* between LIGHTS and BUTTONS!

Ready for some good times?

CodeTrek:

```
1  from codex import *
2  import time
```

Instead of importing the `sleep` function `from` `time` you can `import` the *module* reference.

- This allows you to access anything in the `time` module using "dot notation".

```
3  import random
4
5  # Clear screen.
6  display.clear()
7
8  # All pixels off.
9  pixels.set([BLACK, BLACK, BLACK, BLACK])
10
11 display.print(3, scale=6)
12 time.sleep(1)
```

Since you didn't explicitly bring in `sleep` you will need to use `time.sleep()` instead of just `sleep()`.

- There are a few `sleep()` calls below you need to fix also!

```
13 display.print(2, scale=6)
14 time.sleep(1)
15 display.print(1, scale=6)
16 time.sleep(1)
17 display.clear()
18
19 # Get 1000 to 5000 millisecond delay.
20 ms = random.randrange(1000, 5000)
21
22 # Convert milliseconds to seconds.
23 delay_time = ms / 1000
24 time.sleep(delay_time)
```

Since you didn't explicitly bring in `sleep` you will need to use `time.sleep()` instead of just `sleep()`.

- There are a few `sleep()` calls below you need to fix also!

```
25
26 pixels.set([GREEN, GREEN, GREEN, GREEN])
27
28 start_time = time.ticks_ms()
```

Record the `start_time` just after the lights come on.

- ...now CodeX is waiting for the human to react :-)

```
29
30 # Wait for button A.
31 while True:
32     if buttons.was_pressed(BTN_A):
33         break
```

Now wait for the user to press **Button A**.

```
34
35  end_time = time.ticks_ms()
```

As soon as **Button A** was pressed record the `end_time`.

```
36
37  display.print(start_time)
38  display.print(end_time)
```

Just show start and end on the display.

- This won't mean much but you will learn how to use these values soon!

Goals:

- Use `time.ticks_ms()` twice and save the result in *two* 🔧*variables :*
  - One must be named `start_time`
  - The other must be named `end_time`
- Use the `buttons.was_pressed()` function to wait for `BTN_A`.
  - Print the `start_time` and `end_time` after the button press.

Tools Found: CPU and Peripherals, Time Module, import, Variables

Solution:

```
from codex import *
import time
import random

# Clear screen.
display.clear()

# All pixels off.
pixels.set([BLACK, BLACK, BLACK, BLACK])

display.print(3, scale=6)
time.sleep(1)
display.print(2, scale=6)
time.sleep(1)
display.print(1, scale=6)
time.sleep(1)
display.clear()

# Get 1000 to 5000 millisecond delay.
ms = random.randrange(1000, 5000)

# Convert milliseconds to seconds.
delay_time = ms / 1000
time.sleep(delay_time)

pixels.set([GREEN, GREEN, GREEN, GREEN])

start_time = time.ticks_ms()

# Wait for button A.
while True:
    if buttons.was_pressed(BTN_A):
        break

end_time = time.ticks_ms()

display.print(start_time)
display.print(end_time)
```

***Objective 4* - Time Differential**

You have a `start_time` and an `end_time` !

Now how do you calculate the *reaction time?*

There is one more function from the 🔧time module you need to learn about!

```
elapsed = time.ticks_diff(end, start)
```

This function gives you the elapsed time between the `start_time` and `end_time`.

Wait, but couldn't I just do `elapsed = end - start`?

- That will work most of the time!
- See the 🔧monotonic tool if you're curious why plain subtraction is *dangerous...*

Test Your Reaction Time!

This is starting to get fun :-)

CodeTrek:

```
from codex import *
import time
import random

# Clear screen.
display.clear()

# All pixels off.
pixels.set([BLACK, BLACK, BLACK, BLACK])

display.print(3, scale=6)
time.sleep(1)
display.print(2, scale=6)
time.sleep(1)
display.print(1, scale=6)
time.sleep(1)
display.clear()

# Get 1000 to 5000 millisecond delay.
ms = random.randrange(1000, 5000)

# Convert milliseconds to seconds.
delay_time = ms / 1000
time.sleep(delay_time)

pixels.set([GREEN, GREEN, GREEN, GREEN])

start_time = time.ticks_ms()

# Wait for button A.
while True:
    if buttons.was_pressed(BTN_A):
        break

end_time = time.ticks_ms()

reaction_time = time.ticks_diff(end_time, start_time)
```

Use `time.ticks_diff()` here.

The function takes two parameters.

- The first should be `end_time`.
- The second should be `start_time`.

```

display.print("Reaction time:")
```

```
40  display.print(reaction_time)
41  display.print("milliseconds")
```

print() the reaction time to the display!

Goals:

- Use `time.ticks_diff()` to measure the time between `start_time` and `end_time`.
- Print the reaction time on the display

Tools Found: Time Module, Monotonic

Solution:

```
1  from codex import *
2  import time
3  import random
4
5  # Clear screen.
6  display.clear()
7
8  # All pixels off.
9  pixels.set([BLACK, BLACK, BLACK, BLACK])
10
11 display.print(3, scale=6)
12 time.sleep(1)
13 display.print(2, scale=6)
14 time.sleep(1)
15 display.print(1, scale=6)
16 time.sleep(1)
17 display.clear()
18
19 # Get 1000 to 5000 millisecond delay.
20 ms = random.randrange(1000, 5000)
21
22 # Convert milliseconds to seconds.
23 delay_time = ms / 1000
24 time.sleep(delay_time)
25
26 pixels.set([GREEN, GREEN, GREEN, GREEN])
27
28 start_time = time.ticks_ms()
29
30 # Wait for button A.
31 while True:
32     if buttons.was_pressed(BTN_A):
33         break
34
35 end_time = time.ticks_ms()
36
37 reaction_time = time.ticks_diff(end_time, start_time)
38
39 display.print("Reaction time:")
40 display.print(reaction_time)
41 display.print("milliseconds")
```

***Objective 5* - Let's Keep Playing**

Play Again?

Great job so far! The *"reaction game"* is fun, but what if you want to play more than once?

Make the game *wait for a button press*, then start again!

- You need a loop that contains *all the code* from "3-2-1" on down.
- Add code to *wait for a button press* before continuing the loop.

Hey! You *already* have code that waits for a button press...

Feel free to copy that "wait for BTN_A" code.

Go ahead and place everything after `import random` inside a `while` loop.

- *Remember you can select it all and use the TAB key to indent a whole block*
- You *have* been using the 🔧Editor Shortcuts, right?

CodeTrek:

```
1  from codex import *
2  import time
3  import random
4
5  while True:
```

Add an infinite `while True` loop so that you can keep playing the game!

```
6      display.print("Press Button A")
```

Tell the user it is time to press **Button A**!

```
7      # wait for button A
8      while True:
9          if buttons.was_pressed(BTN_A):
10             break
```

Wait for **Button A** before moving into the game!

(even on the first run through)

```
11
12     # Clear screen.
13     display.clear()
14
15     # All pixels off.
16     pixels.set([BLACK, BLACK, BLACK, BLACK])
17
18     display.print(3, scale=6)
19     time.sleep(1)
20     display.print(2, scale=6)
21     time.time.sleep(1)
22     display.print(1, scale=6)
23     time.sleep(1)
24     display.clear()
25
26     # Get 1000 to 5000 millisecond delay.
27     ms = random.randrange(1000, 5000)
28
29     # Convert milliseconds to seconds.
30     delay_time = ms / 1000
31     time.sleep(delay_time)
32
33     pixels.set([GREEN, GREEN, GREEN, GREEN])
34
35     start_time = time.ticks_ms()
36
37     # Wait for button A.
38     while True:
39         if buttons.was_pressed(BTN_A):
40             break
41
42     end_time = time.ticks_ms()
43
44     reaction_time = time.ticks_diff(end_time, start_time)
45
46     display.print("Reaction time:")
```

```
47      display.print(reaction_time)
48      display.print("milliseconds")
```

Goals:

- Add an infinite `while True` loop to keep playing your game.
- Wait for the user to press `BTN_A` to play again.
    - *Your code should now have 3 `while` loops in total!*

Tools Found: Loops, Editor Shortcuts

Solution:

```
1  from codex import *
2  import time
3  import random
4
5  while True:
6      display.print("Press Button A")
7      # wait for button A
8      while True:
9          if buttons.was_pressed(BTN_A):
10             break
11
12     # Clear screen.
13     display.clear()
14
15     # All pixels off.
16     pixels.set([BLACK, BLACK, BLACK, BLACK])
17
18     display.print(3, scale=6)
19     time.sleep(1)
20     display.print(2, scale=6)
21     time.sleep(1)
22     display.print(1, scale=6)
23     time.sleep(1)
24     display.clear()
25
26     # Get 1000 to 5000 millisecond delay.
27     ms = random.randrange(1000, 5000)
28
29     # Convert milliseconds to seconds.
30     delay_time = ms / 1000
31     time.sleep(delay_time)
32
33     pixels.set([GREEN, GREEN, GREEN, GREEN])
34
35     start_time = time.ticks_ms()
36
37     # Wait for button A.
38     while True:
39         if buttons.was_pressed(BTN_A):
40             break
41
42     end_time = time.ticks_ms()
43
44     reaction_time = time.ticks_diff(end_time, start_time)
45
46     display.print("Reaction time:")
47     display.print(reaction_time)
48     display.print("milliseconds")
```

Objective 6 **- Reduce Repetition**

Take a look at your code.

Do you notice a block of code that's ***repeated?***

It works just fine, **but** you can make this code more 🔧readable and *maintainable*.

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

Here is ancient coding wisdom:

> *Never write the same code twice.*

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

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

> *All that copy-and-paste business? Nah, make a 🔧function instead!*

CodeTrek:

```
from codex import *
import time
import random

def wait_button():
```

Make the `wait_button()` function.

> Then move all the code to wait for **Button A** inside it!

```
    # Wait for button A.
    while True:
        if buttons.was_pressed(BTN_A):
            break

while True:
    display.print("Press Button A")
    wait_button()
```

Wait on first load.

```

    # Clear screen.
    display.clear()

    # All pixels off.
    pixels.set([BLACK, BLACK, BLACK, BLACK])

    display.print(3, scale=6)
    time.sleep(1)
    display.print(2, scale=6)
    time.sleep(1)
    display.print(1, scale=6)
    time.sleep(1)
    display.clear()

    # Get 1000 to 5000 millisecond delay.
    delay_time = random.randrange(1000, 5000) / 1000
```

You can divide by `1000` here if you prefer.

- That will just reduce a line of code...

```
    time.sleep(delay_time)

    pixels.set([GREEN, GREEN, GREEN, GREEN])

    start_time = time.ticks_ms()

    wait_button()
```

```
    Wait again after the pixels turn GREEN.


    end_time = time.ticks_ms()

    reaction_time = time.ticks_diff(end_time, start_time)

    display.print("Reaction time:")
    display.print(reaction_time)
    display.print("milliseconds")
```

Goals:

- Create a function called `wait_button()`.
- Call the `wait_button()` function twice in your code.

Tools Found: Readability, Functions

Solution:

```
from codex import *
import time
import random

def wait_button():
    # Wait for button A.
    while True:
        if buttons.was_pressed(BTN_A):
            break

while True:
    display.print("Press Button A")
    wait_button()

    # Clear screen.
    display.clear()

    # All pixels off.
    pixels.set([BLACK, BLACK, BLACK, BLACK])

    display.print(3, scale=6)
    time.sleep(1)
    display.print(2, scale=6)
    time.sleep(1)
    display.print(1, scale=6)
    time.sleep(1)
    display.clear()

    # Get 1000 to 5000 millisecond delay.
    delay_time = random.randrange(1000, 5000) / 1000
    time.sleep(delay_time)

    pixels.set([GREEN, GREEN, GREEN, GREEN])

    start_time = time.ticks_ms()

    wait_button()

    end_time = time.ticks_ms()

    reaction_time = time.ticks_diff(end_time, start_time)

    display.print("Reaction time:")
    display.print(reaction_time)
    display.print("milliseconds")
```

Quiz 1 **- Quiz Timing**

Question 1: How many milliseconds are in a second?

✓ 1000

✗ 100

✗ 0.001

✗ 1000000

Question 2: Select the three correct statements about functions.

✓ They help keep your code organized.

✓ You can reuse them multiple times.

✓ It is easier to make a change in one place than in repeated code.

✗ They ensure values are always increasing monotonically.

Question 3: What does the `time.ticks_diff(end, start)` function do?

✓ It returns the time difference between start and end.

✗ It changes the clock on your computer by the diff.

✗ It predicts the end of time given a start time.

Objective 7 - No Cheating

Fix a *BUG!*

Oh No!! Users are pressing the button during the delay and getting **ULTRA** fast times.

> The `buttons.was_pressed()` function is always listening. Even during the random delay...

So how can you stop it?

Check `buttons.was_pressed()` **JUST** before turning on the `GREEN` light!

- Remember from the 🔧CodeX Buttons Toolbox help, the `was_pressed()` function remembers whether the button was pressed since the last time it was called. That means it *resets* to "not-pressed" after you call it!
- So call `buttons.was_pressed(BTN_A)` to reset the button state and prevent *cheating!*

Make it *cheat-proof!*

CodeTrek:

```
from codex import *
import time
import random

def wait_button():
    # Wait for button A.
    while True:
        if buttons.was_pressed(BTN_A):
            break

while True:
    display.print("Press Button A")
    wait_button()

    # Clear screen.
    display.clear()
```

```
17
18      # All pixels off.
19      pixels.set([BLACK, BLACK, BLACK, BLACK])
20
21      display.print(3)
22      display.print(2)
23      display.print(1)
24
25      # Get 1000 to 5000 millisecond delay.
26      delay_time = random.randrange(1000, 5000) / 1000
27      time.sleep(delay_time)
28
29      # Reset button state to prevent cheating
30      buttons.was_pressed(BTN_A)
```

Calling `buttons.was_pressed()` here will reset the condition and only accept **NEW** presses.

```
31
32      pixels.set([GREEN, GREEN, GREEN, GREEN])
33
34      start_time = time.ticks_ms()
35
36      wait_button()
37
38      end_time = time.ticks_ms()
39
40      reaction_time = time.ticks_diff(end_time, start_time)
41
42      display.print("Reaction time:")
43      display.print(reaction_time)
44      display.print("milliseconds")
```

Goal:

- Reset the `buttons.was_pressed(BTN_A)` just before setting the pixels to `GREEN`.

Tools Found: CodeX Buttons

Solution:

```
1   from codex import *
2   import time
3   import random
4
5   def wait_button():
6       # Wait for button A.
7       while True:
8           if buttons.was_pressed(BTN_A):
9               break
10
11  while True:
12      display.print("Press Button A")
13      wait_button()
14
15      # Clear screen.
16      display.clear()
17
18      # All pixels off.
19      pixels.set([BLACK, BLACK, BLACK, BLACK])
20
21      display.print(3, scale=6)
22      time.sleep(1)
23      display.print(2, scale=6)
24      time.sleep(1)
25      display.print(1, scale=6)
26      time.sleep(1)
27      display.clear()
28
29      # Get 1000 to 5000 millisecond delay.
```

```
    delay_time = random.randrange(1000, 5000) / 1000
    time.sleep(delay_time)

    # Reset button state to prevent cheating
    buttons.was_pressed(BTN_A)

    pixels.set([GREEN, GREEN, GREEN, GREEN])

    start_time = time.ticks_ms()

    wait_button()

    end_time = time.ticks_ms()

    reaction_time = time.ticks_diff(end_time, start_time)

    display.print("Reaction time:")
    display.print(reaction_time)
    display.print("milliseconds")
```

Mission 10 Complete

Impeccable Timing!

Computers *measure time* in all types of applications.

- Football *play clocks* and *stop watches for other sports*.
- Electronic Drum Machines
- Microwave Oven timers
- Alarm clocks

Time to move on?

Mission 11 **- Spirit Level**

Level Up!

How level is your desk or table?

Write some code to find out! In this project you'll build a *spirit level!*

You will create a **digital level** using the CodeX's built-in 🔧accelerometer and display. You'll physically rotate the CodeX to *move* the digital "bubble" on the display!

Project Goals:

- Display a numeric "tilt" value from the accelerometer.
- *Scale* the raw tilt value to show 0° to 90° incline.
- Replace the number display with a **graphical** *ball* simulation!

Ready to get started?

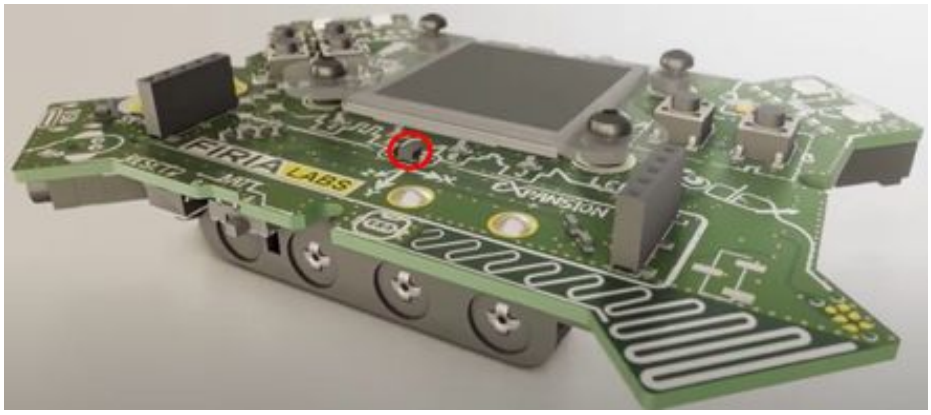
Objective 1 **- Accel**

First step is to find the 🔧**accelerometer**.

The CodeX uses a 3-axis accelerometer to detect orientation.

> The 3 axes are **X**, **Y**, and **Z**.

Take a look at the *front* of your **CodeX** just *below* the display. You should see a little **3-axis diagram** that shows all three axes.



Just above the **3-axis diagram** is the *tiny little* accelerometer chip!

Goals:

- Create a **new** file named `Spirit_Level`.
- Find the CodeX's **accelerometer** in the 3D viewer.

- Also *check out* the 3-axis diagram just below it!

Tools Found: Accelerometer

Solution:

N/A

Objective 2 **- Tilt-o-Matic**

The next step is to read the 🔧accelerometer.

When you `read()` from the accelerometer it returns a 🔧tuple `(x, y, z)`

A tuple is a lot like a 🔧list! The only real difference is that you can't *change* the values in a 🔧tuple.

This is an example of a tuple: `(0, 0, 0)`

- Notice it uses `()` instead of `[]` like a list!

This is how you read from the CodeX accelerometer:

```
val = accel.read() # ex: val is (0, 0, -16383)
```

Using the above example, the **raw value** of **x** would be 0. You can access the **x** value with `val[0]`.

Acceleration Values

The tuple has non-zero values if there is **acceleration** detected!

- When the CodeX is not *moving*, the only **acceleration** it *feels* is the earth's gravity.
- That will come in handy for this project - *Earth's gravity is the ultimate authority on "Level" after all.*
- **The full force of gravity (1g) will show up as: +/- 16383**

If you *move* or *shake* the CodeX, you can create *larger* acceleration values! In the next step, you'll add an `if` statement to make sure out-of-range values don't mess things up.

Follow the *CodeTrek* to write some test code...

Run your code and tilt your CodeX to watch values change!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  while True:
5      val = accel.read()
```

Read a 🔧tuple from the accelerometer.

```
6      display.print(val[0])
```

Print the **x** values on the display.

- Be sure to tilt your CodeX in the **x** direction.

```
7      sleep(0.2)
```

Give yourself time to read the values as they scroll!

Goals:

- Use the built-in `accel.read()` to assign an `(x, y, z)` 🔧tuple to a variable.
- Access the **x** value of the 🔧tuple using the `0` index!
    - Example: `val[0]`

Tools Found: Accelerometer, tuple, list

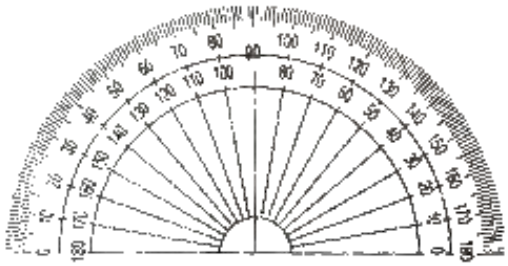
Solution:

```
from codex import *
from time import sleep

while True:
    val = accel.read()
    display.print(val[0])
    sleep(0.2)
```

Objective 3 **- Scale to Degrees**

To get an accurate *digital* reading, you need *real* units of measure!

If a protractor is placed horizontally on a level surface, **0° is level.**



Make a *digital level* that shows *degrees*

To start with:

- `x` should be between `-16384` and `16384` when **only** gravity acts on it!

The tricky part is **converting** from the raw value to degrees.

- In the code below, do you see how `x / 16384` will be a fraction between -1 and +1?
- First you have to make sure it doesn't exceed those limits!
- Then use a bit of *trigonometry* * to calculate the angle.

* If you haven't learned this math yet, don't worry! You can just type in the code as shown. But if you're interested to know how it works, see the *Hints* for more info!

```
# Scale the value to +/- 1.0
scaled = (tilt_x / 16384)

# Cap max and min value
scaled = min(max(scaled, -1), 1)

# Calculate degrees
degrees = math.asin(scaled) * 180 / math.pi

# Just an integer, please
degrees = int(degrees)
```

CodeTrek:

```
from codex import *
from time import sleep
import math
```

Using some *math*

This is for the angle calculation below. *Relax...* the math is here to help you!

```


while True:
    val = accel.read()

    # Get only the x value
```

```
    tilt_x = val[0]

    # Scale the value to +/- 1.0
    scaled = (tilt_x / 16384)
```

The max *resting* value of the accelerometer should be 16383

- So this fraction should be less than 1.0

```

    # Cap max and min value
    scaled = min(max(scaled, -1), 1)
```

Bonus Built-ins!

Restrict `scaled` to a max of 1 and a min of -1

- Those `min()` and `max()` functions are Python built-ins

```

    # Calculate degrees
    degrees = math.asin(scaled) * 180 / math.pi
```

The `math.asin()` function returns an angle in *radians*.

- Multiply by `180 / math.pi` to convert that to *degrees*.
- Check *hints* for more detail on the trig if you're interested!

```

    # Just an integer, please
    degrees = int(degrees)
```

We want the degree value as an int not a float!

```

    display.print(degrees)

    sleep(0.2)
```

Hint:

- ## Calculating the Angle

  Earth's gravity is pulling downward. So as you tilt CodeX, you're changing the angle between the X-axis and the actual gravity axis pointing down (Z-axis in pic below)

  - When X is pointing straight down (90°) it is the same as the Z-axis.
  - When X is horizontal (0°), there is no Z-axis component (Z=0)

The sine function relates the opposite and hypotenuse to the angle "a":

$$sin(a) = \frac{Z}{16384}$$

To calculate the angle "a" we need to use *inverse sine*, which is `math.asin()`

```
a = math.asin(z / 16384)
```

That gives us an angle in Radians. Convert this to degrees as follows:

$$deg = rad \times \frac{180}{\pi}$$

Goals:

- Create a scaled value with this formula: `(tilt_x / 16384)`
- Use the `min()` and `max()` 🔧built-in functions to limit the scaled value to ±1
- Convert the scaled value to degrees using `math.asin()` and `math.pi`.

Tools Found: Built-In Functions, Accelerometer, int

Solution:

```
from codex import *
from time import sleep
import math

while True:
    val = accel.read()

    # Get only the x value
    tilt_x = val[0]

    # Scale the value to +/- 1.0
    scaled = (tilt_x / 16384)

    # Cap max and min value
    scaled = min(max(scaled, -1), 1)

    # Calculate degrees
    degrees = math.asin(scaled) * 180 / math.pi

    # Just an integer, please
    degrees = int(degrees)

    display.print(degrees)

    sleep(0.2)
```

Objective 4 **- Static Ball**

Time to learn a little about drawing on the display!!

Here are the functions you will need for your **spirit level**:

Function	Description
`display.fill(color)`	Fill the display with a **color**
`display.draw_line(x1, y1, x2, y2, color)`	Draw a line from (x1, y1) to (x2, y2)
`display.draw_circle(x, y, radius, color)`	Draw a circle with center at (x, y)

CONCEPT: *The Display*

The CodeX LCD 🔧display is 240 pixels x 240 pixels

Each *tiny* pixel works JUST like the 4 RGB *LED pixels* at the top of the CodeX.

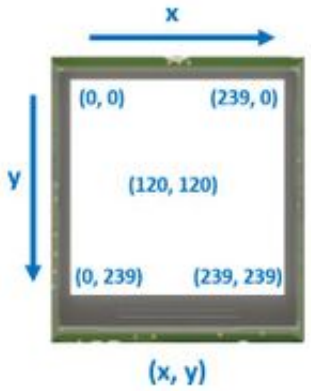
- *x* in the (x, y) is the display *width*
- *y* is the display *height*

See the image at right for a *visual*.

Now type in the code from the CodeTrek!!

This Objective will just be a "static" drawing at first... Next you'll hook it into the accelerometer values!

You will get way more drawing fun in later lessons!



CodeTrek:

```
1  from codex import *
2  from time import sleep
3  import math
4
5  CENTER = 120
```

The center **x** and center **y** pixel is `120`!

```
6
7  # Create a center line on the display
8  display.fill(WHITE)
```

Start by making the whole display `WHITE`.

```
9  display.draw_line(CENTER, 0, CENTER, 105, BLACK)
10 display.draw_line(CENTER, 135, CENTER, 239, BLACK)
```

Add a center line to see if your ball is **level**.

- It is broken into two parts to let the ball pass through its *middle*.

```
11
12 while True:
13     val = accel.read()
14
15     # Get only the x value
16     tilt_x = val[0]
17
18     # Scale the value to +/- 1.0
19     scaled = (tilt_x / 16384)
20
21     # Cap max and min value
22     scaled = min(max(scaled, -1), 1)
23
24     # Calculate degrees
25     degrees = math.asin(scaled) * 180 / math.pi
26
27     # Just an integer, please
28     degrees = int(degrees)
29
30     # Draw the ball
31     display.draw_circle(CENTER, CENTER, 15, ORANGE)
```

Draw a circle to represent your ball.

- This ball will move *left* and *right* on the display.
- If the ball is in the center that is **level**.

Right now the ball will just stay in the center!!

```
32
33     sleep(0.2)
```

Goals:

- Use `display.fill()` to color the display `WHITE`.
- Create a center line with `display.draw_line()`.

- Create the **level** indicator with `display.draw_circle()`.

Tools Found: Display, Accelerometer

Solution:

```
from codex import *
from time import sleep
import math

CENTER = 120

# Create a center line on the display
display.fill(WHITE)
display.draw_line(CENTER, 0, CENTER, 105, BLACK)
display.draw_line(CENTER, 135, CENTER, 239, BLACK)

while True:
    val = accel.read()

    # Get only the x value
    tilt_x = val[0]

    # Scale the value to +/- 1.0
    scaled = (tilt_x / 16384)

    # Cap max and min value
    scaled = min(max(scaled, -1), 1)

    # Calculate degrees
    degrees = math.asin(scaled) * 180 / math.pi

    # Just an integer, please
    degrees = int(degrees)

    # Draw the ball
    display.draw_circle(CENTER, CENTER, 15, ORANGE)

    sleep(0.2)
```

***Objective 5* - Rolling Stone**

Time to make that ball move

You already know that the `CENTER` **X** value of the display is `120`.

You also have a `degrees` value from `-90` to `90`.

Putting it all together, *check it!*

When `degrees == 0` you want your ball right in the *center* of the display.

```
# When degrees == 0?
x = degrees + CENTER   # x = (0 + 120) = 120
```

Let's check the other extremes:

```
# When degrees == -90?
x = degrees + CENTER  # x = (-90 + 120) = 30
```

Your circle's **radius** is `15` pixels, so if you put your circle's center at (30, 120) you will still be able to draw the *whole* circle. *Checks out!*

```
# When degrees == +90?
x = degrees + CENTER  # x = (90 + 120) = 210
```

Same thing if you put your circle's center at (210, 120)!!

This is gonna work nicely!

CodeTrek:

```
from codex import *
from time import sleep
import math

CENTER = 120

# Create a center line on the display
display.fill(WHITE)
display.draw_line(CENTER, 0, CENTER, 105, BLACK)
display.draw_line(CENTER, 135, CENTER, 239, BLACK)

while True:
    val = accel.read()

    # Get only the x value
    tilt_x = val[0]

    # Scale the value to +/- 1.0
    scaled = (tilt_x / 16384)

    # Cap max and min value
    scaled = min(max(scaled, -1), 1)

    # Calculate degrees
    degrees = math.asin(scaled) * 180 / math.pi

    # Just an integer, please
    degrees = int(degrees)

    x = CENTER + degrees
```

Calculate the **x** value of the circle by adding 120.

If `degrees` is 0 the **x** value will be 120.

- That is the center of the display!!

```

    # Draw the new circle
    display.draw_circle(x, CENTER, 15, ORANGE)
```

Use your new calculated **x** value as the circle's *center x* value!

```

    sleep(0.2)
```

Goals:

- Create a variable named `x` that will be the circle's center.
- Use your `x` variable in the `display.draw_circle()` function!

Solution:

```
from codex import *
from time import sleep
import math

CENTER = 120

# Create a center line on the display
display.fill(WHITE)
display.draw_line(CENTER, 0, CENTER, 105, BLACK)
display.draw_line(CENTER, 135, CENTER, 239, BLACK)

```

```
while True:
    val = accel.read()

    # Get only the x value
    tilt_x = val[0]

    # Scale the value to +/- 1.0
    scaled = (tilt_x / 16384)

    # Cap max and min value
    scaled = min(max(scaled, -1), 1)

    # Calculate degrees
    degrees = math.asin(scaled) * 180 / math.pi

    # Just an integer, please
    degrees = int(degrees)

    x = CENTER + degrees

    # Draw the new circle
    display.draw_circle(x, CENTER, 15, ORANGE)

    sleep(0.2)
```

Quiz 1 **- Accelisplay**

Question 1: If the accelerometer returns an (x, y, z) tuple then what direction force is the `d` variable below?

```
val = accel.read()
d = val[1]
```

✓ y

✗ z

✗ x

Question 2: How many pixels is the CodeX display (width x height)?

✓ 240 x 240

✗ 120 x 120

✗ 1080 x 1080

Question 3: Why is `tilt` divided by `16384` in the code below?

```
val = accel.read()
tilt = val[0]
scaled = (tilt / 16384)
```

✓ 16384 is the max expected value for tilt, so (tilt / 16384) will be ≤ 1

✗ 16384 is the universal gravity wave coefficient.

✗ There are 16384 accelerons per degree.

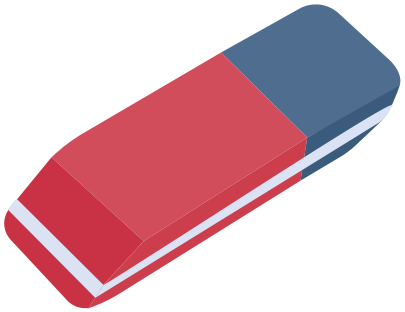
Objective 6 **- Eraser First**

Spirit Level - *Final Touches*

What is going on? It's not working quite right yet...

Why is the ball always drawing on top of itself?

*It's because you are never **erasing** it!*



- Making a circle on the display just changes the color of the pixels you are *drawing*.

Covering Your Tracks

To erase the ball, you just need to draw a `WHITE` circle on top of the old one!

- That does mean you need to keep track of where the old one is...

To the *CodeTrek!*

CodeTrek:

```
1  from codex import *
2  from time import sleep
3  import math
4
5  CENTER = 120
6
7  # Create the center line on the display
8  display.fill(WHITE)
9  display.draw_line(CENTER, 0, CENTER, 105, BLACK)
10 display.draw_line(CENTER, 135, CENTER, 239, BLACK)
11
12 x = CENTER
```

Define the variable `x` before the `while` loop.

- That will make sure it is available for the next **loop** so you can *erase* your old circle!

```
13
14 while True:
15     val = accel.read()
16
17     # Get only the x value
18     tilt_x = val[0]
19
20     # Scale the value to +/- 1.0
21     scaled = (tilt_x / 16384)
22
23     # Cap max and min value
24     scaled = min(max(scaled, -1), 1)
25
26     # Calculate degrees
27     degrees = math.asin(scaled) * 180 / math.pi
28
29     # Just an integer, please
30     degrees = int(degrees)
31
32     # Erase the old circle
33     display.draw_circle(x, CENTER, 15, WHITE)
```

Draw a `WHITE` circle on top of the **old** *colored* circle to **erase** it!

```
34
35     x = CENTER + degrees
36
37     # Draw the new circle
38     display.draw_circle(x, CENTER, 15, ORANGE)
39
40     sleep(0.2)
```

Goal:

- Draw a `WHITE` circle over the old one to erase it *BEFORE* drawing the new circle!

Solution:

```python
from codex import *
from time import sleep
import math

CENTER = 120

# Create the center line on the display
display.fill(WHITE)
display.draw_line(CENTER, 0, CENTER, 105, BLACK)
display.draw_line(CENTER, 135, CENTER, 239, BLACK)

x = CENTER

while True:
    val = accel.read()

    # Get only the x value
    tilt_x = val[0]

    # Scale the value to +/- 1.0
    scaled = (tilt_x / 16384)

    # Cap max and min value
    scaled = min(max(scaled, -1), 1)

    # Calculate degrees
    degrees = math.asin(scaled) * 180 / math.pi

    # Just an integer, please
    degrees = int(degrees)

    # Erase the old circle
    display.draw_circle(x, CENTER, 15, WHITE)

    x = CENTER + degrees

    # Draw the new circle
    display.draw_circle(x, CENTER, 15, ORANGE)

    sleep(0.2)
```

Mission 11 Complete

Really, Level With Me!

Take a few minutes to play with the spirit level. If you disconnect the USB cable and add batteries, you can take the level with you anywhere you want!

Accelerometers used as *tilt sensors* are important and used every day for:

- Controlling your phone screen (landscape or portrait)
- Building a house
- Flying Airplanes
- Keeping Solar Panels pointed at the Sun
- ...and tons of other applications!

Congratulations, you're leveling-up!

Mission 12 - Night Light

Make a smart Night Light that turns ON when the room gets dark.

You'll use the CodeX's built-in 🔧light sensor to detect ambient light and the pixels as a night light!

Project Goals: create *two* versions of the Night Light:

1. Simple on/off control
- Light turns ON when sensor crosses a pre-set "dark threshold".
2. Variable dimming
- The darker it gets, the brighter it shines!

Ready to light up the night?

"May it be a light to you in dark places, when all other lights go out."

> Galadriel (J.R.R. Tolkien), *Fellowship of the Ring*

Objective 1 - Let There Be Sensor

So you want to make a night light?

That is going to be easy with the CodeX!

The CodeX has its own light sensor

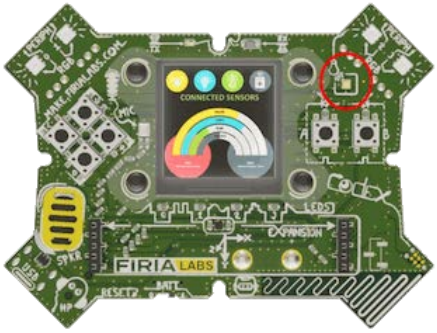
The CodeX 🔧light sensor can read the amount of *ambient* light that reaches it.

> Just like your eyes, it can detect the light within the visible *wavelengths*!

It is also really easy to use.

- Just a little Python code and you're sensing light!

The light sensor is on the front of the CodeX just to the right of the 🔧display:



Goals:

- Create a **new** file named `NightLight`.
- Find the digital ambient **light sensor** in the 3D view.

Tools Found: Light Sensor, Display

Solution:

N/A

Objective 2 - Light Sensing Code

Write some code to "read" from the light sensor!

Getting a basic reading is pretty easy...

```
value = light.read()
```

The light sensor converts **light level** into a *digital* value.

- **dark** = lower values
- **bright** = higher values

The `light.read()` function returns an ADC value. It's a 16 bit number, so the max value is (2^{16} - 1) or 65,535.

Any value below `2000` or so is pretty dark!

Run your program

And try shading the light sensor with your hand!!

What happens to the values?

CodeTrek:

```
from codex import *

while True:
    value = # TODO: read from light sensor
```

Use `light.read()`!

```
    display.print(value)
```

Start by printing the values.

- You will remove this in the next step!

Goal:

- Read from the ambient light sensor with `light.read()`

Tools Found: Light Sensor, Analog to Digital Conversion, Binary Numbers

Solution:

```
from codex import *

while True:
    value = light.read()
    display.print(value)

```

Objective 3 - Pixel Filler

Now you need to make a **light** that you can control.

The *Code* in the Arena

Some big Arenas and NFL stadiums have **huge** LED lights controlled by **code** running on *tiny* wireless electronic boards like the CodeX!

For your *Night Light*, rather than showing an image or text, just light up **ALL** the *RGB pixel LEDs* on the CodeX!

- You will find out how bright they can be soon!

Here is a new function for you to use:

```
pixels.fill(WHITE)
```

It is the same as setting all 4 pixels like this:

```
pixels.set([WHITE, WHITE, WHITE, WHITE])
```

Checking a Threshold

IF it's dark, *turn ON the light!* **ELSE**, turn it OFF.

- The **CodeTrek** will guide you if you need a refresher on 🔧control flow in Python.

Test Your Nightlight!

CodeTrek:

```
1  from codex import *
2
3  while True:
4      value = light.read()
5
6      if value < 2000:
```

Adjust this value for *your* environment.

- `2000` is just an approximate setting.

```
7          # It's getting dark in here!
8          # TODO: fill all pixels WHITE
```

Use the new `pixels.fill()` function!

```
9      else:
10         # It's bright enough now.
11         # TODO: turn off all pixels
```

Turning all the pixels `BLACK` will turn them off.

Hints:

- ## Finding *Your* Threshold

  The value `2000` is just an approximate value, based on typical readings in a room with a "moderate" amount of ambient light.

  - Feel free to adjust as needed for *your* environment.

- ### Not Dark Enough?

  - If you have a **bright** environment, such as a window with **sunlight** streaming in, it may be *difficult* to completely **shade** the sensor.
  - Try moving to a darker area or using an opaque material to completely cover the sensor.

Goals:

- Use the `pixels.fill()` function to set all the pixels `WHITE`.
- Use an `if... else` 🔧control flow statement to check your light level against a threshold.

Tools Found: Branching

Solution:

```
from codex import *

while True:
    value = light.read()

    if value < 2000:
        pixels.fill(WHITE)
    else:
        pixels.fill(BLACK)
```

***Objective 4* - Dimmable Light Sensor**

Dim It!

Your night light is either fully ON or completely OFF.

*But if it's only **slightly** dark, just a little light will do...*

Make the night light *gradually* brighten as the room gets darker!

But how do you dim the pixels?

Well, `pixels.fill()` has an optional argument called `brightness`.

`brightness` takes a value from 0 to 100.

Here is how you use it:

```
pixels.fill(WHITE, brightness=20)
```

Warning!!

Pixels can get VERY bright!

Be careful not to look directly at them at higher brightness levels!

CodeTrek:

```
from codex import *

# The "ambient" room light level.
# Darker than this and the nightlight should shine!
ROOM = 15000
```

Define a constant named `ROOM` for the point where the night light **FIRST** turns **ON**!

```

while True:
    value = light.read()

    if value < ROOM:
        scaled = (value / ROOM) * 20
```

Scale the light sensor value by the new `ROOM` value.

We are going to make the max brightness `20` percent.

- Any more than that is just blinding!!

```
12
13          level = int(scaled)
```

Only want the 🔧int value, not a fraction.

```
14
15          pixels.fill(WHITE, brightness=level)

16      else:
17          pixels.fill(BLACK)
```

Hints:

- ## Test Your Room

  Try `light.read()` to find out what your `ROOM` level should be.

- ## Expect a Problem!

  The nightlight won't work properly... *YET!*

  - Proceed to the next Objective for a fix :-)

Goal:

- Use `pixels.fill()` with the `brightness` parameter.
  - You must use a 🔧keyword argument (`brightness` must be spelled out inside `pixels.fill()`)

Tools Found: Default function parameters, Keyword and Positional Arguments, Constants, int

Solution:

```
1  from codex import *
2
3  # The "ambient" room light level.
4  # Darker than this and the nightlight should shine!
5  ROOM = 15000
6
7  while True:
8      value = light.read()
9
10     if value < ROOM:
11         scaled = (value / ROOM) * 20
12
13         level = int(scaled)
14
15         pixels.fill(WHITE, brightness=level)
16     else:
17         pixels.fill(BLACK)
```

***Quiz 1* - Light Test**

Question 1: What does `light.read()` do in the CodeX built-in library?

✓ Returns the level of ambient light.

✗ Checks if there is enough light for you to read.

✗ Reads the light level of the display.

Question 2: What are the colors of the 4 CodeX pixels after running this code?

```
from codex import *
pixels.set([BLUE, BLUE, BLUE, BLUE])
pixels.set(2, RED)
```

✓ BLUE, BLUE, RED, BLUE

✗ OFF, OFF, RED, OFF

✗ BLUE, BLUE, BLUE, BLUE

✗ BLUE, RED, BLUE, BLUE

***Objective 5* - Reversed**

The light is getting darker as the room gets darker

That is not very helpful...

You want it to get **brighter** as the room gets **darker**.

- You will need to reverse the impact of the light sensor value.

Take some time to examine your code...

Consider This:

Say `value / ROOM` is `0.2`

- Try subtracting it from 1: `(1 - 0.2)` is `0.8`
- So, if you subtract the ratio from `1` it will make the `scaled` variable get bigger as the sensor `value` gets smaller.

```
scaled = (1 - value / ROOM) * 20
```

You can do this!

CodeTrek:

```
1  from codex import *
2
3  # The "ambient" room light level.
4  # Darker than this and the nightlight should shine!
5  ROOM = 15000
6
7  while True:
8      value = light.read()
9
10     if value < ROOM:
11         scaled = (1 - value / ROOM) * 20
```

Reverse the sensor value's impact by subtracting the ratio from 1.

- This will make the light get **brighter** as the room gets **darker**.

```
12
13         level = int(scaled)
14
15         pixels.fill(WHITE, brightness=level)
16     else:
17         pixels.fill(BLACK)
```

Goal:

- Subtract the *ratio* from 1 to reverse the impact of your sensor reading.
    - The "minus sign" takes your light reading in the *negative* direction...
    - *More bright, LESS light!*

Solution:

```
from codex import *

# The "ambient" room light level.
# Darker than this and the nightlight should shine!
ROOM = 15000

while True:
    value = light.read()

    if value < ROOM:
        scaled = (1 - value / ROOM) * 20

        level = int(scaled)

        pixels.fill(WHITE, brightness=level)
    else:
        pixels.fill(BLACK)
```

Mission 12 Complete

Welcome to Smart Lighting

This project has introduced you to an area with lots of potential for improving the world!

*Light Sensors and LED lights controlled with **code** can reduce **energy consumed** and make lighting more awesome!*

This code can help a lot of real-world applications:

- **Outdoor Lighting**
  - Street Lights, Parking lots, Home lighting
- **Stadium Lights**
  - Even controlling the light color so it looks better on camera
- **Indoor Lighting**
  - Sensing daylight from windows and skylights is called **Daylight Harvesting** - it saves energy!
  - That's exactly what your last Night Light code was doing!

Mission 13 - Sounds Fun

Picking Up Good Vibrations?

Previously you've played MP3 files on CodeX using the basic audio functions. But there's much more you can do with sound on this amazing device!

In this mission you'll dive deep into the soundlib module, and learn how to:

- Play sounds and music "in the background" while other code is running.
- Make sound effects for games and user feedback.
- Control the pitch and loop your sounds.

Get GUI!

Along the way you'll also make a professional-quality "Graphical User Interface" for the CodeX. Known as a GUI (pronounced gooey), the interactive user experience you'll design will be both familiar and exhilarating!

- Learning to craft your own GUI components is a major milestone in your coding journey!

Objective 1 - Race Day

Race Day

The big cycling race starts in just a few hours. Unfortunately the race officials have announced that their sound system is broken, so unless someone can provide an alternate plan they are going to cancel the event.

Nooooo! *You have to save the day!*

Your CodeX has lots of sound capabilities, and you can plug the output into a guitar amp to get the volume up.

Here's the list of **requirements** from the race officials, and a napkin-sketch they made when you met with them:

- The controller must have an easy-to-use User Interface (UI).
- Must loop the race *theme music* in the background, with PLAY/PAUSE control.
- A way to trigger the "START" sound effect
- Also need a "FINISH" sound.
- Finally, a "WARNING" siren effect is needed.

Check the 'Trek!

First step is to frame-up the UI.

- You'll expand your knowledge of the *display* bitmap functions as you create this!

CodeTrek:

```
1  from codex import *
2
3  def screen_layout():
```

Define a function to draw the basic layout of your screen.

- This will be all the "static" content, that doesn't change.

```
4      display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
5      display.draw_text("RACE CONTROLLER", x=35, y=8, color=BLACK, scale=2)
```

Use *text* and *boxes* to design your screen.

- See the **Hints** panel for details on the `draw_text()`, `draw_rect()`, and `fill_rect()` functions.

```
 6
 7      display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
 8      display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
 9      display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
10      display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)
11
12      display.draw_rect(0, 80, 240, 40, GRAY)
13      display.draw_rect(0, 120, 240, 40, GRAY)
14      display.draw_rect(0, 160, 240, 40, GRAY)
15      display.draw_rect(0, 200, 240, 40, GRAY)
16
17  # Highlight the first "menu selection"
18  # TODO: Dark blue rectangle 240 wide by 40 high, at y=80
```

AFTER your function,

.. and BEFORE drawing the `screen_layout()`,

Highlight a *selected* menu item.

- Draw a filled blue rectangle across the screen.
- y=80 will put it behind **MUSIC**, the first menu item.
- *Later* you will add code to let the user move the selection up and down!

Check the **Hints** panel for more info on that if needed.

```
19
20  screen_layout()
```

Don't forget to call your layout function!

- Notice it is called AFTER the highlight rectangle is drawn.
- That's so it draws *on top* of the highlight.

```
21
```

Hints:

- Use *text* and *boxes* to design your screen.
  - The following display functions are key.
  - Notice `draw_text()` doesn't do ANY scrolling, it simply puts the string exactly where you tell it!

```
# Place text string at exact location
draw_text(text, x, y, color, scale)

# Draw a box filled with color
fill_rect(x1, y1, width, height, color)

# Draw a box outline
draw_rect(x1, y1, width, height, color)
```

  The bitmap tool has more information, plus a link to the full docs.

- ## Your "menu highlight" is a solid blue rectangle

  See the bitmap toolbox help for more details about drawing on the screen.

```
display.fill_rect(0, 80, 240, 40, DARK_BLUE)
```

Goals:

- Create a **new** file named `Race_Control`
- Draw the basic screen layout for your RACE CONTROLLER.

  - The heading text must be "RACE CONTROLLER"
  - Menu options: "MUSIC", "START", "FINISH", "WARNING"
- Place a DARK_BLUE highlight box behind the "MUSIC" menu item.
  - This will make it appear to be the *selected* item.

Tools Found: Bitmap

Solution:

```
from codex import *

def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)


display.fill_rect(0, 80, 240, 40, DARK_BLUE)
screen_layout()

```

***Objective 2* - Scrolling Menu**

Getting Interactive

Your UI layout looks great! Now it's time to hook-in the UP/DOWN scrolling buttons, so the user can select different options.

- Right now the selection is stuck on **MUSIC**.
- Sure, music is great and all. *But They've gotta start the race sometime!*

⚠ Note ⚠

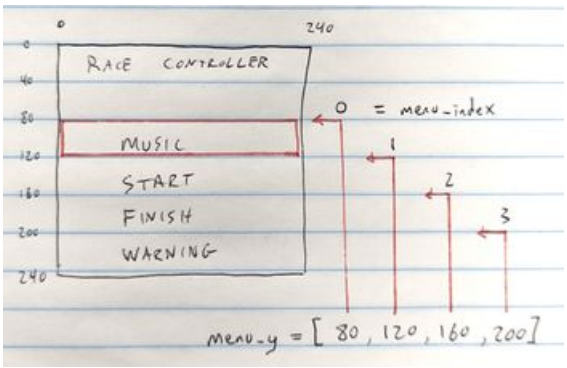
The code in the objective may NOT do what you expect! *Read Carefully!*

Menu 🔧List

Remember in the **Personal Billboard** mission you used a 🔧list to hold different items, and scrolled through the list with an **index** variable, `choice`.

You can do the same thing here!

- But this time your 🔧list will hold the *y-coordinates* where each *rectangle* needs to be drawn. (top left corner of rectangle)



And `menu_index` points to the selection:

```
# Example: Show START menu selected
menu_index = 1  # Point to START menu
y = menu_y[menu_index]  # 120
display.fill_rect(0, y, 240, 40, DARK_BLUE)
```

CodeTrek:

```
from codex import *

def screen_layout():
    """Draw static screen elements"""
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    """Update menu_index based on UI buttons"""
```

Also notice I've added more comments in the code.

- As your programs grow it's even more important to comment your ideas!

```
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0) # Keep index >= 0
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3) # Keep index <= 3
```

Define a function to check the U/D buttons.

- Update a global variable `menu_index` if U or D was pressed.
- Notice this does NOT "wrap" like you did in the *Personal Billboard* mission.
- Instead it uses the built-in functions `min()` and `max()` to stop increasing or decreasing past the limits.

So, the `menu_index` goes: 0→1→2→3 and 3→2→1→0

```

# Global variables
menu_index = 0
menu_y = [80, 120, 160, 200]
```

Y-position list of your 4 menu options.

- These y-axis values match the gray boxes drawn in `screen_layout()`.
- For example, the **MUSIC** menu selection is at `menu_y[0]`.
- You'll use `menu_index` to track the current selection.

```

# Main program loop
while True:

    # Remember the previous selection.
    prev_sel = menu_index

    # Update menu_index based on buttons.
    menu_buttons()

    # If menu_index changed, update screen.
    if menu_index != prev_sel:
```

```
40          # Update selected menu item.
41          display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
42
43          # Draw static layout.
44          screen_layout()
```

Main program loop

Continuously check the U/D buttons, and move the menu highlight as needed.

- Read each comment in this code, and be sure you understand how it works!

```
45
```

Hints:

- ## Read the Comments

  Start at the top of the file and read all the comments in the **CodeTrek**.

  - Follow along with the code. *Be the Computer!*

  Reading code takes time. You must go *slowly* to understand it.

- # Bug Out!

  This Objective can only be completed by hitting a "runtime error" in your code.

  - Don't try to fix the bug in the code presented in the CodeTrek.
  - You'll take care of it in the next Objective!

Goals:

- Define a function `menu_buttons()` that checks if `BTN_U` or `BTN_D` was pressed, and updates `menu_index`.
- Run the code and press **U** or **D** on your CodeX.
  - You will encounter an error!

Tools Found: list, Variables, Built-In Functions, Comments

Solution:

```
1  from codex import *
2
3
4  def screen_layout():
5      display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
6      display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)
7
8      display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
9      display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
10     display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
11     display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)
12
13     display.draw_rect(0, 80, 240, 40, GRAY)
14     display.draw_rect(0, 120, 240, 40, GRAY)
15     display.draw_rect(0, 160, 240, 40, GRAY)
16     display.draw_rect(0, 200, 240, 40, GRAY)
17
18 def menu_buttons():
19     if buttons.was_pressed(BTN_U):
20         menu_index = max(menu_index - 1, 0)
21     elif buttons.was_pressed(BTN_D):
22         menu_index = min(menu_index + 1, 3)
23
24 menu_index = 0
```

```
menu_y = [80, 120, 160, 200]

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel:
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
        screen_layout()

```

***Objective 3* - Going Global**

Bug Bashing

So what's up with this error message?

- *Local variable referenced before assignment*

Concept: *Local vs Global variables*

When you assign to a variable inside a function, Python assumes it's a local variable.

- A local variable is "private" to the function.
- It is a separate variable, even if it has the same name as another variable outside of the function.
- And it only exists while the function is running, so it can't hold its value between calls to the function.

Variables defined *outside* of functions are globals.

- A global variable exists for the entire life of your program.
- Inside a function you can read its value, but you *must* use the `global` statement if you want to change it.

So, *what happened?*

Python saw that your `menu_buttons()` function was *changing* the value of `menu_index`. So it made `menu_index` a local variable.

- Remember, a *local* variable doesn't exist until your function creates it by assigning to it.
- But your function first tried to *read* the value of `menu_index`, so it could add or subtract 1.
  - *...Before this **local** version even existed!*
- So that's what "Local variable referenced before assignment" means.

Use the `global` statement inside your function, so it can update the global `menu_index`.

CodeTrek:

```
from codex import *

def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
```

Simply add this global statement to inform Python that the function uses the `global` version of `menu_index`.

```
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
```

```
        menu_index = min(menu_index + 1, 3)

menu_index = 0
menu_y = [80, 120, 160, 200]

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel:
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
        screen_layout()

```

Hint:

- **The menus still don't quite work as expected?**

  That's okay, the next Objective will make it better!

Goal:

- Add the `global` statement to your `menu_buttons()` function.
  - Run the code: ***press U/D to test your menu!***

Tools Found: Assignment, Functions, Locals and Globals, Variables

Solution:

```
from codex import *


def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3)

menu_index = 0
menu_y = [80, 120, 160, 200]

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel:
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
        screen_layout()

```

Quiz 1 - Globals and Locals

Question 1: Which of the following is true?

✓ A variable which is *assigned to* inside of a function is considered **local**, unless it is explicitly named in a `global` statement..

❌ Any variable used inside of a function is considered **local** to that function, unless it is explicitly named in a `global` statement.

❌ A 🔧global variable is not visible inside a function, unless it is explicitly named in a `global` statement.

Question 2: The following function reports an error on Line 2.

```
1 def add_counter(amount, limit):
2     if counter < limit:
3         counter = counter + amount
```

What is the expected error message?

✔ Local variable referenced before assignment

❌ Global variable cannot be used in comparison.

❌ Undefined variable: counter

Objective 4 **- Covering Your Tracks**

Covering Your Tracks

Oh dear, your program has yet *another* problem. Your menu selection is leaving big footprints behind!

- Sometimes coding can feel like you're constantly moving from one bug to the next.
- Just remember, you are making progress every step of the way!
- *Enjoy the journey, my friend.*

You're already keeping track of the previous menu index. You just need to erase that area when changing to a new selection.

CodeTrek:

```
1  from codex import *
2
3
4  def screen_layout():
5      display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
6      display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)
7
8      display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
9      display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
10     display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
11     display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)
12
13     display.draw_rect(0, 80, 240, 40, GRAY)
14     display.draw_rect(0, 120, 240, 40, GRAY)
15     display.draw_rect(0, 160, 240, 40, GRAY)
16     display.draw_rect(0, 200, 240, 40, GRAY)
17
18 def menu_buttons():
19     global menu_index
20     if buttons.was_pressed(BTN_U):
21         menu_index = max(menu_index - 1, 0)
22     elif buttons.was_pressed(BTN_D):
23         menu_index = min(menu_index + 1, 3)
24
25 menu_index = 0
26 menu_y = [80, 120, 160, 200]
27
28 while True:
29     prev_sel = menu_index
30     menu_buttons()
31
32     # If menu_index changed, update screen.
33     if menu_index != prev_sel:
34         # Erase previous menu item
35         display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
```

```
36
37          # Update selected menu item.
38          display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
```

Erase the previous selected item by filling it with a BLACK background.

```
39
40          # Draw static layout.
41          screen_layout()
42
```

Goal:

- Add a line of code to erase the previous selection before moving to the next one.
    - *Press U/D to test your menu!*

Solution:

```
1  from codex import *
2
3
4  def screen_layout():
5      display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
6      display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)
7
8      display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
9      display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
10     display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
11     display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)
12
13     display.draw_rect(0, 80, 240, 40, GRAY)
14     display.draw_rect(0, 120, 240, 40, GRAY)
15     display.draw_rect(0, 160, 240, 40, GRAY)
16     display.draw_rect(0, 200, 240, 40, GRAY)
17
18 def menu_buttons():
19     global menu_index
20     if buttons.was_pressed(BTN_U):
21         menu_index = max(menu_index - 1, 0)
22     elif buttons.was_pressed(BTN_D):
23         menu_index = min(menu_index + 1, 3)
24
25 menu_index = 0
26 menu_y = [80, 120, 160, 200]
27
28 while True:
29     prev_sel = menu_index
30     menu_buttons()
31     if menu_index != prev_sel:
32         display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
33         display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
34         screen_layout()
35
```

Objective 5 **- Action**

Add Some Action!

Now that your menu-scrolling is on point, it's time to add a way for the user to trigger the selected **action**.

- For now just display a message when **Button A** is pressed.

Status Display

If you look closely at the napkin sketch of the UI in *Objective 1*, there was a "status area" just above the menu.

- You should put some text there for each menu action.

- *Soon you will add the sounds!*

CodeTrek:

```
from codex import *


def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3)

def show_status(msg):
    display.fill_rect(0, 30, 240, 50, BLACK)
    display.draw_text(msg, 10, 50, color=YELLOW, scale=2)
```

The `show_status(message)` function.

- Erase the background, then draw text!

```


def action_buttons():
    if buttons.was_pressed(BTN_A):
        if menu_index == 0:
            show_status("Start music...")
        elif menu_index == 1:
            show_status("Start race...")
        elif menu_index == 2:
            show_status("Finish race...")
        elif menu_index == 3:
            show_status("Warning sound...")
```

The `action_buttons()` function.

- When **A** is pressed, call `show_status()`
- The `menu_index` points to the items 0:MUSIC, 1:START, 2:FINISH, 3:WARNING.

```


menu_index = 0
menu_y = [80, 120, 160, 200]

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel:
        display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
        screen_layout()

    action_buttons()
```

Don't forget to call the `action_buttons()` function in your 🔧loop.

- That's how you detect when button **A** was pressed!

```
52
```

Goals:

- Define a new function `def show_status(message)` that displays a message in the *status area*.
  - Be sure to erase the background before drawing the message text.
- Define a new function `def action_buttons()` that checks for `BTN_A` and displays a status message based on the currently selected `menu_index`.
  - **Test your menu actions by scrolling U/D and pressing A for each item!**

Tools Found: Loops

Solution:

```
from codex import *


def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3)

def show_status(msg):
    display.fill_rect(0, 30, 240, 50, BLACK)
    display.draw_text(msg, 10, 50, color=YELLOW, scale=2)

def action_buttons():
    if buttons.was_pressed(BTN_A):
        if menu_index == 0:
            show_status("Start music...")
        elif menu_index == 1:
            show_status("Start race...")
        elif menu_index == 2:
            show_status("Finish race...")
        elif menu_index == 3:
            show_status("Warning sound...")

menu_index = 0
menu_y = [80, 120, 160, 200]

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel:
        display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
        screen_layout()

    action_buttons()

```

***Objective 6* - Initialization**

Make a Good First Impression

Your menu system is looking good, but it could be more *user-friendly.* A race official should be able to use this without any instructions!

Here are a couple of small problems you need to fix:

- The user has to press U or D before anything is displayed.
- There are no instructions about how to use this thing!

Check the 'Trek!

You're gonna need a 🔧variable to detect the first time the code runs through your main loop. It's pretty common to see 🔧global variables which are set during "program initialization", such as a 🔧boolean like `init = True`.

CodeTrek:

```
1  from codex import *
2
3
4  def screen_layout():
5      display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
6      display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)
7
8      display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
9      display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
10     display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
11     display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)
12
13     display.draw_rect(0, 80, 240, 40, GRAY)
14     display.draw_rect(0, 120, 240, 40, GRAY)
15     display.draw_rect(0, 160, 240, 40, GRAY)
16     display.draw_rect(0, 200, 240, 40, GRAY)
17
18 def menu_buttons():
19     global menu_index
20     if buttons.was_pressed(BTN_U):
21         menu_index = max(menu_index - 1, 0)
22     elif buttons.was_pressed(BTN_D):
23         menu_index = min(menu_index + 1, 3)
24
25 def show_status(msg):
26     display.fill_rect(0, 30, 240, 50, BLACK)
27     display.draw_text(msg, 10, 50, color=YELLOW, scale=2)
28
29 def action_buttons():
30     if buttons.was_pressed(BTN_A):
31         if menu_index == 0:
32             show_status("Start music...")
33         elif menu_index == 1:
34             show_status("Start race...")
35         elif menu_index == 2:
36             show_status("Finish race...")
37         elif menu_index == 3:
38             show_status("Warning sound...")
39
40 menu_index = 0
41 menu_y = [80, 120, 160, 200]
42 # TODO: initialize the 'init' variable
```

Create the `init` 🔧variable

You must add a line of code here!

- What should the value of `init` be at the start, `True` or `False`?
- Think through what happens the first time through the loop below...

```
43
44  show_status("Scroll=U/D Select=A")
```

Display the instructions.

- This is *before* your main loop, so it only runs once when the program first starts.

```
45
46  while True:
47      prev_sel = menu_index
48      menu_buttons()
49      if menu_index != prev_sel or init:
50          init = False
```

Draw the whole screen on the initial (`init`) run of the program.

- This simulates that first **U/D** button press you've been doing!
- See the 🔧logical operator `or` in the `if condition`?
- Be sure to set `init = False`, or this will run constantly and make the menu "flicker".

```
51          display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
52          display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
53          screen_layout()
54
55      action_buttons()
56
```

Goals:

- Call `show_status("...")` before your main program loop, to display **instructions** for operating the *Race Controller.*
- Draw the whole screen plus menu selection when the program first runs.
  - No more need to press **U** or **D** to show the initial screen!

Tools Found: Variables, Locals and Globals, bool, Logical Operators

Solution:

```
1  from codex import *
2
3
4  def screen_layout():
5      display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
6      display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)
7
8      display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
9      display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
10     display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
11     display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)
12
13     display.draw_rect(0, 80, 240, 40, GRAY)
14     display.draw_rect(0, 120, 240, 40, GRAY)
15     display.draw_rect(0, 160, 240, 40, GRAY)
16     display.draw_rect(0, 200, 240, 40, GRAY)
17
18 def menu_buttons():
19     global menu_index
20     if buttons.was_pressed(BTN_U):
21         menu_index = max(menu_index - 1, 0)
22     elif buttons.was_pressed(BTN_D):
23         menu_index = min(menu_index + 1, 3)
24
25 def show_status(msg):
26     display.fill_rect(0, 30, 240, 50, BLACK)
27     display.draw_text(msg, 10, 50, color=YELLOW, scale=2)
28
29 def action_buttons():
```

```
    if buttons.was_pressed(BTN_A):
        if menu_index == 0:
            show_status("Start music...")
        elif menu_index == 1:
            show_status("Start race...")
        elif menu_index == 2:
            show_status("Finish race...")
        elif menu_index == 3:
            show_status("Warning sound...")

menu_index = 0
menu_y = [80, 120, 160, 200]
init = True

show_status("Scroll=U/D Select=A")

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel or init:
        init = False
        display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
        screen_layout()

    action_buttons()

```

Objective 7 **- Start Race**

START!!

Now it's time to put some ACTION in your *menu actions!*

To start the race you're going to want a punchy, distinctive sound.

- Something like *Reveille on Bugle* - "Ta Ta-Ta-Taaaaa!"
- You've played MP3 files on the CodeX, but to craft your own sound effects like this you will need to create tones and change the pitch (frequency) directly with code!

Concept: *The soundlib Module*

Your CodeX has an awesome Python module for creating music and sound effects. The `soundmaker` object from the `soundlib` module has functions to create different types of tones, as well as playing recorded samples and MP3s.

Example: *Play a tone for 1.5 seconds*

```
from soundlib import *
from time import *

tone = soundmaker.get_tone('trumpet')
tone.set_pitch(880)
tone.play()
sleep(1.5)
tone.stop()
```

Ready to make some noise ?

CodeTrek:

```
from codex import *
from soundlib import *
from time import sleep
```

Oh yeah, don't forget your `import`s!

- Gotta have the `soundlib` 🔧module so you can use `soundmaker`.
- And your sound effects use `sleep()` for timing :-)

```
4
5  def screen_layout():
6      display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
7      display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)
8
9      display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
10     display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
11     display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
12     display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)
13
14     display.draw_rect(0, 80, 240, 40, GRAY)
15     display.draw_rect(0, 120, 240, 40, GRAY)
16     display.draw_rect(0, 160, 240, 40, GRAY)
17     display.draw_rect(0, 200, 240, 40, GRAY)
18
19 def menu_buttons():
20     global menu_index
21     if buttons.was_pressed(BTN_U):
22         menu_index = max(menu_index - 1, 0)
23     elif buttons.was_pressed(BTN_D):
24         menu_index = min(menu_index + 1, 3)
25
26 def show_status(msg):
27     display.fill_rect(0, 30, 240, 50, BLACK)
28     display.draw_text(msg, 10, 50, color=YELLOW, scale=2)
29
30 def start_race():
31     show_status("Start race...")
32     trumpet.set_pitch(440)
33     trumpet.play()
```

Define the `start_race()` function.

- Keep it simple for this first test.
- *You'll be able to customize the sound to your personal style later!*

```
34     sleep(0.1)
35     trumpet.stop()
36     sleep(0.1)
37     trumpet.set_pitch(880)
38     trumpet.play()
39     sleep(0.2)
40     trumpet.stop()
41
42 def action_buttons():
43     if buttons.was_pressed(BTN_A):
44         if menu_index == 0:
45             show_status("Start music...")
46         elif menu_index == 1:
47             start_race()
```

The **START** action will happen in a new 🔧function.

- Replace the status message with a call to the `start_race()` function!

```
48         elif menu_index == 2:
49             show_status("Finish race...")
50         elif menu_index == 3:
51             show_status("Warning sound...")
52
53 menu_index = 0
54 menu_y = [80, 120, 160, 200]
55 init = True
56 show_status("Scroll=U/D Select=A")
57
58 trumpet = soundmaker.get_tone('trumpet')
```

Get a tone from the `soundmaker`.

- Each call to `get_tone()` gets a new sound that can be played independently.
- You can have up to 16 tones playing at the same time!

- But for the **START** sound you only need one tone.
- Get it here as a *global* variable, since we only need to call `get_tone()` once.

```
59
60  while True:
61      prev_sel = menu_index
62      menu_buttons()
63      if menu_index != prev_sel or init:
64          init = False
65          display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
66          display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
67          screen_layout()
68
69      action_buttons()
70
```

Goals:

- Define a `start_race()` 🔧function.
- Call the `start_race()` function from inside `action_buttons()`.
- Don't forget to `import` the 🔧modules for *sound* and *timing*.
- Get a *tone* from `soundmaker`

Tools Found: Functions, import

Solution:

```
1  from codex import *
2  from soundlib import *
3  from time import sleep
4
5  def screen_layout():
6      display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
7      display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)
8
9      display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
10     display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
11     display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
12     display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)
13
14     display.draw_rect(0, 80, 240, 40, GRAY)
15     display.draw_rect(0, 120, 240, 40, GRAY)
16     display.draw_rect(0, 160, 240, 40, GRAY)
17     display.draw_rect(0, 200, 240, 40, GRAY)
18
19 def menu_buttons():
20     global menu_index
21     if buttons.was_pressed(BTN_U):
22         menu_index = max(menu_index - 1, 0)
23     elif buttons.was_pressed(BTN_D):
24         menu_index = min(menu_index + 1, 3)
25
26 def show_status(msg):
27     display.fill_rect(0, 30, 240, 50, BLACK)
28     display.draw_text(msg, 10, 50, color=YELLOW, scale=2)
29
30 def start_race():
31     show_status("Start race...")
32     trumpet.set_pitch(440)
33     trumpet.play()
34     sleep(0.1)
35     trumpet.stop()
36     sleep(0.1)
37     trumpet.set_pitch(880)
38     trumpet.play()
39     sleep(0.2)
40     trumpet.stop()
41
```

```
def action_buttons():
    if buttons.was_pressed(BTN_A):
        if menu_index == 0:
            show_status("Start music...")
        elif menu_index == 1:
            start_race()
        elif menu_index == 2:
            show_status("Finish race...")
        elif menu_index == 3:
            show_status("Warning sound...")

menu_index = 0
menu_y = [80, 120, 160, 200]
init = True
show_status("Scroll=U/D Select=A")

trumpet = soundmaker.get_tone('trumpet')

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel or init:
        init = False
        display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
        screen_layout()

    action_buttons()

```

***Objective 8* - Fanfare**

More Fanfare!

Your **START** sound could use a bit more "excitement".

- Repeating the first tone several times would be a good lead-in for starting the race.
- And you know how to do that without copying and pasting that code a bunch of times, right?

  You need to use a loop !

But this time you can make the code even simpler...

Concept: *for loop*

The `for` loop is made for looping across a range of numbers, or iterating over other kinds of sequences like lists.

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.

Get creative!

Feel free to experiment with looping notes until it sounds good to you.

CodeTrek:

```
from codex import *
from soundlib import *
from time import sleep

def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
```

```
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3)

def show_status(msg):
    display.fill_rect(0, 30, 240, 50, BLACK)
    display.draw_text(msg, 10, 50, color=YELLOW, scale=2)

def start_race():
    show_status("Start race...")
    for i in range(4):
        trumpet.set_pitch(440)
        trumpet.play()
        sleep(0.1)
        trumpet.stop()
        sleep(0.1)
    trumpet.set_pitch(880)
```

Add a `for loop` to your jam!

- Try it as shown first, then feel free to get creative...

Notice, you can just *select* the block of code you already had, and 🔧indent it beneath the `for i in range(4):` to make it loop 4 times!

```
    trumpet.play()
    sleep(0.2)
    trumpet.stop()

def action_buttons():
    if buttons.was_pressed(BTN_A):
        if menu_index == 0:
            show_status("Start music...")
        elif menu_index == 1:
            start_race()
        elif menu_index == 2:
            show_status("Finish race...")
        elif menu_index == 3:
            show_status("Warning sound...")

menu_index = 0
menu_y = [80, 120, 160, 200]
init = True
show_status("Scroll=U/D Select=A")

trumpet = soundmaker.get_tone('trumpet')

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel or init:
        init = False
        display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
        screen_layout()

    action_buttons()

```

Goal:

- Add a `for loop` to your `start_race()` function.

- *And play some sounds inside the loop!*

Tools Found: Loops, Iterable, list, Ranges, Variables, Indentation

Solution:

```
from codex import *
from soundlib import *
from time import sleep

def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3)

def show_status(msg):
    display.fill_rect(0, 30, 240, 50, BLACK)
    display.draw_text(msg, 10, 50, color=YELLOW, scale=2)

def start_race():
    show_status("Start race...")
    for i in range(4):
        trumpet.set_pitch(440)
        trumpet.play()
        sleep(0.1)
        trumpet.stop()
        sleep(0.1)
    trumpet.set_pitch(880)
    trumpet.play()
    sleep(0.2)
    trumpet.stop()

def action_buttons():
    if buttons.was_pressed(BTN_A):
        if menu_index == 0:
            show_status("Start music...")
        elif menu_index == 1:
            start_race()
        elif menu_index == 2:
            show_status("Finish race...")
        elif menu_index == 3:
            show_status("Warning sound...")

menu_index = 0
menu_y = [80, 120, 160, 200]
init = True
show_status("Scroll=U/D Select=A")

trumpet = soundmaker.get_tone('trumpet')

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel or init:
        init = False
        display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
```

```
68          screen_layout()
69
70      action_buttons()
71
```

Quiz 2 - For Loops

Question 1: What is displayed by the following code?

```
for i in range(4):
    display.print(i, end=',')
```

✓ 0,1,2,3

✗ 1,2,3,4

✗ 0,1,2,3,4

Question 2: What is displayed by the following code?

```
for i in range(1, 5):
    display.print(i, end=',')
```

✓ 1,2,3,4

✗ 1,2,3,4,5

✗ 0,1,2,3,4,5

Objective 9 - Music

Make Some Music!

Moving on to the MUSIC menu option.

- Change your code to start and stop the music when the MUSIC menu is selected.

Playing an MP3 in the Background

With the `soundlib` module, you have a *new* way to play MP3 files:

Example:

```
# Get an MP3 song object. Returns immediately (non-blocking)
song = soundmaker.get_mp3("sounds/roll")
```

Concept: *Blocking vs. Non-Blocking Functions*

The advantage of using `soundmaker` for MP3s is that it doesn't make your code wait for the sound to finish playing.

- Functions that block your code from continuing until they finish are called ***blocking*** functions.
- The `soundmaker` functions are ***non-blocking***. Your code can `start()` and `stop()` a sound, and the sound keeps playing while your code runs.

Toggling On/Off

When the user selects **MUSIC** on the menu, it should either turn the music ON or OFF.

- Depends on whether the music already `is_playing`, right?
- So your code needs a 🔧variable for the current state `is_playing`, `True` or `False`.
- And when they press the MUSIC button, the state should flip:
  - `True` → `False` ...or `False` → `True`.

That's exactly what the `not` logical operator does. You will often use this operator when you need to toggle a bool value!

CodeTrek:

```
from codex import *
from soundlib import *
from time import sleep

def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3)

def show_status(msg):
    display.fill_rect(0, 30, 240, 50, BLACK)
    display.draw_text(msg, 10, 50, color=YELLOW, scale=2)

def start_race():
    show_status("Start race...")
    for i in range(4):
        trumpet.set_pitch(440)
        trumpet.play()
        sleep(0.1)
        trumpet.stop()
        sleep(0.1)
    trumpet.set_pitch(880)
    trumpet.play()
    sleep(0.2)
    trumpet.stop()

def toggle_music():
    #TODO: Add a statement so Python knows is_playing is a global variable.
    is_playing = not is_playing
    if is_playing:
        show_status("Started music...")
        music_track.play(loop=True)
    else:
        show_status("Stopped music!")
        music_track.stop()
```

Define the `toggle_music()` function.

- Check out the cool `not` action here :-)
- Oh, and mind the *`#TODO`* comment!
  - You might get a *familiar* error if you don't.
  - `is_playing` should be global, eh?

```

def action_buttons():
    if buttons.was_pressed(BTN_A):
        if menu_index == 0:
            toggle_music()
```

Call your new `toggle_music()` function to turn the tunes ON/OFF.

```
57          elif menu_index == 1:
58              start_race()
59          elif menu_index == 2:
60              show_status("Finish race...")
61          elif menu_index == 3:
62              show_status("Warning sound...")
63
64  menu_index = 0
65  menu_y = [80, 120, 160, 200]
66  init = True
67  show_status("Scroll=U/D Select=A")
68
69  trumpet = soundmaker.get_tone('trumpet')
70  music_track = soundmaker.get_mp3('sounds/funk', play=False)
71  is_playing = False
```

Add some 🔧global *music* data.

- First you need a song to play. I've chosen *"funk.mp3"* as an inspiring selection.
- Next, create a 🔧bool variable to hold the state `is_playing`.
  - This is your *Race Controller's* memory of whether it's currently playing MUSIC or not!

```
72
73  while True:
74      prev_sel = menu_index
75      menu_buttons()
76      if menu_index != prev_sel or init:
77          init = False
78          display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
79          display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
80          screen_layout()
81
82      action_buttons()
83
```

Hints:

- **Getting an error?**
  - Check the **CodeTrek**
  - You'll need a 🔧global declaration in your `toggle_music()` function.
- **Music Starts Automatically?**

  Does your MP3 start playing right away?

  - Look closely at the `get_mp3()` 🔧arguments below. The `play=False` is the secret to loading the song in the *stopped* mode.

    ```
    music_track = soundmaker.get_mp3('sounds/funk', play=False)
    ```

Goals:

- Get an MP3 file to play using the `soundmaker` object.
- Define a new function `def toggle_music()` that toggles the music ON and OFF.
  - It should use `not` to flip the state of a 🔧global `is_playing` 🔧variable `True / False`
- Call your new `toggle_music()` function from `action_buttons()`.
  - When the user selects **MUSIC** and presses **A** the music should turn ON or OFF.

Tools Found: Variables, Logical Operators, bool, Locals and Globals

Solution:

```
1  from codex import *
2  from soundlib import *
```

```
from time import sleep

def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3)

def show_status(msg):
    display.fill_rect(0, 30, 240, 50, BLACK)
    display.draw_text(msg, 10, 50, color=YELLOW, scale=2)

def start_race():
    show_status("Start race...")
    for i in range(4):
        trumpet.set_pitch(440)
        trumpet.play()
        sleep(0.1)
        trumpet.stop()
        sleep(0.1)
    trumpet.set_pitch(880)
    trumpet.play()
    sleep(0.2)
    trumpet.stop()

def toggle_music():
    global is_playing
    is_playing = not is_playing
    if is_playing:
        show_status("Started music...")
        music_track.play(loop=True)
    else:
        show_status("Stopped music!")
        music_track.stop()

def action_buttons():
    if buttons.was_pressed(BTN_A):
        if menu_index == 0:
            toggle_music()
        elif menu_index == 1:
            start_race()
        elif menu_index == 2:
            show_status("Finish race...")
        elif menu_index == 3:
            show_status("Warning sound...")

menu_index = 0
menu_y = [80, 120, 160, 200]
init = True
show_status("Scroll=U/D Select=A")

trumpet = soundmaker.get_tone('trumpet')
music_track = soundmaker.get_mp3('sounds/funk', play=False)
is_playing = False

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel or init:
        init = False
```

```
78          display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
79          display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
80          screen_layout()
81
82      action_buttons()
83
```

Quiz 3 - Blocking and Toggling

Question 1: What is a **blocking** function?

- ✓ A function that blocks program execution and does not return until it has completed its goal.
- ✗ A function that creates square 2D shapes or 3D cubes.
- ✗ A function that returns immediately, even if that would block it from completing its goal.
- ✗ A function composed of blocks.

Question 2: `sleep(5)` will delay the program for 5 seconds.

Is it a "blocking function"?

- ✓ Yes
- ✗ No

Question 3: What is the final value of `toggle` after this code runs?

```
toggle = False

toggle = not toggle
toggle = not toggle
toggle = not toggle
```

- ✓ True
- ✗ False
- ✗ 3

Objective 10 - Finish Race

The Sound of Victory

It's time for you to craft another sound.

- This one will be played when a racer crosses the finish line.
- It should be exciting and inspiring - a celebration of the effort!

Your START sound used a `for` 🔧loop to repeat a tone. Now rather than repeating the same tone, try changing the frequency inside your loop using the `set_pitch()` function.

The CodeTrek will lead you to a pretty cool sound. But I bet you can do better. *Make it your own!*

CodeTrek:

```
1  from codex import *
2  from soundlib import *
3  from time import sleep
4
5  def screen_layout():
6      display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
7      display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)
8
```

```
    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3)

def show_status(msg):
    display.fill_rect(0, 30, 240, 50, BLACK)
    display.draw_text(msg, 10, 50, color=YELLOW, scale=2)

def start_race():
    show_status("Start race...")
    for i in range(4):
        trumpet.set_pitch(440)
        trumpet.play()
        sleep(0.1)
        trumpet.stop()
        sleep(0.1)
    trumpet.set_pitch(880)
    trumpet.play()
    sleep(0.2)
    trumpet.stop()

def toggle_music():
    global is_playing
    is_playing = not is_playing
    if is_playing:
        show_status("Started music...")
        music_track.play(loop=True)
    else:
        show_status("Stopped music!")
        music_track.stop()

def finish_race():
    show_status("Finish race...")
    trumpet.play()

    for freq1 in range(500, 1500, 100):
        for freq2 in range(freq1, freq1+1000, 100):
            trumpet.set_pitch(freq2)
            sleep(0.023)
```

Sweep the sound *low* to *high*, from *bass* to *treble!*

- The `set_pitch(freq)` function changes the frequency of a tone.

Here I am using *two* `for` `loops`.

- The second loop is *nested* inside the first one.
- Every time through the *outer* loop, the *inner* loop completes ALL its cycles.

Both of these loops step the frequency by 100 Hz each time around the loop, using `range(start, stop, step)`.

- See the 🔧range docs for more details on that!

```

    for repeats in range(10):
            trumpet.play()
            sleep(0.1)
            trumpet.stop()
            sleep(0.05)

    trumpet.stop()
```

```
69
70  def action_buttons():
71      if buttons.was_pressed(BTN_A):
72          if menu_index == 0:
73              toggle_music()
74          elif menu_index == 1:
75              start_race()
76          elif menu_index == 2:
77              finish_race()
78          elif menu_index == 3:
79              show_status("Warning sound...")
80
81  menu_index = 0
82  menu_y = [80, 120, 160, 200]
83  init = True
84  show_status("Scroll=U/D Select=A")
85
86  trumpet = soundmaker.get_tone('trumpet')
87  music_track = soundmaker.get_mp3('sounds/funk', play=False)
88  is_playing = False
89
90  while True:
91      prev_sel = menu_index
92      menu_buttons()
93      if menu_index != prev_sel or init:
94          init = False
95          display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
96          display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
97          screen_layout()
98
99      action_buttons()
100
```

Goals:

- Define a new function `def finish_race()` that plays your **FINISH** sound.
- Call your `finish_race()` function when the user activates the **FINISH** menu item.

Tools Found: Loops, Ranges

Solution:

```
1  from codex import *
2  from soundlib import *
3  from time import sleep
4
5  def screen_layout():
6      display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
7      display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)
8
9      display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
10     display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
11     display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
12     display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)
13
14     display.draw_rect(0, 80, 240, 40, GRAY)
15     display.draw_rect(0, 120, 240, 40, GRAY)
16     display.draw_rect(0, 160, 240, 40, GRAY)
17     display.draw_rect(0, 200, 240, 40, GRAY)
18
19 def menu_buttons():
20     global menu_index
21     if buttons.was_pressed(BTN_U):
22         menu_index = max(menu_index - 1, 0)
23     elif buttons.was_pressed(BTN_D):
24         menu_index = min(menu_index + 1, 3)
25
26 def show_status(msg):
27     display.fill_rect(0, 30, 240, 50, BLACK)
28     display.draw_text(msg, 10, 50, color=YELLOW, scale=2)
```

```
29
30  def start_race():
31      show_status("Start race...")
32      for i in range(4):
33          trumpet.set_pitch(440)
34          trumpet.play()
35          sleep(0.1)
36          trumpet.stop()
37          sleep(0.1)
38      trumpet.set_pitch(880)
39      trumpet.play()
40      sleep(0.2)
41      trumpet.stop()
42
43  def toggle_music():
44      global is_playing
45      is_playing = not is_playing
46      if is_playing:
47          show_status("Started music...")
48          music_track.play(loop=True)
49      else:
50          show_status("Stopped music!")
51          music_track.stop()
52
53  def finish_race():
54      show_status("Finish race...")
55      trumpet.play()
56
57      for freq1 in range(500, 1500, 100):
58          for freq2 in range(freq1, freq1+1000, 100):
59              trumpet.set_pitch(freq2)
60              sleep(0.023)
61
62      for repeats in range(10):
63              trumpet.play()
64              sleep(0.1)
65              trumpet.stop()
66              sleep(0.05)
67
68      trumpet.stop()
69
70  def action_buttons():
71      if buttons.was_pressed(BTN_A):
72          if menu_index == 0:
73              toggle_music()
74          elif menu_index == 1:
75              start_race()
76          elif menu_index == 2:
77              finish_race()
78          elif menu_index == 3:
79              show_status("Warning sound...")
80
81  menu_index = 0
82  menu_y = [80, 120, 160, 200]
83  init = True
84  show_status("Scroll=U/D Select=A")
85
86  trumpet = soundmaker.get_tone('trumpet')
87  music_track = soundmaker.get_mp3('sounds/funk', play=False)
88  is_playing = False
89
90  while True:
91      prev_sel = menu_index
92      menu_buttons()
93      if menu_index != prev_sel or init:
94          init = False
95          display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
96          display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
97          screen_layout()
98
99      action_buttons()
100
```

***Objective 11* - Warning Siren**

Warning Alert

If there's a problem on the race course, or some danger to warn the riders about, how will the race officials get their attention?

- Your mission is to create a sound effect that makes people take notice!

New soundlib Feature: `glide()`

The CodeTrek will introduce you to another feature of the `soundmaker` Tone object.

- The `glide(new_pitch, duration)` function is an easy way to ramp the pitch from the current setting to a new setting over a specified amount of time.
- You could achieve the same thing with a loop
  - *...but it's nice to let the library do the work for you!*
- Plus, it's non-blocking. So if you wanted to, you could code some flashing lights and stuff while the sound glides on!

CodeTrek:

```
from codex import *
from soundlib import *
from time import sleep

def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3)

def show_status(msg):
    display.fill_rect(0, 30, 240, 50, BLACK)
    display.draw_text(msg, 10, 50, color=YELLOW, scale=2)

def start_race():
    show_status("Start race...")
    for i in range(4):
        trumpet.set_pitch(440)
        trumpet.play()
        sleep(0.1)
        trumpet.stop()
        sleep(0.1)
    trumpet.set_pitch(880)
    trumpet.play()
    sleep(0.2)
    trumpet.stop()

def toggle_music():
    global is_playing
    is_playing = not is_playing
    if is_playing:
        show_status("Started music...")
        music_track.play(loop=True)
    else:
        show_status("Stopped music!")
        music_track.stop()

def finish_race():
    show_status("Finish race...")
```

```
    trumpet.play()

    for freq1 in range(500, 1500, 100):
        for freq2 in range(freq1, freq1+1000, 100):
            trumpet.set_pitch(freq2)
            sleep(0.023)

    for repeats in range(10):
            trumpet.play()
            sleep(0.1)
            trumpet.stop()
            sleep(0.05)

    trumpet.stop()

def warning():
    show_status("Warning sound...")
    siren.set_pitch(440)
    siren.play()
    siren.glide(880, 1.5)
```

A *siren* sound.

The *glide* function takes 2 arguments: `glide(new_pitch, duration)`

- `new_pitch` is the target frequency the tone will ramp to.
- `duration` is how many seconds it will take to get there.

Here I'm just ramping it up for 1.5 seconds, then back down for 1.5 seconds.

- Notice I had to `sleep()` while it's "gliding".
- The `glide()` function is non-blocking, so it returns immediately!
- You know, in case you want to also kick-off a few other tones gliding and whatnot...

```
    sleep(1.5)
    siren.glide(440, 1.5)
    sleep(1.5)
    siren.stop()

def action_buttons():
    if buttons.was_pressed(BTN_A):
        if menu_index == 0:
            toggle_music()
        elif menu_index == 1:
            start_race()
        elif menu_index == 2:
            finish_race()
        elif menu_index == 3:
            warning()
```

Call your new `warning()` function when the user activates the **WARNING** menu item.

```

menu_index = 0
menu_y = [80, 120, 160, 200]
init = True
show_status("Scroll=U/D Select=A")

trumpet = soundmaker.get_tone('trumpet')
music_track = soundmaker.get_mp3('sounds/funk', play=False)
is_playing = False
siren = soundmaker.get_tone('violin')
```

Define a new global sound for your WARNING *siren*.

- You can experiment with different sounds from soundlib, but I like the "violin" for this.

```

while True:
    prev_sel = menu_index
    menu_buttons()
```

```
    if menu_index != prev_sel or init:
        init = False
        display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
        screen_layout()

    action_buttons()

```

Goals:

- Define a new function `def warning()` that makes an *"alarming"* sound.
- Call your new `warning()` function when the user activates the **WARNING** menu item.

Tools Found: soundlib, Loops, import, Keyword and Positional Arguments

Solution:

```
from codex import *
from soundlib import *
from time import sleep

def screen_layout():
    display.fill_rect(0, 0, 240, 30, LIGHT_GRAY)
    display.draw_text("RACE CONTROLLER", 35, 8, color=BLACK, scale=2)

    display.draw_text("MUSIC", x=20, y=90, color=WHITE, scale=3)
    display.draw_text("START", x=20, y=130, color=WHITE, scale=3)
    display.draw_text("FINISH", x=20, y=170, color=WHITE, scale=3)
    display.draw_text("WARNING", x=20, y=210, color=WHITE, scale=3)

    display.draw_rect(0, 80, 240, 40, GRAY)
    display.draw_rect(0, 120, 240, 40, GRAY)
    display.draw_rect(0, 160, 240, 40, GRAY)
    display.draw_rect(0, 200, 240, 40, GRAY)

def menu_buttons():
    global menu_index
    if buttons.was_pressed(BTN_U):
        menu_index = max(menu_index - 1, 0)
    elif buttons.was_pressed(BTN_D):
        menu_index = min(menu_index + 1, 3)

def show_status(msg):
    display.fill_rect(0, 30, 240, 50, BLACK)
    display.draw_text(msg, 10, 50, color=YELLOW, scale=2)

def start_race():
    show_status("Start race...")
    for i in range(4):
        trumpet.set_pitch(440)
        trumpet.play()
        sleep(0.1)
        trumpet.stop()
        sleep(0.1)
    trumpet.set_pitch(880)
    trumpet.play()
    sleep(0.2)
    trumpet.stop()

def toggle_music():
    global is_playing
    is_playing = not is_playing
    if is_playing:
        show_status("Started music...")
        music_track.play(loop=True)
    else:
        show_status("Stopped music!")
        music_track.stop()

```

```
def finish_race():
    show_status("Finish race...")
    trumpet.play()

    for freq1 in range(500, 1500, 100):
        for freq2 in range(freq1, freq1+1000, 100):
            trumpet.set_pitch(freq2)
            sleep(0.023)

    for repeats in range(10):
            trumpet.play()
            sleep(0.1)
            trumpet.stop()
            sleep(0.05)

    trumpet.stop()

def warning():
    show_status("Warning sound...")
    siren.set_pitch(440)
    siren.play()
    siren.glide(880, 1.5)
    sleep(1.5)
    siren.glide(440, 1.5)
    sleep(1.5)
    siren.stop()

def action_buttons():
    if buttons.was_pressed(BTN_A):
        if menu_index == 0:
            toggle_music()
        elif menu_index == 1:
            start_race()
        elif menu_index == 2:
            finish_race()
        elif menu_index == 3:
            warning()

menu_index = 0
menu_y = [80, 120, 160, 200]
init = True
show_status("Scroll=U/D Select=A")

trumpet = soundmaker.get_tone('trumpet')
music_track = soundmaker.get_mp3('sounds/funk', play=False)
is_playing = False
siren = soundmaker.get_tone('violin')

while True:
    prev_sel = menu_index
    menu_buttons()
    if menu_index != prev_sel or init:
        init = False
        display.fill_rect(0, menu_y[prev_sel], 240, 40, BLACK)
        display.fill_rect(0, menu_y[menu_index], 240, 40, DARK_BLUE)
        screen_layout()

    action_buttons()

```

Mission 13 Complete

SOUNDS Like a Plan!

Excellent work!!

You've saved the day at the cycling event, AND gained some useful *audio engineering* skills.

Plus you made an *amazing UI* that could be adapted to a LOT of other applications!

RIDE ON !

Mission 14 **- Line Art**

Digital Artistry

This mission will lead you on a journey to discover the magic of computer graphics:

> *Making beautiful visual art with just a few lines of code!*

Pixel Power

It all starts with a pixel drawn on the screen. But as you've seen, things get much more interesting when you loop your code to create patterns of logic, sounds, and **light!**

`for` the Win!

As you complete this mission you'll gain a mastery of the `for` loop, a *versatile* tool to have in your coding arsenal.

- Ready to *visualize* a range of colorful pixels streaming across your LCD screen?

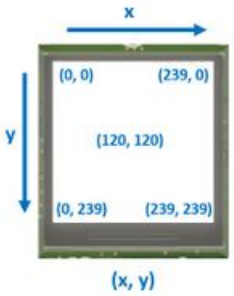
Objective 1 **- Pixel Power**

Pixel Power

You've already drawn on the screen using the bitmap functions, like `display.draw_rect()` and friends.

- **Rectangles** and **circles** - outlined and filled - are the basic shapes in that API.
- But there'a a more fundamental particle of graphics goodness, that those other shapes are made from.

```
# Draw a single RED pixel at x=10, y=10
display.set_pixel(10, 10, RED)
```



The *CodeX* *LCD* contains 240x240 pixels.

- That's 240 pixels from left to right: x=0 → x=239
- And 240 pixels from top to bottom: y=0 → y=239

Reading Pixels

Your code can *write* pixel colors to the display, but it can also *read* them back!

```
# Read the color of the pixel at x=10, y=10
c = display.get_pixel(10, 10)
print("Color = ", c)  # Print to the console
```

To the Console!

Remember the debug console? Click the ☰ at the bottom-right to open it up.

- That's where you *inspected* *variables* before.
- But you can also ***print messages*** there!
- This will be *very* useful since your LCD is now dedicated to *ART!*

Create a New File!

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

Check the 'Trek!

CodeTrek:

```
from codex import *

# Draw a white pixel on the screen
display.set_pixel(120, 120, WHITE)
```

Just a single bright point in the middle of your screen!

```


# Draw six pixels with different locations and colors
display.set_pixel(212, 58, RED)
# TODO: Light up at least five more pixels
```

Get Creative

Scatter some brilliant RGB colors across the screen.

```


# Get and print one of the pixel colors
my_pix = display.get_pixel(212, 58)
print ('Pixel:', my_pix)
```

"Farmboy, fetch me that *Pixel?"*

- As you wish.

(Just be sure the x,y location is one you lit up above!)

```

```

Goals:

- **First pixel**

  Draw a pixel on the screen with `display.set_pixel()` at x=120 and y=12 0.

  - Use a bright color like `YELLOW` or `WHITE`. Try any of the built-in RGB Colors, or define your own tuple!

- **Pixel *MIX*el!**

  Draw 6 more pixels at different locations on the screen with different colors.

  - Try to figure out where they should appear on the screen before you run the code.
  - *Are they where you expected?*

- **Reading Pixels**

  Use `display.get_pixel()` to read the RGB Color value of one of the pixels you drew.

  - Use `print()` to display the tuple value in the console window.

Tools Found: Bitmap, API, LCD, Print Function, Variables, RGB Colors, tuple

Solution:

```
from codex import *

# Draw a white pixel on the screen
display.set_pixel(120, 120, WHITE)

# Draw six pixels at different locations with different colors (constants)
display.set_pixel(212, 58, RED)
display.set_pixel(210, 62, GREEN)
display.set_pixel(219, 53, BLUE)
display.set_pixel(202, 47, YELLOW)
```

```
11  display.set_pixel(195, 42, CYAN)
12  display.set_pixel(222, 64, MAGENTA)
13
14  # Get one of the pixel colors
15  print ('Pixel:', display.get_pixel(212, 58))
16
```

Quiz 1 **- Pixel Basics**

Answer the questions below about the following code:

```
display.set_pixel(200, 200, CYAN)
c = display.get_pixel(200, 200)
print("Color = ", c)
```

Question 1: Where on the CodeX 🔧display does the CYAN pixel appear?

✓ Lower Right

✗ Upper Left

✗ Upper Right

✗ Lower Left

Question 2: What is printed on the 🔧console?

- You either need to consult the 🔧RGB Colors documentation, or just try it for yourself!

✓ `Color = (0, 252, 248)`

✗ `Color = (0, 0, 0)`

✗ `Color = (0, 255, 255)`

✗ `Color = (144, 210, 48)`

Objective 2 **- Line Up**

Line Up!

Now that you've mastered 🔧pixels...

- Seriously, you pretty much know all there is to know about them!

What do you call a bunch of pixels in a row?

wait for it...

A LINE!

So what are you waiting for?

- The 🔧LCD is 240 pixels wide.
- Just copy-and-paste that `display.set_pixel()` 240 times, right?

Don't You Dare!

This is what 🔧loops are *made* for!

Modify Your Code

Delete all those `set_pixel()` lines, and replace them with a 🔧loop.

Check the 'Trek!

A single *lovely* `for` loop is all you need to acheive the goal below!

CodeTrek:

```
from codex import *

# Red line
for x in range(???):  # TODO
    display.set_pixel(x, ???, RED)  # TODO
```

Fix Those TODO's

1. The 🔧range of `x` is the *width* of the screen. How many pixels wide is it?
2. Mark the center line at a **y** value of `120`

Goal:

- Draw a horizontal RED line across the screen.
  - Span the full width: x=0 → x=239
  - The y-coordinate should be constant y=120

Tools Found: Pixel, LCD, Loops, Ranges

Solution:

```
from codex import *

# Red line
for x in range(240):
    display.set_pixel(x, 120, RED)
```

***Objective 3* - Two Axes to Grind**

Add a Vertical Axis

You have a nice *horizontal* line. Adding a *vertical* line to match will create a perfect reference for drawing additional "line art" in this mission.

Getting Centered

Your code currently uses "magic numbers" like 240 for the display width, and 120 for the center.

- Numbers that just appear in the code with no explanation *must* be magic... but not the good kind.
- Other programmers trying to understand your code might have no idea what they mean.
- And when things change in the future, like getting a bigger display for example, the magic goes "poof!"

You can eliminate the magic numbers here by getting the display *width* and *height* straight from the source - the display 🔧bitmap itself.

```
# display is a bitmap, and every bitmap has width and height properties
w = display.width
h = display.height
```

Now it's obvious what `w` and `h` are!

- *Look mom, no magic!*

⚠Heads Up!⚠

You'll get an ERROR when this code runs.

- Fear not. *Proceed to the next Objective to get it sorted out...*

CodeTrek:

```
from codex import *

# Use half display width and half height to find
# the center of the screen.
x_center = display.width / 2
y_center = display.height / 2
```

Calculate the *center* by dividing the screen in half!

```

# Draw a horizontal line of pixels in the center
# of the screen.
for x in range(display.width):
    display.set_pixel(x, y_center, RED)
```

Horizontal line

Just modify your *line loop* to use `display.width` and the new `y_center` 🔧variable.

```

# Draw a vertical line of pixels in the center of the screen
for y in range(display.height):
    display.set_pixel(x_center, y, RED)
```

Vertical line

Add a new loop, just like your *horizontal* one above.

- Except here the `y` value 🔧ranges over `display.height`
- And the `x_center` is constant.

```

```

Goals:

- Eliminate Magic Numbers

  Use `display.width` and `display.height` rather than literal numbers.

- The X Axis

  Draw a horizontal line of red pixels in the center of the screen.

- The Y Axis

  Draw a vertical line of red pixels in the center of the screen.

Tools Found: Bitmap, Variables, Ranges

Solution:

```
from codex import *

# Use half display width and half height to find the center of the screen
x_center = display.width / 2
y_center = display.height / 2

```

```
# Draw a horizontal line of pixels in the center of the screen
for x in range(display.width):
    display.set_pixel(x, y_center, RED)

# Draw a vertical line of pixels in the center of the screen
for y in range(display.height):
    display.set_pixel(x_center, y, RED)

```

Objective 4 - Bug Fix

Bug Fix

If you dig deep into the bitmap documentation for `set_pixel()` you will find that the `x` and `y` arguments you give it must be ints.

- Quite often when coding you learn things like this as much by *trying it* as you do by studying the documentation.
- *Be fearless - try stuff!*

Check the 'Trek!

The CodeTrek will guide you in fixing this bug,

CodeTrek:

```
from codex import *

# Use half display width and half height to find the center of the screen
x_center = int(display.width / 2)
y_center = int(display.height / 2)
```

Type Conversion to int

Easy-Peasy fix!

- The division resulted in the answer `120.0`
- That's correct, but the decimal point means it's a float, not an int.

Since you now know `display.set_pixel()` wants ints for `x` and `y` coordinates, you can use the `int()` bulit-in to convert it!

```

# Draw a horizontal line of pixels in the center of the screen
for x in range(display.width):
    display.set_pixel(x, y_center, RED)

# Draw a vertical line of pixels in the center of the screen
for y in range(display.height):
    display.set_pixel(x_center, y, RED)

```

Goal:

- Eliminate the error and show me some *horizontal* and *vertical* axes!

Tools Found: Bitmap, Keyword and Positional Arguments, int, float

Solution:

```
from codex import *

# Use half display width and half height to find the center of the screen
x_center = int(display.width / 2)
y_center = int(display.height / 2)

# Draw a horizontal line of pixels in the center of the screen
```

```
for x in range(display.width):
    display.set_pixel(x, y_center, RED)

# Draw a vertical line of pixels in the center of the screen
for y in range(display.height):
    display.set_pixel(x_center, y, RED)

```

***Objective 5* - Graphical Grid**

Graphical Grid

Your X and Y axes will help with symmetry and balance as you create artistic designs.

- But it is still difficult to judge *scale* at a glance.
- You need to create a **grid of *dots*** to clearly show pixel spacing over the whole screen!

Dot dot dot...

Each "dot" is just a single white pixel.

You could draw a *single line* of white dots like this:

```
for x in range(0, display.width, 10):
    display.set_pixel(x, y, WHITE)
```

Notice how this uses the `step` parameter of the range`(start, stop, step)` function to advance `x` by 10 every loop

Enter the Matrix

You'll need more than a single line of dots to complete this Objective.

- A grid that covers the whole screen is what you're after!
- But isn't that just a bunch of single dotted lines, drawn top to bottom?

Check the 'Trek!

The CodeTrek will show you how to *loop the loop* and make a merry matrix!

CodeTrek:

```
from codex import *

# Grid spacing (pixels)
GRID = 10
```

Define a grid spacing constant.

- Excellent self-documenting code!

```


# Use half display width and half height to find the center of the screen
x_center = int(display.width / 2)
y_center = int(display.height / 2)

# Draw a horizontal line of pixels in the center of the screen
for x in range(display.width):
    display.set_pixel(x, y_center, RED)

# Draw a vertical line of pixels in the center of the screen
for y in range(display.height):
    display.set_pixel(x_center, y, RED)

# Draw a grid of white pixels to cover the entire screen
for y in range(0, display.height, GRID):
    for x in range(0, display.width, GRID):
```

```
21          display.set_pixel(x, y, WHITE)
```

Draw the Grid

There are 2 loops here, an *inner* and an *outer* loop.

- The inner loop draws a horizontal dotted line.
- The outer loop steps to the next `y` and does it again!

```
22
```

Goal:

- The Pixel Grid

  Draw a grid of white pixels to cover the entire screen with a 10 pixel space between each white pixel.

Tools Found: Ranges, Constants, Loops

Solution:

```
from codex import *

# Grid spacing (pixels)
GRID = 10   #@1

# Use half display width and half height to find the center of the screen
x_center = int(display.width / 2)
y_center = int(display.height / 2)

# Draw a horizontal line of pixels in the center of the screen
for x in range(display.width):
    display.set_pixel(x, y_center, RED)

# Draw a vertical line of pixels in the center of the screen
for y in range(display.height):
    display.set_pixel(x_center, y, RED)

# Draw a grid of white pixels to cover the entire screen
for y in range(0, display.height, GRID):
    for x in range(0, display.width, GRID):
        display.set_pixel(x, y, WHITE)   #@2

```

Quiz 2 - Graphics Ranger

The following code draws a dashed line across the screen.

```
for d in range(0, 240, 20):
    for x in range(d, d + 10):
        display.set_pixel(x, 120, WHITE)
```

Question 1: What is the orientation of the *dashed line?*

✓ Horizontal

✗ Vertical

✗ Diagonal

Question 2: How many pixels *long* is each dash?

✓ 10

❌ 20

❌ 40

❌ 1

***Objective 6* - Keep It Simple**

Simplify and Optimize

You've transformed your screen into a *fantastic canvas* for graphical artistry!

- But before you move on you should neaten it up a bit.

Line Drawing Function

So far you've drawn horizontal and vertical lines using `for` 🔧loops, which is awesome.

- But the CodeX 🔧bitmap module has built-in line drawing functions which are *faster*, *simpler*, and support drawing *diagonal* lines too!

So, it's time to simplify and optimize your code, by replacing your line-drawing loops with the 🔧bitmap function, which is defined as follows:

```
# Draw a line from point (x1,y1) to
# point (x2,y2)
display.draw_line(x1, y1, x2, y2, color)
```

Frame it Up!

While you're at it, use the 🔧bitmap outlined rectangle function to create a BLUE border around the screen.

Here's how that function is defined:

```
# Draw a rectangle outline with upper left
# corner (x1,y1) and given width,height.
draw_rect(x1, y1, width, height, color)
```

Save to a New File!

Use the **File** → **Save As** menu to create a new file called ***LineArt***.

CodeTrek:

```
1  from codex import *
2
3  # Grid spacing (pixels)
4  GRID = 10
5
6  # Use half display width and half height to find the center of the screen
7  x_center = int(display.width / 2)
8  y_center = int(display.height / 2)
9
10 # Draw a grid of white pixels to cover the entire screen
11 for y in range(0, display.height, GRID):
12     for x in range(0, display.width, GRID):
13         display.set_pixel(x, y, WHITE)
14
15 # Draw a horizontal line in the center of the screen
16 display.draw_line(0, y_center, display.width - 1, y_center, RED)
17
18 # Draw a vertical line in the center of the screen
19 display.draw_line(x_center, 0, x_center, display.height - 1, RED)
```

Replace your *line loops*

```
    You might want to move them to AFTER you draw the grid, so the RED lines are on TOP.

20
21  # Draw a blue border
22  display.draw_rect(0, 0, display.width, display.height, BLUE)

    Blue bounding box border, baby.

        Booyah!

23
```

Goals:

- The X Axis Line

  Draw a horizontal red line in the center of the screen using the `display.draw_line()` function.

- The Y Axis Line

  Draw a vertical red line in the center of the screen using the `display.draw_line()` function.

- The Border Rectangle

  Draw a blue border rectangle using the `display.draw_rect()` function.

Tools Found: Loops, Bitmap

Solution:

```
1  from codex import *
2
3  # Grid spacing (pixels)
4  GRID = 10
5
6  # Use half display width and half height to find the center of the screen
7  x_center = int(display.width / 2)
8  y_center = int(display.height / 2)
9
10 # Draw a grid of white pixels to cover the entire screen
11 for y in range(0, display.height, GRID):
12     for x in range(0, display.width, GRID):
13         display.set_pixel(x, y, WHITE)
14
15 # Draw a horizontal line in the center of the screen
16 display.draw_line(0, y_center, display.width - 1, y_center, RED)
17
18 # Draw a vertical line in the center of the screen
19 display.draw_line(x_center, 0, x_center, display.height - 1, RED)  #@1
20
21 # Draw a blue border
22 display.draw_rect(0, 0, display.width, display.height, BLUE)  #@2
23
```

***Objective 7* - Get Artistic**

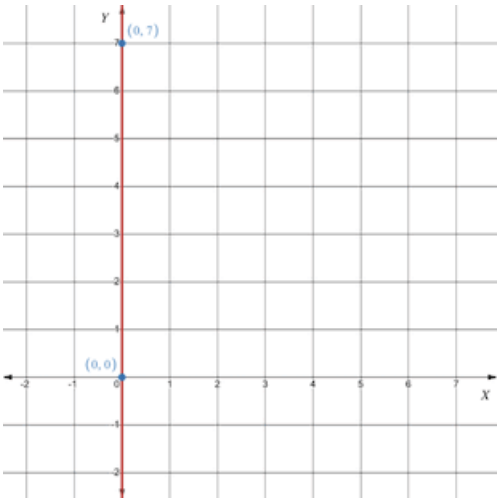
Time to Get Artistic!

You're working with straight lines, how artistic can you get?

- Well, you might be surprised!
- Straight lines can get downright *curvy!*

Draw some lines

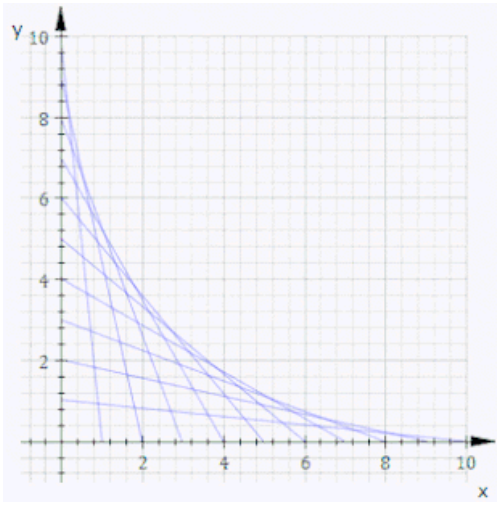
Notice as starting Y moves *down,* ending X moves to the *right*.



Note: Unlike the graph above, CodeX Y-axis values *increase* from the top down.

Whoa! String Art :-)

Watch the animated curve below! That's an envelope friends...



By Sam Derbyshire - English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=5232197

Check the 'Trek!

Alright, no more *stringing you along,* it's time to DO THIS. The CodeTrek will guide you artfully.

CodeTrek:

```
from codex import *

# Grid spacing (pixels)
GRID = 10

# Use half display width and half height to find the center of the screen
x_center = int(display.width / 2)
y_center = int(display.height / 2)

# Draw a grid of white pixels to cover the entire screen
for y in range(0, display.height, GRID):
    for x in range(0, display.width, GRID):
        display.set_pixel(x, y, WHITE)

# Draw a horizontal line in the center of the screen
```

```
display.draw_line(0, y_center, display.width - 1, y_center, RED)

# Draw a vertical line in the center of the screen
display.draw_line(x_center, 0, x_center, display.height - 1, RED)

# Draw a blue border
display.draw_rect(0, 0, display.width, display.height, BLUE)

# Draw a white spider web (aka. envelope) in the lower left corner
# with 7 lines spaced every 40 pixels
#              start(x,y)  end(x,y)
display.draw_line(0, 0,   0,   239, WHITE)
display.draw_line(0, 40,  40,  239, WHITE)
display.draw_line(0, 80,  80,  239, WHITE)
display.draw_line(0, 120, 120, 239, WHITE)
display.draw_line(0, 160, 160, 239, WHITE)
display.draw_line(0, 200, 200, 239, WHITE)
display.draw_line(0, 239, 239, 239, WHITE)
```

NOTICE !

Start X and End Y are always the same: `x=0` and `y=239`

- Increasing *start Y* and *end X* is how you *slide* the line down the screen...

```

```

Hint:

- ***Tedious* String Art?**

  Okay, this code is pretty repetitive. Don't worry, you'll be upgrading this to use a 🔧loop in the next Objective.

  - But for now you must *suffer!*

Goal:

- White Webbing

  Draw one white spider web (aka. envelope) in the lower left hand corner of your CodeX 🔧display using 7 lines spaced every 40 pixels.

Tools Found: Display

Solution:

```
from codex import *

# Grid spacing (pixels)
GRID = 10

# Use half display width and half height to find the center of the screen
x_center = int(display.width / 2)
y_center = int(display.height / 2)

# Draw a grid of white pixels to cover the entire screen
for y in range(0, display.height, GRID):
    for x in range(0, display.width, GRID):
        display.set_pixel(x, y, WHITE)

# Draw a horizontal line in the center of the screen
display.draw_line(0, y_center, display.width - 1, y_center, RED)

# Draw a vertical line in the center of the screen
display.draw_line(x_center, 0, x_center, display.height - 1, RED)

# Draw a blue border
display.draw_rect(0, 0, display.width, display.height, BLUE)

```

```
# Draw a white spider web (aka. envelope) in the lower left corner
# with 7 lines spaced every 40 pixels
display.draw_line(0, 0,   0,   239, WHITE)
display.draw_line(0, 40,  40,  239, WHITE)
display.draw_line(0, 80,  80,  239, WHITE)
display.draw_line(0, 120, 120, 239, WHITE)
display.draw_line(0, 160, 160, 239, WHITE)
display.draw_line(0, 200, 200, 239, WHITE)
display.draw_line(0, 239, 239, 239, WHITE)

```

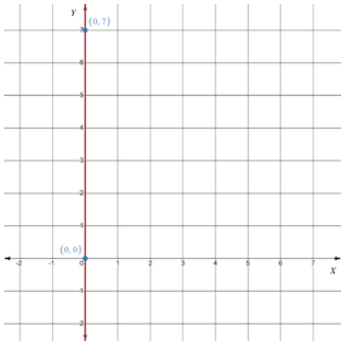
***Objective 8* - Loop Art**

Automate Your Art

Hey, that's a pretty cool display!

- But if you're gonna make more "webs" it would be nice to *automate* some of those magic numbers, and reduce the lines of code.

Your web spinning followed this plan:



1. *End1* of the string starts at Upper Left (UL) of screen `(0,0)`
2. *End2* of string starts at Lower Left (LL) of screen `(0,239)`
3. Move *End1* down 40 pixels (+Y)
4. Move *End2* right 40 pixels (+X)
5. Repeat moves

Track the line endpoints with your hands:

- Left index finger → ***End1***
- Right index finger → ***End2***
- Notice where you start.
- Left moves down, Right moves to the right...

Now that it's firmly in your mind...

Try it with a 🔧loop !

```
for i in range(0, 240, 40):
    display.draw_line(0, i,    # End1: from UL go down
                      i, 239,  # End2: from LL go right
                      WHITE)
```

I've expanded `display.draw_line(x1, y1, x2, y2, color)` across 3 lines to put comments on the *End1* and *End2* parts:

Check the 'Trek!

Your Turn!

Replace your **white web** code with the loop version above!

CodeTrek:

```
from codex import *

# Grid spacing (pixels)
GRID = 10

# Use half display width and half height to find the center of the screen
x_center = int(display.width / 2)
y_center = int(display.height / 2)

# Draw a grid of white pixels to cover the entire screen
for y in range(0, display.height, GRID):
    for x in range(0, display.width, GRID):
        display.set_pixel(x, y, WHITE)

```

```
# Draw a horizontal line in the center of the screen
display.draw_line(0, y_center, display.width - 1, y_center, RED)

# Draw a vertical line in the center of the screen
display.draw_line(x_center, 0, x_center, display.height - 1, RED)

# Draw a blue border
display.draw_rect(0, 0, display.width, display.height, BLUE)

# Draw a white spider web (aka. envelope) in the lower left corner
WEB_SPACING = ??
for i in range(0, 240, WEB_SPACING):
    display.draw_line(0, i,    # from UL go down
                      i, 239,  # from LL go right
                      WHITE)
```

Replace the web-drawing code with a loop.

- I've added a constant for `WEB_SPACING`.
- What should the value of that be?

Goals:

- Use a loop to draw a web in the lower-left corner.
- Create a constant `WEB_SPACING` and try setting it to a value less than 4 0.
  - *Experiment with a few values!*

Tools Found: Loops, Constants

Solution:

```
from codex import *

# Grid spacing (pixels)
GRID = 10

# Use half display width and half height to find the center of the screen
x_center = int(display.width / 2)
y_center = int(display.height / 2)

# Draw a grid of white pixels to cover the entire screen
for y in range(0, display.height, GRID):
    for x in range(0, display.width, GRID):
        display.set_pixel(x, y, WHITE)

# Draw a horizontal line in the center of the screen
display.draw_line(0, y_center, display.width - 1, y_center, RED)

# Draw a vertical line in the center of the screen
display.draw_line(x_center, 0, x_center, display.height - 1, RED)

# Draw a blue border
display.draw_rect(0, 0, display.width, display.height, BLUE)

# Draw a white spider web (aka. envelope) in the lower left corner
WEB_SPACING = 20
for i in range(0, 240, WEB_SPACING):
    display.draw_line(0, i,    # from UL go down
                      i, 239,  # from LL go right
                      WHITE)
```

***Objective 9* - Get Colorful**

A Splash of Color!

Take some time to experiment with the code you have now.

- *Art is all about experimentation!*

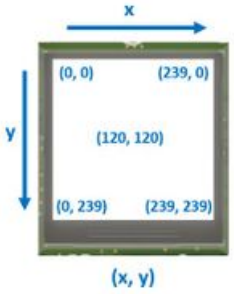
Run It!

Try some different colors.

- **CHALLENGE:** Can you get webs in *all four corners?*

```
# Example to get you started!
for i in range(0, 240, WEB_SPACING):
    display.draw_line(0, i,    # from UL go down
                      i, 239,  # from LL go right
                      MAGENTA)

    display.draw_line(i, 239,  # from LL go right
                      239, 239 - i,  # from LR go up
                      GREEN)
```

More Flexible Webbing



Check the 'Trek!

CodeTrek:

```
from codex import *

# Grid spacing (pixels)
GRID = 10

# Use half display width and half height to find the center of the screen
x_center = int(display.width / 2)
y_center = int(display.height / 2)

# Draw a grid of white pixels to cover the entire screen
for y in range(0, display.height, GRID):
    for x in range(0, display.width, GRID):
        display.set_pixel(x, y, WHITE)

# Draw a horizontal line in the center of the screen
display.draw_line(0, y_center, display.width - 1, y_center, RED)

# Draw a vertical line in the center of the screen
display.draw_line(x_center, 0, x_center, display.height - 1, RED)

# Draw a blue border
display.draw_rect(0, 0, display.width, display.height, BLUE)

def draw_web(x1, y1, x2, y2, count, color):
    """Draw web, rotating line counterclockwise,
       P1 chasing P2."""
```

Define a new function.

- Looks a lot like `display.draw_line()` but with *one more* parameter, `count`.
- It draws `count` lines while moving P1 `x1,y1` toward P2 `x2,y2`.

Notice the docstring comment? Triple-quotes surround this multi-line style comment which is used in Python to describe functions and modules.

(You don't HAVE to copy this comment as-is... Make it your own!)

```
    # Calculate step size "deltas" for x and y
    dx1 = int((x2 - x1) / count)
    dy1 = int((y2 - y1) / count)
```

```
30      dx2 = dy1
31      dy2 = -dx1
```

Calculate the *deltas*:

- A "delta" is a small change to a value.
- For example if you're changing `x` you might call a small change `dx`, for "delta x".

Here you are moving P1 toward P2, in `count` steps.

- Use the `int()` type conversion 🔧built-in since `set_pixel()` requires 🔧integers.
- P2 moves perpendicular to P1, in a counterclockwise direction.

```
32
33      # Draw line and move endpoints by dx,dy each loop
34      for i in range(count):
35          display.draw_line(x1, y1, x2, y2, color)
36          x1 = x1 + dx1
37          y1 = y1 + dy1
38          x2 = x2 + dx2
39          y2 = y2 + dy2
```

Step and Repeat!

- Draw a line
- Adjust endpoints by "deltas"
- Do it again!

```
40
41  WEB_SPACING = 20
42
43  # String some art!
44  draw_web(0,0,     0,239,   WEB_SPACING, GREEN)
45  draw_web(0,239,   239,239, WEB_SPACING, YELLOW)
46  draw_web(??,??,   239,0,   WEB_SPACING, CYAN)
47  draw_web(239,0,   ??,??,   WEB_SPACING, MAGENTA)
48  draw_web(120,0,   120,120, WEB_SPACING, ORANGE)
49  draw_web(??,??,   120,120, WEB_SPACING, RED)
50  draw_web(120,239, ??,??,   WEB_SPACING, WHITE)
51  draw_web(239,120, 120,120, WEB_SPACING, PINK)
```

Make it Beautiful!

- I've given you a start, but you'll need to fill in some points!
- You can use ANY two points for `x1,y1` and `x2,y2`.
- But remember the "toothpick" only rotates *counterclockwise* as P1 slides in to P2's spot.

```
52
```

Goals:

- Define a function `def draw_web(x1, y1, x2, y2, count, color):` that starts with a line between P1--P2 and spins it counterclockwise to make a web!
- Draw at least 6 different colorful webs using your new 🔧function.
    - *Unleash your inner **artiste***

Tools Found: Functions, Parameters, Arguments, and Returns, Comments, import, Built-In Functions, int

Solution:

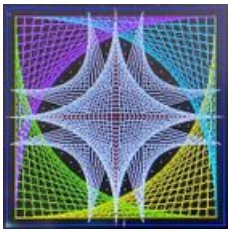
```
1  from codex import *
2
3  # Grid spacing (pixels)
4  GRID = 10
5
6  # Use half display width and half height to find the center of the screen
```

```
x_center = int(display.width / 2)
y_center = int(display.height / 2)

# Draw a grid of white pixels to cover the entire screen
for y in range(0, display.height, GRID):
    for x in range(0, display.width, GRID):
        display.set_pixel(x, y, WHITE)

# Draw a horizontal line in the center of the screen
display.draw_line(0, y_center, display.width - 1, y_center, RED)

# Draw a vertical line in the center of the screen
display.draw_line(x_center, 0, x_center, display.height - 1, RED)

# Draw a blue border
display.draw_rect(0, 0, display.width, display.height, BLUE)

def draw_web(x1, y1, x2, y2, count, color):
    """Draw web, rotating line counterclockwise,
       end1 chasing end2."""
    # Calculate step size "deltas" for x and y
    dx1 = int((x2 - x1) / count)
    dy1 = int((y2 - y1) / count)
    dx2 = dy1
    dy2 = -dx1

    # Draw line and move endpoints by dx,dy each loop
    for i in range(count):
        display.draw_line(x1, y1, x2, y2, color)
        x1 = x1 + dx1
        y1 = y1 + dy1
        x2 = x2 + dx2
        y2 = y2 + dy2

WEB_SPACING = 20

# String some art!
draw_web(0,0,     0,239,   WEB_SPACING, GREEN)
draw_web(0,239,   239,239, WEB_SPACING, YELLOW)
draw_web(239,239, 239,0,   WEB_SPACING, CYAN)
draw_web(239,0,   0,0,     WEB_SPACING, MAGENTA)
draw_web(120,0,   120,120, WEB_SPACING, ORANGE)
draw_web(0,120,   120,120, WEB_SPACING, RED)
draw_web(120,239, 120,120, WEB_SPACING, WHITE)
draw_web(239,120, 120,120, WEB_SPACING, PINK)


```

Mission 14 Complete

Sweet Drawings!

Of course, "string art" is only a small taste of how you can *get creative* with computer graphics.



- An excellent "creativity hack" is to experiment with *constraints* - like only drawing with straight lines.
- You can of course do ANYTHING with individual pixels, but limiting yourself to *lines only* opened up a surprising amount of *artistic discovery!*

Imagine...

Smallness is one of the cool things about the CodeX.

- There is only so much space in which to play.
- So with a little bit of effort, you can make a big impact!
- And there are an *infinite* number of ways you'll discover to let your creativity shine!

Mission 15 **- Handball**

Handball!

Ready to develop a truly iconic video game?

This is the first of a 2-part Mission sequence

1. Handball

Build a handheld gaming framework, culminating in a fun, playable game.

2. Breakout

Enhance your game, reproducing an all-time arcade classic!

The Game Plan

This mission will lead you on a step-by-step journey to develop a retro video game of *American Handball.*

- The game is like a 1-player version of the classic "Pong".
- Buttons move a paddle side-to-side across the bottom of the screen.
- A ball bounces off the sides and top of the screen.
- Score points by hitting the ball with your paddle.
- You get 3 "lives" - lose those balls and it's GAME OVER!

Buckle up! You're following in the footsteps of game development LEGENDS.

Objective 1 **- BallX**

Settle Into the Effort...

Relax, this Mission will *not* throw a bunch of new Python knowledge at you!

- You will mostly be using *what you have already learned.*
- Not too difficult, right?

WRONG!

You'll recognize the Python concepts, but it will still be a challenge to piece together the logic and flow of the game.

- But you CAN do it!
- *Slow down* and be sure you understand each **Objective** before moving on!
- *DO NOT TYPE CODE YOU DON'T UNDERSTAND!*

Create a New File!

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

Begin with a Ball

For starters, draw a ball and move it horizontally across the screen.

- The ball in *retro-handball* is a square :-)
- Just like before when you did *animation*, remember to erase the old position.

Check the 'Trek!

Basically, you need a loop where you:

- Draw the ball
- Update the position (just increase X for now)
- Erase the ball

- Repeat!

Physics Much?

You're creating a very basic *physics engine* here! This Python code models the mechanics of velocity, distance, and time. Check the Hints for more details.

CodeTrek:

```
1  from codex import *
2  import time
```

What, no `sleep` ?

- This time you're doing 🔧import like the pros.
- Relax, if you need `sleep()` you can say `time.sleep()`

```
3
4  BALL_SZ = 4
5
6  def draw_ball():
7      global ball_pix
8      pix = round(ball_pos)  # Round to nearest pixel
9      if pix != ball_pix:
10         display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, BLACK)
11         ball_pix = pix
12         display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, WHITE)
```

Draw the Ball

It's just a rectangle. Erase the old location with a BLACK rectangle, then Draw the new WHITE one.

- Notice the `round()` 🔧built-in function?
- The main 🔧loop uses 🔧floats for accuracy, but round to an 🔧int for drawing on the screen.
- Save the `ball_pix` global variable so you know where to ***erase*** next time!

```
13
14 def serve_ball():
15     global ball_pos, ball_v, ball_pix
16     ball_v = 0.200  # 200 pixels per second
17     ball_pos = 0.0
18     ball_pix = round(ball_pos)
```

Serve the Ball

The ball is defined by some 🔧global variables:

- `ball_pos` = The 🔧float screen x-coordinate of the ball's *position*.
- `ball_v` = The ball's *velocity* (speed) in the X direction. The velocity is in "pixels per millisecond". So x1000 gives you "pixels per second".
- `ball_pix` = The ball's 🔧int screen x-pixel location.

```
19
20 serve_ball()
21
22 while True:
23     dt = 10  # ms
24
25     # Update ball position
26     x = ball_pos
27     x = x + ball_v * dt
28
29     ball_pos = x
30     draw_ball()
31
32     # Pace the animation
```

```
33      time.sleep_ms(dt)
```

Main Game Loop

This is set up for you to expand on later.

- Each loop takes "delta time" of 10ms (remember, delta means *change*)
- Update the position: $distance = speed * time$
- Draw the ball (erase old position first)
- Sleep to pace the movement

```
34
35
36
37
38
```

Hint:

- **Understanding the Timing**

  This code uses `sleep_ms()` which is like `sleep()` but delays for the specified time in *milliseconds* rather than seconds.

  - There are 1000 *milliseconds* in 1 second.

  To update the ball's x-coordinate, you need to know two things:

  1. How *fast* is the ball moving? *(the ball **velocity**)*
  2. How much *time* has passed since you last moved the ball? *(dt = delta time)*

  Example:

  If the ball moves 200 pixels every 1000 milliseconds, how far does it travel in `dt = 10` ms?

$$velocity = \frac{distance}{time} = \frac{200pix}{1000ms} = 0.200pix/ms$$

$$distance = velocity \cdot time = 0.200\frac{pix}{\cancel{ms}} \cdot 10\cancel{ms} = 2pixels$$

  So at that speed (0.200 pixels / ms) you'd need to move the ball by 2 pixels each time, at 10ms per loop.

```
# Add distance to X, each time through loop.
x = x + ball_v * dt
```

Goals:

- Define a `draw_ball()` 🔧function.
- Define a `serve_ball()` 🔧function.
- Call `serve_ball()` before your main 🔧loop.
- Call `draw_ball()` inside the 🔧loop.

Tools Found: Loops, Functions, import, Locals and Globals, float, int, Built-In Functions

Solution:

```
1  from codex import *
2  import time
3
4  BALL_SZ = 4
5
6  def draw_ball():
7      global ball_pix
8      pix = round(ball_pos)  # Round to nearest pixel
9      if pix != ball_pix:
```

```
        display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = 0.2
    ball_pos = 0.0
    ball_pix = round(ball_pos)

serve_ball()

while True:
    dt = 10  # ms

    # Update ball
    x = ball_pos
    x = x + ball_v * dt

    ball_pos = x
    draw_ball()

    # Slow
    time.sleep_ms(dt)
```

Quiz 1 **- Start Up!**

Question 1: How many *milliseconds* are in 1 second?

✓ 1000

✗ 0.001

✗ 1 million

✗ 100

Question 2: Say the ball moves at a velocity of $\frac{1}{2}$ pixel per millisecond.

- How far would it move in 10 milliseconds?

✓ 5 pixels

✗ 10 pixels

✗ 50 pixels

✗ 20 pixels

Question 3: Your *game loop* uses a 🔧global variable `ball_v` for the ball's velocity. Where is this 🔧variable initialized? *(first assigned to)*

✓ Inside the `serve_ball()` function.

✗ At the beginning of the game loop.

✗ Inside the `draw_ball()` function.

Objective 2 **- Bounce X**

Tracking the Ball?

Your ball should fly across the CodeX screen from *left to right* each time you run your program.

- *If that's not happening, stop and debug your code!*
- After it flies off the right side it disappears forever.

Next Step - *Bounce!*

In **Handball** the ball bounces off walls.

- Keep your ball on the screen by making it *bounce* when it reaches either edge of the screen.

Run It!

Test Your Code

Is your ball bouncing back and forth?

CodeTrek:

```
1  from codex import *
2  import time
3
4  BALL_SZ = 4
5
6  def draw_ball():
7      global ball_pix
8      pix = round(ball_pos)  # Round to nearest pixel
9      if pix != ball_pix:
10         display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, BLACK)
11         ball_pix = pix
12         display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, WHITE)
13
14 def serve_ball():
15     global ball_pos, ball_v, ball_pix
16     ball_v = 0.2
17     ball_pos = 0.0
18     ball_pix = round(ball_pos)
19
20 serve_ball()
21
22 while True:
23     dt = 10  # ms
24
25     # Update ball
26     x = ball_pos
27     x = x + ball_v * dt
28
29     # Check for collision with walls
30     if x <= 0 or x >= 240 - BALL_SZ:
```

Stay in Bounds

The 🔧display is 240 pixels wide, so the edges are `0` and `239`.

- Notice on the right side you have to account for `BALL_SZ` since you want to bounce as soon as the right edge of the ball hits the wall!

```
31         ball_v = ?? # TODO: reverse direction
```

Bounce by *reversing the direction* along the X-axis.

- Multiplying by `-1` will change the sign from *positive* X direction to *negative* and vice-versa.

```
32
33     ball_pos = x
34     draw_ball()
35
36     # Slow
37     time.sleep_ms(dt)
```

Hint:

- ## Rebound Math

  When your ball hits a wall, how should its *velocity* change?

  The Handball game keeps things simple: the walls *perfectly reverse* the direction of the ball.

  - The left and right walls reverse the X component of velocity.
  - Later when you add top and bottom walls, they will reverse the Y component.

    ```
    # Multiply by (-1) to reverse the ball's X direction.
    ball_v = ball_v * -1
    ```

Goal:

- Reverse the direction of the ball when it hits the left or right edge of the screen.

Tools Found: Display

Solution:

```
from codex import *
import time

BALL_SZ = 4

def draw_ball():
    global ball_pix
    pix = round(ball_pos)  # Round to nearest pixel
    if pix != ball_pix:
        display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = 0.2
    ball_pos = 0.0
    ball_pix = round(ball_pos)

serve_ball()

while True:
    dt = 10  # ms

    # Update ball
    x = ball_pos
    x = x + ball_v * dt

    # Check for collision with walls
    if x <= 0 or x >= 240 - BALL_SZ:
        ball_v = ball_v * -1

    ball_pos = x
    draw_ball()

    # Slow
    time.sleep_ms(dt)
```

***Objective 3* - No Sleep**

No Sleeping!

This is to be a *fast paced* game, right?

- You'll want to check button inputs quickly, and you can't do that when CodeX is sleeping.

I know, your 10ms 🔧loop is working just fine right now. But as you add more features to the game, the code inside the loop will take time to run too.

- *Think about it.* That `sleep_ms()` can only make your game *slower*, right?
- You'll want to avoid any such ***blocking*** functions in your game loop!

Check the 'Trek!

Replace `sleep_ms()` with *Calculated* `dt`

You don't have to sleep, but you DO need to know what `dt` is, since you're using it to move the ball!

CodeTrek:

```
1  from codex import *
2  import time
3
4  BALL_SZ = 4
5
6  def draw_ball():
7      global ball_pix
8      pix = round(ball_pos)  # Round to nearest pixel
9      if pix != ball_pix:
10         display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, BLACK)
11         ball_pix = pix
12         display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, WHITE)
13
14 def serve_ball():
15     global ball_pos, ball_v, ball_pix
16     ball_v = 0.2
17     ball_pos = 0.0
18     ball_pix = round(ball_pos)
19
20 def elapsed_ms():
21     """Returns milliseconds elapsed since last called"""
22     global ms
23     now = time.ticks_ms()
24     diff = time.ticks_diff(now, ms)
25     ms = now
26     return diff
```

A 🔧function to Remember

This function *remembers* the millisecond count `ms` the last time you called it, and returns the difference in milliseconds between then and now.

- That *elapsed time* in milliseconds will be your *delta time* `dt`.
- But wait, what about the first time you call `elapsed_ms()`. How does `ms` get initialized?

```
27
28 serve_ball()
29
30 ms = ??  # initialize ms global to the current ticks_ms()
```

Initialize!

When your program first starts, set the value of the 🔧global `ms`.

- After this, `elapsed_ms()` will work perfectly.

```
31
32 while True:
33     dt = elapsed_ms()
```

***Calculate* the `dt` delta time.**

- Instead of sleeping, you're *measuring* the time it takes.

- No more constant 10ms loops.
- Depending on how much is happening inside your loop, `dt` will increase or decrease.

```
34
35      # Update ball
36      x = ball_pos
37      x = x + ball_v * dt
38
39      # Check for collision with walls
40      if x <= 0 or x >= 240 - BALL_SZ:
41          ball_v = ball_v * -1
42
43      ball_pos = x
44      draw_ball()
45
46      # Remove sleep!
```

No more sleeping!

Remove your sleep delay.

- From now on your loop will run as fast as possible.
- Hmmm... you're still using `dt` to move the ball above. So if you're not controlling the speed with `sleep_ms()` what does `dt` even mean??

```
47
48
```

Hint:

- ## Initialization

  Programs often need to set things up the initial (first) time. This is called "initialization".

  "At the dawn of time... when the program first runs... do this."

```
# Example:
ms = time.ticks_ms()  # initialize ms global to the current millisecond count
```

Goals:

- Remove the `time.sleep_ms()` call from your 🔧loop.
- Define a new function `elapsed_ms()` that 🔧returns the number of milliseconds elapsed since the last time it was called.
- Initialize `ms` to the current `ticks_ms()` before your loop begins.
- Use `elapsed_ms()` to set your `dt` inside the loop.

Tools Found: Loops, Parameters, Arguments, and Returns, Functions, Locals and Globals

Solution:

```
1  from codex import *
2  import time
3
4  BALL_SZ = 4
5
6  def draw_ball():
7      global ball_pix
8      pix = round(ball_pos)  # Round to nearest pixel
9      if pix != ball_pix:
10         display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, BLACK)
11         ball_pix = pix
12         display.fill_rect(ball_pix, 120, BALL_SZ, BALL_SZ, WHITE)
13
14 def serve_ball():
15     global ball_pos, ball_v, ball_pix
16     ball_v = 0.2
```

```
    ball_pos = 0.0
    ball_pix = round(ball_pos)

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

serve_ball()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()

    # Update ball
    x = ball_pos
    x = x + ball_v * dt

    # Check for collision with walls
    if x <= 0 or x >= 240 - BALL_SZ:
        ball_v *= -1

    ball_pos = x
    draw_ball()
```

Quiz 2 - Delta Force

Question 1: What do the letters D T stand for in the 🔧variable `dt` ?

- ✓ "delta time"
- ✗ "dog tired"
- ✗ "difference time"
- ✗ "delta tricep"
- ✗ "data test"

Question 2: What would you expect the value of `dt` to be after the following code runs?

```
elapsed_ms()
time.sleep_ms(42)
dt = elapsed_ms()
```

- ✓ 42
- ✗ Error: No target for assignment.
- ✗ 40
- ✗ 10

Question 3: How does the 🔧function `elapsed_ms()` "remember" the millisecond value the last time it was called?

- ✓ The 🔧global `ms` saves milliseconds between function calls.
- ✗ All 🔧variables inside a function are preserved across calls.
- ✗ Computers don't remember. Eschew anthropomorphism.

***Objective 4* - Bounce 2D**

Enter the 2nd Dimension

You're *having a ball* with the X-axis :-)

But your game needs movement in the Y-axis too!

- The player needs to be able to bounce the ball at any angle across the screen.

X and Y *...living in 2D*

Right now your code keeps only X values for *three* 🔧variables controlling the ball:

```
ball_pos  # The precise (float) position
ball_pix  # The pixel position
ball_v    # The velocity
```

You could add *three more* variables to hold the Y values... BUT maybe there's a better way! How about changing your variables to hold a 🔧list or 🔧tuple?

For example, the tuple `(x, y)` would be a nice way to position the ball:

```
# Set position as a tuple (x, y)
ball_pos = (120.0, 120.0)
```

- The X value is `ball_pos[0]`
- The Y value is `ball_pos[1]`

Then you could make a `ball_pix` 🔧tuple by rounding `ball_pos` to 🔧ints, like so:

```
ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
```

Check the 'Trek!

The CodeTrek will guide you to convert those X 🔧variables to 🔧tuples or 🔧lists so you can bounce in 2D!

Run It!

Your ball should be bouncing off ALL FOUR WALLS!

- If it's not, stop now and do some troubleshooting.

CodeTrek:

```
1  from codex import *
2  import time
3
4  BALL_SZ = 4
5
6  def draw_ball():
7      global ball_pix
8      pix = (round(ball_pos[0]), round(ball_pos[1]))
9      if pix != ball_pix:
10         display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
11         ball_pix = pix
12         display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
```

Make `pix` a 🔧tuple.

- The 🔧comparison operators can compare 🔧tuples!

Also, in `fill_rect()` use `ball_pix[0]` and `ball_pix[1]` for the X and Y position.

```
13
14 def serve_ball():
```

```
15      global ball_pos, ball_v, ball_pix
16      ball_v = [0.2, 0.35]
17      ball_pos = (120.0, 120.0)
18      ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
```

Start with the Serve!

This is where your *ball control* variables are initialized.

- Make the velocity `ball_v` a 🔧list so you can update it in-place.
- Make `ball_pos` and `ball_pix` 🔧tuples.

I've provided some nice initialization values: a reasonable speed, and center-of-screen position.

```
19
20  def elapsed_ms():
21      """Returns milliseconds elapsed since last called"""
22      global ms
23      now = time.ticks_ms()
24      diff = time.ticks_diff(now, ms)
25      ms = now
26      return diff
27
28  serve_ball()
29
30  ms = time.ticks_ms()
31
32  while True:
33      dt = elapsed_ms()
34
35      # Update ball
36      x, y = ball_pos
37      x = x + ball_v[0] * dt
38      y = y +  ???  * dt
```

In your loop, update both X and Y position.

- Based on the X and Y velocity, naturally!

```
39
40      # Check for collision with walls
41      collision = False
42      if x <= 0 or x >= 240 - BALL_SZ:
43          collision = True
44          ball_v[0] = ball_v[0] * -1
45      if y <= 0 or y >= 240 - BALL_SZ:
46          collision = True
47          ball_v[1] = ???  * -1
```

A few changes to wall collision detection:

- Set a 🔧bool `collision` if ball hit a wall.
- Modify X bounds-check to reverse `ball_v[0]`
- Add Y bounds-check, reversing `ball_v[1]`

```
48
49      if not collision:
50          ball_pos = (x, y)
51          draw_ball()
```

Finally, set a new `ball_pos` 🔧tuple and call `draw_ball()` *only* if there was no collision.

- Otherwise, let the `ball_v` change take effect first. You don't want to draw the ball if it's out of bounds.

```
52
```

Goals:

- Update `serve_ball()` to use a 🔧list for `ball_v` and 🔧tuples for `ball_pos` and `ball_pix`.
- Update `draw_ball()` to use 🔧tuples for `ball_pos` and `ball_pix`.
- Inside your loop, use `ball_v[0]` and `ball_v[1]` to update both X and Y components of `ball_pos`
- Change wall collision to check X and Y bounds.
    - Also set `collision` 🔧bool if a wall was hit.
- Set a new `ball_pos` 🔧tuple and draw the ball IF there wasn't a collision.

Tools Found: Variables, list, tuple, int, bool, Comparison Operators

Solution:

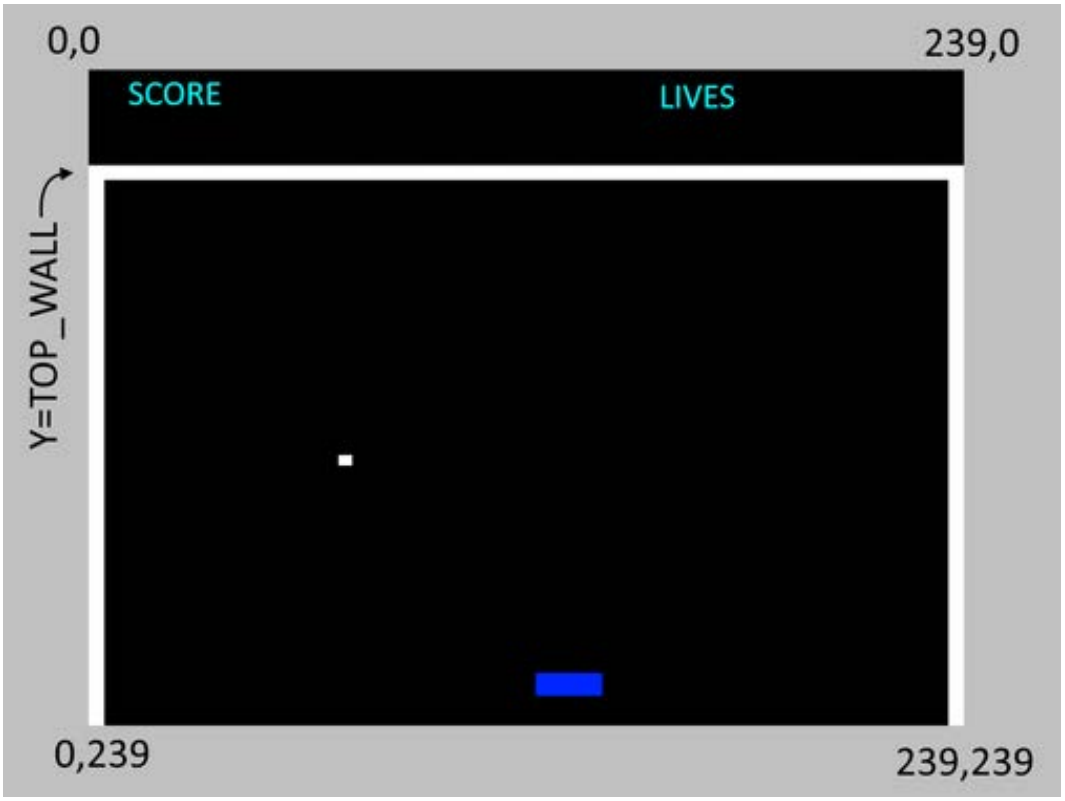
```
from codex import *
import time

BALL_SZ = 4

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = [0.1, 0.15]
    # ball_v = [0.2, 0.35]
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

serve_ball()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 0 or x >= 240 - BALL_SZ:
        collision = True
        ball_v[0] = ball_v[0] * -1
    if y <= 0 or y >= 240 - BALL_SZ:
        collision = True
        ball_v[1] = ball_v[1] * -1

    if not collision:
        ball_pos = (x, y)
        draw_ball()

```

***Objective 5* - Layout**

Screen Layout

Your ball physics are working great. But Handball is not played on an empty black screen!

- It's time to decorate the screen with walls.
- And you need a place to show the SCORE and LIVES remaining during the game.



The Plan

The picture above shows where you're headed with the screen layout.

- The SCORE and LIVES will be displayed at the top.
- Draw 1-pixel wide walls down both sides, and across the top at `y = TOP_WALL`.

Check the 'Trek!

The CodeTrek outlines a `draw_screen_layout()` function that matches the picture.

- *Try just adding that, and watch your wrecking ball bounce right through the layout drawing!*

Have Fun!

You may need to experiment a bit to figure out the proper bounds for *wall collisions*.

- Be sure your ball is bouncing around between the walls!

CodeTrek:

```
1  from codex import *
2  import time
3
4  # Screen layout
5  TOP_WALL = 20
6  BALL_SZ = 4
```

Add layout constants.

- This layout puts the score at the TOP.
- So you need to move the top *ball-boundary* down

A new constant `TOP_WALL` will do the trick.

```
7
8  def draw_ball():
9      global ball_pix
10     pix = (round(ball_pos[0]), round(ball_pos[1]))
11     if pix != ball_pix:
12         display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
13         ball_pix = pix
14         display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
15
16 def serve_ball():
17     global ball_pos, ball_v, ball_pix
18     ball_v = [0.2, 0.35]
19     ball_pos = (120.0, 120.0)
20     ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
21
22 def elapsed_ms():
23     """Returns milliseconds elapsed since last called"""
24     global ms
25     now = time.ticks_ms()
26     diff = time.ticks_diff(now, ms)
27     ms = now
28     return diff
29
30 def draw_screen_layout():
31     display.draw_line(0, TOP_WALL, 0, 239, WHITE)
32     display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
33     display.draw_line(239, TOP_WALL, 239, 239, WHITE)
34     display.draw_text("SCORE", 4, 0, BLUE, 1)
35     display.draw_text("LIVES", 150, 0, BLUE, 1)
```

Add a function to draw the screen layout.

- The left and right edges are at `x = 0` and `x = 239`
- The "SCORE" and "LIVES" labels are positioned in the TOP area.

*I know, there are some **magic numbers** in here.*
I'm compromising to keep the code brief, for your sake!

```
36
37 draw_screen_layout()
```

Don't forget to call the new `draw_screen_layout()` function once, at the beginning of time.

```
38 serve_ball()
39
40 ms = time.ticks_ms()
41
42 while True:
43     dt = elapsed_ms()
44
45     # Update ball
46     x, y = ball_pos
47     x = x + ball_v[0] * dt
48     y = y + ball_v[1] * dt
49
50     # Check for collision with walls
51     collision = False
52     if x <= ?? or x >= ?? - BALL_SZ:
53         collision = True
54         ball_v[0] = ball_v[0] * -1
55     if y <= ?? or y >= ?? - BALL_SZ:
56         collision = True
57         ball_v[1] = ball_v[1] * -1
```

Update Your Boundaries!

- Are `0` and `240` still your X limits? *Seems like you need to scoot them in 1 pixel.*
- Are `0` and `240` still your Y limits? *Hmmm... I heard `TOP_WALL + 1` is the new `0` in Y-town :-)*

```
58
59      if not collision:
60          ball_pos = (x, y)
61          draw_ball()
62
```

Goals:

- Add the `TOP_WALL` 🔧constant y-pixel location.
- Define a `def draw_screen_layout()` function that draws 3 walls and 2 text labels.
- Call your `draw_screen_layout()` function before your main 🔧loop starts.
- Update your wall-collision boundaries.
    - Don't let the ball wreck your walls!
    - Be sure to use `TOP_WALL` somewhere in your calculations.

Tools Found: Constants, Loops, Functions

Solution:

```
1  from codex import *
2  import time
3
4  # Screen layout
5  TOP_WALL = 20
6  BALL_SZ = 4
7
8  def draw_ball():
9      global ball_pix
10     pix = (round(ball_pos[0]), round(ball_pos[1]))
11     if pix != ball_pix:
12         display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
13         ball_pix = pix
14         display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
15
16 def serve_ball():
17     global ball_pos, ball_v, ball_pix
18     ball_v = [0.2, 0.35]
19     ball_pos = (120.0, 120.0)
20     ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
21
22 def elapsed_ms():
23     """Returns milliseconds elapsed since last called"""
24     global ms
25     now = time.ticks_ms()
26     diff = time.ticks_diff(now, ms)
27     ms = now
28     return diff
29
30 def draw_screen_layout():
31     display.draw_line(0, TOP_WALL, 0, 239, WHITE)
32     display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
33     display.draw_line(239, TOP_WALL, 239, 239, WHITE)
34     display.draw_text("SCORE", 4, 0, BLUE, 1)
35     display.draw_text("LIVES", 150, 0, BLUE, 1)
36
37 draw_screen_layout()
38 serve_ball()
39
40 ms = time.ticks_ms()
41
```

```
while True:
    dt = elapsed_ms()

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or x >= 239 - BALL_SZ:
        collision = True
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1 or y >= 239 - BALL_SZ:
        collision = True
        ball_v[1] = ball_v[1] * -1

    if not collision:
        ball_pos = (x, y)
        draw_ball()

```

***Objective 6* - Sound FX**

Sound Effects!

It's time to add retro arcade ***beeps*** to your bounces.

- You already know how to use 🔧soundlib to create *tones*.
- For Handball, just some short *beeps* is what you're after.

Non-Blocking Beeps!?

A cool feature of 🔧soundlib tones is they're *non-blocking*.

- You'll recall, that means your code can *start* a **tone** and then continue running code *while it plays!*
- That's awesome! After all, you still need to move the ball and check for player input while sound is playing.

But how do you play a short beep? How about:

```
# A short beep!
tone.play()
sleep_ms(50)  # Yikes! BLOCKING!
tone.stop()
```

No. *That's not the way!*

Yes, it's a short beep. BUT it totally stops your program for **50 whole milliseconds!**

Check the 'Trek!

The CodeTrek will show you a much better way. It still uses `tone.play()` and `tone.stop()`, but the *timing* of the beep is done using *milliseconds* `dt` in the game loop.

- *No time is wasted!*

CodeTrek:

```
from codex import *
import time
from soundlib import *

# Screen layout
TOP_WALL = 20
BALL_SZ = 4

# Sounds
tone = soundmaker.get_tone('trumpet')
```

```
11  SIDES_TONE = 392
12  TOP_TONE = 494
```

Initialize your sound tone.

- Keeping it simple with just a "trumpet tone" for the *retro beep* sound.
- Change the *pitch* of the beep for different collision types. Define 🔧constants for SIDE and TOP pitches (frequencies in Hertz).

```
13  sound_cut = 0 # ms until sound effect stops
```

A 🔧variable to *stop* the sound.

- You'll check this in the main 🔧loop, so you can stop the sound after a certain number of milliseconds.

```
14
15  def draw_ball():
16      global ball_pix
17      pix = (round(ball_pos[0]), round(ball_pos[1]))
18      if pix != ball_pix:
19          display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
20          ball_pix = pix
21          display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
22
23  def serve_ball():
24      global ball_pos, ball_v, ball_pix
25      ball_v = [0.2, 0.35]
26      ball_pos = (120.0, 120.0)
27      ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
28
29  def elapsed_ms():
30      """Returns milliseconds elapsed since last called"""
31      global ms
32      now = time.ticks_ms()
33      diff = time.ticks_diff(now, ms)
34      ms = now
35      return diff
36
37  def draw_screen_layout():
38      display.draw_line(0, TOP_WALL, 0, 239, WHITE)
39      display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
40      display.draw_line(239, TOP_WALL, 239, 239, WHITE)
41      display.draw_text("SCORE", 4, 0, BLUE, 1)
42      display.draw_text("LIVES", 150, 0, BLUE, 1)
43
44  def beep(freq):
45      global sound_cut
46      tone.set_pitch(freq)
47      tone.play()
48      sound_cut = 50  # ms countdown
```

Make a `beep()` 🔧function.

- Call this with different *frequency* values depending on what the ball hit.

NOTE: 🔧soundlib tones do NOT stop on their own!

- That's what `sound_cut` is for.
- Your main loop should *stop* the sound after `sound_cut` milliseconds.

```
49
50  draw_screen_layout()
51  serve_ball()
52
53  ms = time.ticks_ms()
54
55  while True:
56      dt = elapsed_ms()
57
58      # Check sound timer
```

```
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()
```

Check if a sound is playing: `sound_cut > 0`.

- If so, decrement `sound_cut` by `dt`.
- And when it gets to zero... *CUT THE SOUND!*

```

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or x >= 239 - BALL_SZ:
        collision = True
        ball_v[0] = ball_v[0] * -1
        # TODO: beep the "SIDES" tone
    if y <= TOP_WALL + 1 or y >= 239 - BALL_SZ:
        collision = True
        ball_v[1] = ball_v[1] * -1
        # TODO: beep the "TOP" tone
```

Now the *easy* part: Throw in some `beep()`'s!

- Different beep tones for sides and top/bottom will be nice.

```

    if not collision:
        ball_pos = (x, y)
        draw_ball()

```

Hints:

- ## Volume Adjust

  You may want to reduce the volume of the sound effects. An easy way to do that is shown below: just call `tone.set_level()` right after creating the tone.

  ```
  # Sounds
  tone = soundmaker.get_tone('trumpet')
  tone.set_level(15)  # Reduce Volume!
  ```

- ## Beeping

  Are you beeps sounding right?

  - Be sure you're using the 🔧constant pitches. *The values I suggest in the CodeTrek are pretty sweet, but feel free to tune 'em up to your personal preference!*
  - And 🔧indenting the `beep()` call beneath your `if` statement!

  Ex:

  ```
  if x <= 1 or x >= 239 - BALL_SZ:
      collision = True
      beep(SIDES_TONE)
  ```

Goals:

- Create a `tone` 🔧variable using `soundmaker.get_tone('trumpet)`.

  Don't forget `from soundlib import *`

- Define a function `def beep(freq)` that starts playing a *tone* at the specified frequency.

  It will also need to set a 🔧global `sound_cut` timeout value, so the main loop can *stop* the tone later.

- Check the sound timer `sound_cut` inside your *game loop*.
- Add *beeps* when the ball collides with side or top walls.

Tools Found: soundlib, Variables, Locals and Globals, Constants, Loops, Functions

Solution:

```
from codex import *
import time
from soundlib import *

# Screen layout
TOP_WALL = 20
BALL_SZ = 4

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = [0.2, 0.35]
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

draw_screen_layout()
serve_ball()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
```

```
            tone.stop()

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1 or y >= 239 - BALL_SZ:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1

    if not collision:
        ball_pos = (x, y)
        draw_ball()

```

***Objective 7* - Player 1**

Player 1 - *The Paddle*

Your player needs a *paddle* to hit the ball.

- That's too bad.
- The Python language has no such concept.

(...might as well just go home then, right?)

Wait, YOU are the developer!

You can implement any concept imaginable with code!

- Remember from the *Layout* Objective, the paddle is a *blue* rectangle that moves from side to side at the bottom of the screen.
- Feel free to click back on that Objective to see the diagram.

Got Skillz?

Before you open the CodeTrek, think about how you would implement the paddle feature based on the knowledge you already have.

- Can you draw a `filled_rect()` near the bottom of the screen?
- Do you know how to check for `buttons.is_pressed()` ?
- Can you give the paddle a *position* and *velocity* like you did for the ball, and let the buttons control that?

Check the 'Trek!

The paddle is the second *animated* object in your game. Notice the similarity with the code you've written to control the ball.

- Don't just "type-in" the code!
- Understand the purpose of each piece of code here.

Run It!

Experience the *Interactivity!*

- Move the paddle back and forth.
- Try to go past the edges of the screen...
- How does the ball impact the paddle?

CodeTrek:

```
from codex import *
import time
from soundlib import *

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220
```

Size-Up Your Paddle

- It's a *rectangle*, so you need **width** and **height**.
- Set the Y-coordinate near the bottom of screen.

```

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100
```

Initialize the `pad_pos` and `pad_pix` 🔧variables
to track the precise X-coordinate position and pixel location on screen.

- Just like how you already track the *ball*.
- Define a `pad_speed` for the paddle *velocity* when the player presses a button to move the paddle.
- Notice `pad_pix` is *different* than `pad_pos`. Why is that? If they were the same, what would happen on the initial call to `draw_paddle()` below?

If the paddle can move all the way across the screen in under a second, is that fast enough?

```

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)
```

And you'll need a `draw_paddle()` function.

- Looks *very* much like `draw_ball()`, right?

```

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = [0.1, -0.15]
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
```

```
49      diff = time.ticks_diff(now, ms)
50      ms = now
51      return diff
52
53  def draw_screen_layout():
54      display.draw_line(0, TOP_WALL, 0, 239, WHITE)
55      display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
56      display.draw_line(239, TOP_WALL, 239, 239, WHITE)
57      display.draw_text("SCORE", 4, 0, BLUE, 1)
58      display.draw_text("LIVES", 150, 0, BLUE, 1)
59
60  def beep(freq):
61      global sound_cut
62      tone.set_pitch(freq)
63      tone.play()
64      sound_cut = 50  # ms countdown
65
66  def check_buttons():
67      global pad_v
68      if buttons.is_pressed(BTN_L):
69          pad_v = -pad_speed
70      elif buttons.is_pressed(BTN_B):
71          pad_v = +pad_speed
72      else:
73          pad_v = 0  # Stop
```

Define a function to check for *button* presses.

- Its job is to update `pad_v`
- Use the "outside edge" buttons of the CodeX to control the paddle.

```
74
75
76  draw_screen_layout()
77  serve_ball()
78  draw_paddle()
```

Initialize the paddle

- After this you won't draw it unless it moves!

```
79
80  ms = time.ticks_ms()
81
82  while True:
83      dt = elapsed_ms()
84      check_buttons()
```

Check the buttons each time through the game loop!

```
85
86      # Update paddle
87      if pad_v != 0:
88          pad_pos = pad_pos + pad_v * dt
```

Update the `pad_pos`, IF it's *moving*.

- Compare this to how you're updating the `ball_pos`.
- *Same physics calculations!*

```
89          pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
90          draw_paddle()
```

Keep the paddle on-screen.

- Using 🔧built-in `min()` and `max()` functions.
- Don't let `pad_pos` get smaller than 1 or bigger than (238 - PADDLE_W).

```
91
92      # Check sound timer
93      if sound_cut > 0:
94          sound_cut = sound_cut - dt
95          if sound_cut <= 0:
96              tone.stop()
97
98      # Update ball
99      x, y = ball_pos
100     x = x + ball_v[0] * dt
101     y = y + ball_v[1] * dt
102
103     # Check for collision with walls
104     collision = False
105     if x <= 1 or x >= 239 - BALL_SZ:
106         collision = True
107         beep(SIDES_TONE)
108         ball_v[0] = ball_v[0] * -1
109     if y <= TOP_WALL + 1 or y >= 239 - BALL_SZ:
110         collision = True
111         beep(TOP_TONE)
112         ball_v[1] = ball_v[1] * -1
113
114     if not collision:
115         ball_pos = (x, y)
116         draw_ball()
117
118
```

Goals:

- Add constants for paddle width, height, and Y-position.
  - Name them `PADDLE_W`, `PADDLE_H`, and `PADDLE_Y`.
- Define a `def draw_paddle()` function.
  - Erase the old paddle rectangle, and fill-in the new position!
  - Make sure to update your new global `pad_pix` variable.
- Define a `def check_buttons()` function.
  - This should update your global `pad_v` paddle velocity.
  - Call this function each time through your game loop.
- Update the `pad_pos` inside your game loop.
  - Use the `min()` and `max()` built-ins to keep the paddle on-screen.
  - Call `draw_paddle()` after the `pad_pos` is changed.

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

Solution:

```
1  from codex import *
2  import time
3  from soundlib import *
4
5  # Screen layout
6  TOP_WALL = 20
7  BALL_SZ = 4
8  PADDLE_W = 20
9  PADDLE_H = 8
10 PADDLE_Y = 220
11
12 # Sounds
13 tone = soundmaker.get_tone('trumpet')
14 sound_cut = 0 # ms until sound effect stops
```

```
SIDES_TONE = 392
TOP_TONE = 494

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = [0.1, -0.15]
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v
    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop


draw_screen_layout()
serve_ball()
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v != 0:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
```

```
90          draw_paddle()
91
92      # Check sound timer
93      if sound_cut > 0:
94          sound_cut = sound_cut - dt
95          if sound_cut <= 0:
96              tone.stop()
97
98      # Update ball
99      x, y = ball_pos
100     x = x + ball_v[0] * dt
101     y = y + ball_v[1] * dt
102
103     # Check for collision with walls
104     collision = False
105     if x <= 1 or x >= 239 - BALL_SZ:
106         collision = True
107         beep(SIDES_TONE)
108         ball_v[0] = ball_v[0] * -1
109     if y <= TOP_WALL + 1 or y >= 239 - BALL_SZ:
110         collision = True
111         beep(TOP_TONE)
112         ball_v[1] = ball_v[1] * -1
113
114     if not collision:
115         ball_pos = (x, y)
116         draw_ball()
117
118
```

Quiz 3 **- Midway!**

Question 1: Which *three* of the following 🔧comparisons are `True` ?

✓ `(1, 2, 3) == (1, 2, 3)`

✗ `(1, 2, 3) == (3, 2, 1)`

✓ `("Right", "On") == ("Right", "On")`

✓ `10 > 9`

✗ `1 < 0`

✗ `"one" == 1`

Question 2: What's the purpose of the `sound_cut` variable in your Handball program?

✓ To count down the milliseconds till you turn off the sound.

✗ It is the cut-off frequency of the sound.

✗ To count up seconds until sound stops.

Question 3: What is `max(min(3, 2), 1)`

✓ 2

✗ 1

✗ 3

✗ 4

***Objective 8* - Contact**

Stop *Trying* to Hit Me

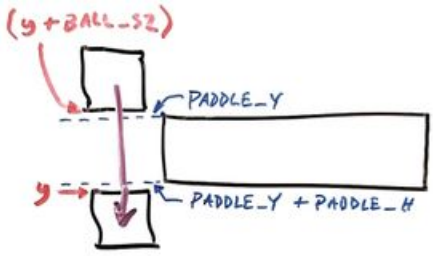
...and *HIT* me!

Enough with the "ghost paddle". It's time to put some *swat* in this thing!

Checking for Collisions

How do you know if the ball has hit the paddle? First check the Y-coordinate. Check the picture below.

Remember, y is the ball's Y-position at the upper left corner.



Imagine the ball traveling down toward the paddle. Y is *increasing*.

- It hits the paddle when `y + BALL_SZ == PADDLE_Y`
- And passes below it when `y == PADDLE_Y + PADDLE_H`

You can check if there is a *potential* collision based on the Y-coordinate with a single `if` statement:

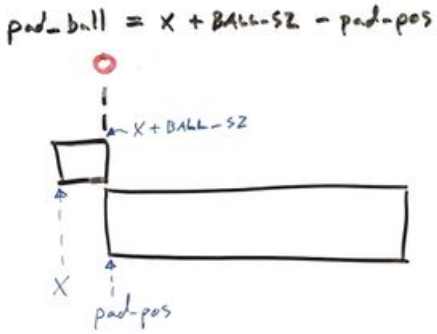
```
if (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
```

Now Check X

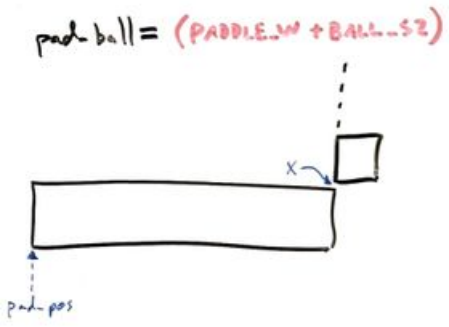
Okay, lets say the ball's Y-coordinate is in the "paddle zone". What X-coordinates would indicate a collision? Track the ball's X position relative to the paddle with a new 🔧variable, `pad_ball`.

- Hitting the paddle's left corner, `pad_ball = 0`
- Hitting the right corner, `pad_ball = PADDLE_W + BALL_SZ`

Left side paddle hit:



Right side paddle hit:



Based on the above diagrams, you can calculate `pad_ball` and check for an X-coordinate `hit` event with the following code:

```
pad_ball = x + BALL_SZ - pad_pos
hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
```

Check the 'Trek!

Now get to it. Just a couple of `if` statements are all that stand between you and a solid paddle that can stand up to any ball!

Run It!

Your ball should now bounce off the paddle!

- It still bounces off the "floor" also... gotta fix that in the next Objective!

CodeTrek:

```
from codex import *
import time
from soundlib import *

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
```

```
39
40  def serve_ball():
41      global ball_pos, ball_v, ball_pix
42      ball_v = [0.1, -0.15]
43      ball_pos = (120.0, 120.0)
44      ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
45
46  def elapsed_ms():
47      """Returns milliseconds elapsed since last called"""
48      global ms
49      now = time.ticks_ms()
50      diff = time.ticks_diff(now, ms)
51      ms = now
52      return diff
53
54  def draw_screen_layout():
55      display.draw_line(0, TOP_WALL, 0, 239, WHITE)
56      display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
57      display.draw_line(239, TOP_WALL, 239, 239, WHITE)
58      display.draw_text("SCORE", 4, 0, BLUE, 1)
59      display.draw_text("LIVES", 150, 0, BLUE, 1)
60
61  def beep(freq):
62      global sound_cut
63      tone.set_pitch(freq)
64      tone.play()
65      sound_cut = 50  # ms countdown
66
67  def check_buttons():
68      global pad_v
69      if buttons.is_pressed(BTN_L):
70          pad_v = -pad_speed
71      elif buttons.is_pressed(BTN_B):
72          pad_v = +pad_speed
73      else:
74          pad_v = 0  # Stop
75
76
77  draw_screen_layout()
78  serve_ball()
79  draw_paddle()
80
81  ms = time.ticks_ms()
82
83  while True:
84      dt = elapsed_ms()
85      check_buttons()
86
87      # Update paddle
88      if pad_v:
89          pad_pos = pad_pos + pad_v * dt
90          pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
91          draw_paddle()
92
93      # Check sound timer
94      if sound_cut > 0:
95          sound_cut = sound_cut - dt
96          if sound_cut <= 0:
97              tone.stop()
98
99      # Update ball
100     x, y = ball_pos
101     x = x + ball_v[0] * dt
102     y = y + ball_v[1] * dt
103
104     # Check for collision with walls
105     collision = False
106     if x <= 1 or x >= 239 - BALL_SZ:
107         collision = True
108         beep(SIDES_TONE)
109         ball_v[0] = ball_v[0] * -1
110     if y <= TOP_WALL + 1 or y >= 239 - BALL_SZ:
111         collision = True
112         beep(TOP_TONE)
113         ball_v[1] = ball_v[1] * -1
```

```
114
115     # Check for collision with paddle
116     if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
```

Paddle Collision Detection

- Check for paddle hit right after you check for *wall collisions.*
- Don't bother checking the paddle if it already hit a wall.

Check if the ball's Y coordinate is in the *paddle zone*.

```
117         # Calculate ball position relative to paddle
118         pad_ball = x + BALL_SZ - pad_pos
119         hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
120         if hit:
```

Now check the X coordinate

- Calculate `pad_ball`, the ball's position *relative to the paddle*.
- Is any part of the ball in contact with the paddle?

```
121             ball_v[1] = ball_v[1] * -1  # bounce
```

Bounce off the paddle!

- Keep the same X velocity, just bounce the Y.
- *Remember, multiply by `-1` to flip the sign and thus the direction of the ball!*

```
122             ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
123             beep(PADDLE_TONE)
124             collision = True
```

Keep it Solid!

- A fast-moving ball might go a few pixels "beneath the surface" of the paddle.
- Keep that from happening by bumping `ball_pos` up to 1 pixel above the paddle.
- Remember, *above* means *lower Y value*, so **subtract** those Y pixels!

Don't forget to `beep()` and flag this as a *collision!*

- Wait, did you define a `PADDLE_TONE` 🔧constant, right?

```
125
126     # Draw ball
127     if not collision:
128         ball_pos = (x, y)
129         draw_ball()
130
```

Goals:

- In your game loop add an `if` statement that checks if the ball is in the Y range of the paddle.
  - It must use `y`, `PADDLE_Y`, `PADDLE_H`, and `BALL_SZ` 🔧constants to test this 🔧condition.
- Create a 🔧variable named `pad_ball` that tracks the ball's X position relative to the paddle.
  - Use `pad_ball` and another 🔧if condition to complete your check for *collision!*

Tools Found: Variables, Constants, bool, Branching

Solution:

```
1  from codex import *
2  import time
3  from soundlib import *
```

```
4
5  # Screen layout
6  TOP_WALL = 20
7  BALL_SZ = 4
8  PADDLE_W = 20
9  PADDLE_H = 8
10 PADDLE_Y = 220
11
12 # Sounds
13 tone = soundmaker.get_tone('trumpet')
14 sound_cut = 0 # ms until sound effect stops
15 SIDES_TONE = 392
16 TOP_TONE = 494
17 PADDLE_TONE = 587
18
19 # Paddle state
20 pad_speed = 0.28  # 280px / 1000ms
21 pad_pos = 110.0  # Paddle X position
22 pad_pix = 100
23
24 def draw_paddle():
25     global pad_pix
26     pix = round(pad_pos)
27     if pix != pad_pix:
28         display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
29         pad_pix = pix
30         display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)
31
32 def draw_ball():
33     global ball_pix
34     pix = (round(ball_pos[0]), round(ball_pos[1]))
35     if pix != ball_pix:
36         display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
37         ball_pix = pix
38         display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
39
40 def serve_ball():
41     global ball_pos, ball_v, ball_pix
42     ball_v = [0.1, -0.15]
43     ball_pos = (120.0, 120.0)
44     ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
45
46 def elapsed_ms():
47     """Returns milliseconds elapsed since last called"""
48     global ms
49     now = time.ticks_ms()
50     diff = time.ticks_diff(now, ms)
51     ms = now
52     return diff
53
54 def draw_screen_layout():
55     display.draw_line(0, TOP_WALL, 0, 239, WHITE)
56     display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
57     display.draw_line(239, TOP_WALL, 239, 239, WHITE)
58     display.draw_text("SCORE", 4, 0, BLUE, 1)
59     display.draw_text("LIVES", 150, 0, BLUE, 1)
60
61 def beep(freq):
62     global sound_cut
63     tone.set_pitch(freq)
64     # tone.play()
65     sound_cut = 50  # ms countdown
66
67 def check_buttons():
68     global pad_v
69     if buttons.is_pressed(BTN_L):
70         pad_v = -pad_speed
71     elif buttons.is_pressed(BTN_B):
72         pad_v = +pad_speed
73     else:
74         pad_v = 0  # Stop
75
76
77 draw_screen_layout()
78 serve_ball()
```

```
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1 or y >= 239 - BALL_SZ:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            ball_v[1] = ball_v[1] * -1  # bounce
            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()

```

***Objective 9* - Missed**

Swing and a Miss!

Is it really a *game* if you can't lose?

Change your game so when the player fails to hit the ball, it zooms off the bottom of the screen.

- After that, your game should wait a few seconds and serve another ball!

Check the 'Trek!

You'll need to change your *wall collision* code to *not* bounce on the bottom of the screen `y >= 239 - BALL_SZ`.

- Instead you'll let it go... BUT when it's past the bottom `y > 240` you'll need to set up for serving a new ball a few seconds later.

CodeTrek:

```
from codex import *
import time
from soundlib import *

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = [0.1, -0.15]
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v
    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
```

```
    else:
        pad_v = 0  # Stop

def new_ball():
    global serve_timer
    serve_timer = 2000
```

Define the `def new_ball()` function.

- Use a 🔧global countdown timer, similar to the `sound_cut` you used to time the *beeps*.

This `serve_timer` will be checked inside your game loop.

```

draw_screen_layout()
new_ball()
draw_paddle()
```

Replace the initial *serve* with your new `new_ball()`

```

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
```

Add a `serve_timer` check in your game loop.

- This looks almost *exactly* like the *sound timer* check above, eh?

```
        else:
            continue
```

Skip the rest of the game loop if you're waiting on a serve.

- The 🔧continue statement jumps back to the top of the 🔧loop.
- Like the `break` statement, it can only be used inside of a loop!

```

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
```

```
121     if y <= TOP_WALL + 1:
122         collision = True
```

Modify your Y wall check to *remove* the bottom wall.

```
123         beep(TOP_TONE)
124         ball_v[1] = ball_v[1] * -1
125     # TODO: otherwise it's a miss! ...get new ball.
```

Add an `elif` 🔧branch here.

- You need to call a 🔧function `new_ball()` when the ball goes off the bottom of the screen.
- That means when `y > 240`...

Next step is to actually *define* the `new_ball()` function!

```
126
127     # Check for collision with paddle
128     if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
129         # Calculate ball position relative to paddle
130         pad_ball = x + BALL_SZ - pad_pos
131         hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
132         if hit:
133             ball_v[1] = ball_v[1] * -1  # bounce
134             ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
135             beep(PADDLE_TONE)
136             collision = True
137
138     # Draw ball
139     if not collision:
140         ball_pos = (x, y)
141         draw_ball()
142
```

Hint:

- The `continue` statement serves a couple of purposes here:
  1. It prevents the ball from continuing to "bounce off the imaginary walls" after it leaves the screen.
  2. Before the first serve, it prevents running code that depends on the initialization from `serve_ball()`.

Goals:

- Modify your Y wall collision code to let the ball go off the bottom.
  - Add an `elif` statement to set up a `new_ball` when the ball goes off-screen.
- Define a function `def new_ball()` that sets a global `serve_timer` countdown in milliseconds.
- Replace your initial call to `serve_ball()` with a call to your new `new_ball()` function.
- Add a serve timer check in your game loop.
  - It should check `serve_timer`, decrement it if needed, and call `serve_ball()`.
  - Use the 🔧continue statement to skip the rest of your game loop while waiting to serve.

Tools Found: Break and Continue, Branching, Functions, Locals and Globals, Loops

Solution:

```
1  from codex import *
2  import time
3  from soundlib import *
4
5  # Screen layout
6  TOP_WALL = 20
```

```
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100


def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = [0.1, -0.15]
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v
    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

def new_ball():
    global serve_timer
    serve_timer = 2000

draw_screen_layout()
```

```
new_ball()
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            ball_v[1] = ball_v[1] * -1  # bounce
            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()

```

***Objective 10* - Score**

Score!

Who's keeping score?

- **Nobody** at the moment. *But I bet you can program the **CodeX** to do it!*

You've made a lovely place at the top of the screen to show the SCORE and the LIVES remaining in the game.

- In this game, LIVES are how many more times a new ball will be served before it's "game over, dude."
- As for SCORE, how about giving the player *one point* each time they hit the ball with the paddle?

You can do it! Give your game a *real-live* **scoreboard!**

Check the 'Trek!

Watch for `# TODO`s. Read the code carefully and be sure to follow exactly how the `score` and `n_lives` are tracked.

*When this is running properly, you'll have a **playable** game!*

Run It!

Play your game!

- Is the SCORE updating when you hit the ball?
- How about the LIVES?

If not, it's time to debug and fix it. Next objective you'll be making the game a little more *user-friendly!*

CodeTrek:

```
1  from codex import *
2  import time
3  from soundlib import *
4
5  # Screen layout
6  TOP_WALL = 20
7  BALL_SZ = 4
8  PADDLE_W = 20
9  PADDLE_H = 8
10 PADDLE_Y = 220
11
12 # Sounds
13 tone = soundmaker.get_tone('trumpet')
14 sound_cut = 0 # ms until sound effect stops
15 SIDES_TONE = 392
16 TOP_TONE = 494
17 PADDLE_TONE = 587
18
19 # Paddle state
20 pad_speed = 0.28  # 280px / 1000ms
21 pad_pos = 110.0  # Paddle X position
22 pad_pix = 100
23
24 # Game state
25 START_LIVES = 3  # Lives remaining at start of game
26 score = 0
27 n_lives = START_LIVES + 1
28 serve_timer = 2000
```

Initialize variables to keep track of the `score` and number of "lives" remaining, `n_lives`.

- Add 1 to the initial `n_lives` value for the first ball.
- Every new ball will consume a "life", and the game ends when `n_lives == 0`.

```
29
30 def draw_paddle():
31     global pad_pix
32     pix = round(pad_pos)
33     if pix != pad_pix:
34         display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
35         pad_pix = pix
```

```
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = [0.1, -0.15]
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v
    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

def new_ball():
    global serve_timer # TODO: another global?
    # TODO: subtract 1 from n_lives
    update_score()
    if n_lives > 0:
        serve_timer = 2000
```

Modify your `new_ball()` function to update `n_lives`.

- You are modifying a 🔧global, so add `n_lives` to the `global` list!
- Take 1 from `n_lives` each time a new ball is served.
- After you change `n_lives`, call the new function `update_score()`.
- Only reset the `serve_timer` if there are lives remaining.

```

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)
```

Define a new 🔧function to update the score.

- Erase the text area at the top of the screen.
- Draw updated values, using `str()` to convert the 🔧ints to 🔧strings.

```
 94
 95  draw_screen_layout()
 96  new_ball()
 97  draw_paddle()
 98
 99  ms = time.ticks_ms()
100
101  while True:
102      dt = elapsed_ms()
103      check_buttons()
104
105      # Update paddle
106      if pad_v:
107          pad_pos = pad_pos + pad_v * dt
108          pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
109          draw_paddle()
110
111      # Check sound timer
112      if sound_cut > 0:
113          sound_cut = sound_cut - dt
114          if sound_cut <= 0:
115              tone.stop()
116
117      # Check serve timer
118      if serve_timer > 0:
119          serve_timer = serve_timer - dt
120          if serve_timer <= 0:
121              serve_ball()
122          else:
123              continue
124
125      if n_lives == 0:
126          continue
```

Skip the rest of the game loop if there are no lives remaining.

- This still lets the player move the paddle.
- Later you can add the ability to reset for a new serve...

```
127
128      # Update ball
129      x, y = ball_pos
130      x = x + ball_v[0] * dt
131      y = y + ball_v[1] * dt
132
133      # Check for collision with walls
134      collision = False
135      if x <= 1 or 240 > x >= 239 - BALL_SZ:
136          collision = True
137          beep(SIDES_TONE)
138          ball_v[0] = ball_v[0] * -1
139      if y <= TOP_WALL + 1:
140          collision = True
141          beep(TOP_TONE)
142          ball_v[1] = ball_v[1] * -1
143      elif y > 240:
144          new_ball()
145
146      # Check for collision with paddle
147      if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
148          # Calculate ball position relative to paddle
149          pad_ball = x + BALL_SZ - pad_pos
150          hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
151          if hit:
152              ball_v[1] = ball_v[1] * -1  # bounce
153              ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
154              beep(PADDLE_TONE)
155              collision = True
156              score = score + 1
157              update_score()
```

Score a point when you hit a ball!

```
158
159     # Draw ball
160     if not collision:
161         ball_pos = (x, y)
162         draw_ball()
163
164
```

Goals:

- Initialize 🔧variables called `score` and `n_lives` at the beginning of your program where you define the *`# Game state`*.
- Modify the `new_ball()` function.
    - Decrement `n_lives`
    - Only reset `serve_timer` if `n_lives > 0`
- Define an `def update_score()` 🔧function that displays the value of the `score` and `n_lives` 🔧variables at the top of the screen.
- Score a point when the player hits the ball!

Tools Found: Variables, Functions, Locals and Globals, int, str

Solution:

```
1  from codex import *
2  import time
3  from soundlib import *
4
5  # Screen layout
6  TOP_WALL = 20
7  BALL_SZ = 4
8  PADDLE_W = 20
9  PADDLE_H = 8
10 PADDLE_Y = 220
11
12 # Sounds
13 tone = soundmaker.get_tone('trumpet')
14 sound_cut = 0 # ms until sound effect stops
15 SIDES_TONE = 392
16 TOP_TONE = 494
17 PADDLE_TONE = 587
18
19 # Paddle state
20 pad_speed = 0.28  # 280px / 1000ms
21 pad_pos = 110.0  # Paddle X position
22 pad_pix = 100
23
24 # Game state
25 START_LIVES = 3  # Lives remaining at start of game
26 score = 0
27 n_lives = START_LIVES + 1
28 serve_timer = 2000
29
30 def draw_paddle():
31     global pad_pix
32     pix = round(pad_pos)
33     if pix != pad_pix:
34         display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
35         pad_pix = pix
36         display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)
37
38 def draw_ball():
39     global ball_pix
40     pix = (round(ball_pos[0]), round(ball_pos[1]))
41     if pix != ball_pix:
42         display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
43         ball_pix = pix
44         display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
```

```
45
46  def serve_ball():
47      global ball_pos, ball_v, ball_pix
48      ball_v = [0.1, -0.15]
49      ball_pos = (120.0, 120.0)
50      ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
51
52  def elapsed_ms():
53      """Returns milliseconds elapsed since last called"""
54      global ms
55      now = time.ticks_ms()
56      diff = time.ticks_diff(now, ms)
57      ms = now
58      return diff
59
60  def draw_screen_layout():
61      display.draw_line(0, TOP_WALL, 0, 239, WHITE)
62      display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
63      display.draw_line(239, TOP_WALL, 239, 239, WHITE)
64      display.draw_text("SCORE", 4, 0, BLUE, 1)
65      display.draw_text("LIVES", 150, 0, BLUE, 1)
66
67  def beep(freq):
68      global sound_cut
69      tone.set_pitch(freq)
70      tone.play()
71      sound_cut = 50  # ms countdown
72
73  def check_buttons():
74      global pad_v
75      if buttons.is_pressed(BTN_L):
76          pad_v = -pad_speed
77      elif buttons.is_pressed(BTN_B):
78          pad_v = +pad_speed
79      else:
80          pad_v = 0  # Stop
81
82  def new_ball():
83      global n_lives, serve_timer
84      n_lives = n_lives - 1
85      update_score()
86      if n_lives > 0:
87          serve_timer = 2000
88
89  def update_score():
90      display.fill_rect(45, 0, 100, 20, BLACK)
91      display.draw_text(str(score), 45, 0, WHITE, 2)
92      display.fill_rect(195, 0, 45, 20, BLACK)
93      display.draw_text(str(n_lives), 195, 0, WHITE, 2)
94
95  draw_screen_layout()
96  new_ball()
97  draw_paddle()
98
99  ms = time.ticks_ms()
100
101 while True:
102     dt = elapsed_ms()
103     check_buttons()
104
105     # Update paddle
106     if pad_v:
107         pad_pos = pad_pos + pad_v * dt
108         pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
109         draw_paddle()
110
111     # Check sound timer
112     if sound_cut > 0:
113         sound_cut = sound_cut - dt
114         if sound_cut <= 0:
115             tone.stop()
116
117     # Check serve timer
118     if serve_timer > 0:
119         serve_timer = serve_timer - dt
```

```
120         if serve_timer <= 0:
121             serve_ball()
122         else:
123             continue
124 
125     if n_lives == 0:
126         continue
127 
128     # Update ball
129     x, y = ball_pos
130     x = x + ball_v[0] * dt
131     y = y + ball_v[1] * dt
132 
133     # Check for collision with walls
134     collision = False
135     if x <= 1 or 240 > x >= 239 - BALL_SZ:
136         collision = True
137         beep(SIDES_TONE)
138         ball_v[0] = ball_v[0] * -1
139     if y <= TOP_WALL + 1:
140         collision = True
141         beep(TOP_TONE)
142         ball_v[1] = ball_v[1] * -1
143     elif y > 240:
144         new_ball()
145 
146     # Check for collision with paddle
147     if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
148         # Calculate ball position relative to paddle
149         pad_ball = x + BALL_SZ - pad_pos
150         hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
151         if hit:
152             ball_v[1] = ball_v[1] * -1  # bounce
153             ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
154             beep(PADDLE_TONE)
155             collision = True
156             score = score + 1
157             update_score()
158 
159     # Draw ball
160     if not collision:
161         ball_pos = (x, y)
162         draw_ball()
163 
164 
```

***Objective 11* - Messages**

UX

It's time to focus on your game's UX (*User Experience*).

- Add friendly messages to inform the player about what's happening.
- Give the user a "Play Again" button.

Check the 'Trek!

You'll be defining functions to display/clear a message near the center of the screen.

Also you will add a check for *Button U* to **Play Again.**

CodeTrek:

```
1  from codex import *
2  import time
3  from soundlib import *
4 
5  # Screen layout
6  TOP_WALL = 20
```

```
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = [0.1, -0.15]
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()
```

Okay NOW you see why the `clear_message()` function was separate from `show_message()`.

- The message stays up *until the serve happens*.

```

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score
```

```
76
77      if buttons.is_pressed(BTN_L):
78          pad_v = -pad_speed
79      elif buttons.is_pressed(BTN_B):
80          pad_v = +pad_speed
81      else:
82          pad_v = 0  # Stop
83
84      if n_lives == 0 and buttons.is_pressed(BTN_U):
85          n_lives = START_LIVES + 1
86          score = 0
```

Add the "Play Again" feature to your `check_buttons()` function.

- You only want to allow this when `n_lives == 0`.
- Reset both `n_lives` and `score` 🔧globals for a new game!

```
87
88  def new_ball():
89      global n_lives, serve_timer
90      n_lives = n_lives - 1
91      update_score()
92      if n_lives > 0:
93          serve_timer = 2000
94          show_message("Serving...", "Get Ready!", GREEN)
95      else:
96          show_message("Game Over!", "U = play again", RED)
```

Add some friendly messages!

- You're about to get *served!*
- ...and *Game Ovah!*

```
97
98  def update_score():
99      display.fill_rect(45, 0, 100, 20, BLACK)
100     display.draw_text(str(score), 45, 0, WHITE, 2)
101     display.fill_rect(195, 0, 45, 20, BLACK)
102     display.draw_text(str(n_lives), 195, 0, WHITE, 2)
103
104 def clear_message():
105     display.fill_rect(1, 120, 238, 80, BLACK)
106
107 def show_message(banner, note, color):
108     clear_message()
109     display.draw_text(banner, 30, 120, color, 3)
110     display.draw_text(note, 30, 160, WHITE, 2)
```

Define *two* new functions:

1. A function to put a text message in the middle of the screen:

```
show_message(
    banner,  # Short message title
    note,    # Long message subtitle
    color    # Color of the banner
)
```

2. `clear_message()` to erase the message. *(scroll up to see)*

```
111
112 draw_screen_layout()
113 new_ball()
114 draw_paddle()
115
116 ms = time.ticks_ms()
117
118 while True:
119     dt = elapsed_ms()
120     check_buttons()
121
```

```
    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            ball_v[1] = ball_v[1] * -1  # bounce
            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True
            score = score + 1
            update_score()

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()


```

Goals:

- Define a new 🔧function `def show_message(banner, note, color)` that draws a colorful message in the middle of the screen, with a small *subtitle* in WHITE text.
- Define a new function `def clear_message()` that erases the message area used by `show_message()`.
- Add calls to `show_message()` to your `new_ball()` function.
- Check for the "Play Again" button `BTN_U` in your `check_buttons()` function.
    - Reset the 🔧global `n_lives` when it's pressed.

- Call `clear_message()` from your `serve_ball()` function.

Tools Found: Functions, Locals and Globals

Solution:

```
from codex import *
import time
from soundlib import *

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = [0.1, -0.15]
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

```

```
68  def beep(freq):
69      global sound_cut
70      tone.set_pitch(freq)
71      tone.play()
72      sound_cut = 50  # ms countdown
73
74  def check_buttons():
75      global pad_v, n_lives, score
76
77      if buttons.is_pressed(BTN_L):
78          pad_v = -pad_speed
79      elif buttons.is_pressed(BTN_B):
80          pad_v = +pad_speed
81      else:
82          pad_v = 0  # Stop
83
84      if n_lives == 0 and buttons.is_pressed(BTN_U):
85          n_lives = START_LIVES + 1
86          score = 0
87
88  def new_ball():
89      global n_lives, serve_timer
90      n_lives = n_lives - 1
91      update_score()
92      if n_lives > 0:
93          serve_timer = 2000
94          show_message("Serving...", "Get Ready!", GREEN)
95      else:
96          show_message("Game Over!", "U = play again", RED)
97
98  def update_score():
99      display.fill_rect(45, 0, 100, 20, BLACK)
100     display.draw_text(str(score), 45, 0, WHITE, 2)
101     display.fill_rect(195, 0, 45, 20, BLACK)
102     display.draw_text(str(n_lives), 195, 0, WHITE, 2)
103
104 def clear_message():
105     display.fill_rect(1, 120, 238, 80, BLACK)
106
107 def show_message(banner, note, color):
108     clear_message()
109     display.draw_text(banner, 30, 120, color, 3)
110     display.draw_text(note, 30, 160, WHITE, 2)
111
112 draw_screen_layout()
113 new_ball()
114 draw_paddle()
115
116 ms = time.ticks_ms()
117
118 while True:
119     dt = elapsed_ms()
120     check_buttons()
121
122     # Update paddle
123     if pad_v:
124         pad_pos = pad_pos + pad_v * dt
125         pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
126         draw_paddle()
127
128     # Check sound timer
129     if sound_cut > 0:
130         sound_cut = sound_cut - dt
131         if sound_cut <= 0:
132             tone.stop()
133
134     # Check serve timer
135     if serve_timer > 0:
136         serve_timer = serve_timer - dt
137         if serve_timer <= 0:
138             serve_ball()
139         else:
140             continue
141
142     if n_lives == 0:
```

```
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            ball_v[1] = ball_v[1] * -1  # bounce
            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True
            score = score + 1
            update_score()

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()


```

Quiz 4 **- Continue**

Question 1: What is the value of `count` after the following code runs?

```
count = 0
x = 0
while x < 5:
    x = x + 1
    if x == 2:
        continue
    count = count + 1
```

✓ 4

✗ 5

✗ 3

Objective 12 **- Angles**

A Sweet *Angle!*

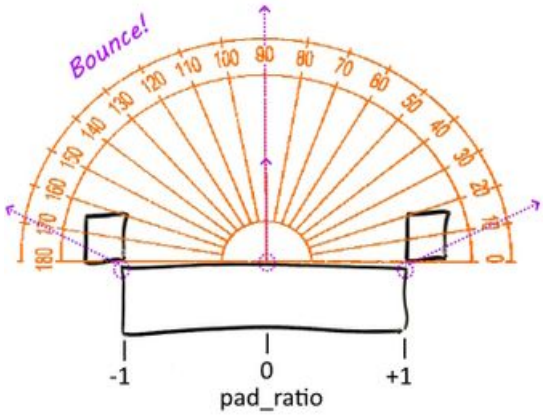
There's just one thing left to fix in your Handball game.

- The paddle has *no control* over how the ball bounces!
- How cool would it be for a skilled player to be able to shoot the ball wherever they wanted?

If you can achieve that, the game will be *awesome!*

Dialed-In Rebounds

Let the player control the rebound angle based on where the ball hits the paddle. Check out the purple *bounce angles* below.



- If the ball hits the CENTER of the paddle → angle=90°
- If the ball hits the RIGHT corner → angle=30°
- If the ball hits the LEFT corner → angle=150°

Your code already has `pad_ball`, the ball position relative to paddle. You just need to convert that to a 🔧float `pad_ratio` between -1.0 and +1.0, and then convert *that* to an angle!

Check the 'Trek!

The `hit_ball(angle)` 🔧function is your key to precise ball control.

- Since you can now direct the ball *anywhere*, this would be a great time to fix the BORING serves the game has been sending up to now...

Run It!

Your game is now complete.

- Play a few rounds of Handball, and enjoy the fruits of your labor!

If you load batteries and go *unplugged* with this game, check the Hints panel for a bug-fix you'll want to add.

CodeTrek:

```
1  from codex import *
2  import time
3  from soundlib import *
4  import math
5  import random
```

Don't forget to `import` the *math* and *random* libraries for your jazzy new angles.

```
6
7  # Screen layout
8  TOP_WALL = 20
9  BALL_SZ = 4
10 PADDLE_W = 20
11 PADDLE_H = 8
12 PADDLE_Y = 220
13
14 # Sounds
15 tone = soundmaker.get_tone('trumpet')
16 sound_cut = 0 # ms until sound effect stops
17 SIDES_TONE = 392
```

```
TOP_TONE = 494
PADDLE_TONE = 587

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    ball_v = [0.1, -0.15]  # Could be [0,0]. hit_ball() overrides this.
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

    # Hit ball toward paddle at a random angle
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
```

No more BORING serves!

- A negative `angle` will hit the ball toward the paddle.
- Pick an angle that's not *too* off-center. (*Maybe I'm being too nice to our player. What do you think?*)

Notice that `hit_ball()` is going to replace the initial value of `ball_v[]` you set above. That's okay, the list has to get initialized somewhere so you can leave it as-is.

```

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global v_pad, n_lives
```

```
83
84      if buttons.is_pressed(BTN_L):
85          v_pad = -pad_speed
86      elif buttons.is_pressed(BTN_B):
87          v_pad = +pad_speed
88      else:
89          v_pad = 0  # Stop
90
91      if n_lives == 0 and buttons.is_pressed(BTN_U):
92          n_lives = START_LIVES + 1
93
94  def new_ball():
95      global n_lives, serve_timer
96      n_lives = n_lives - 1
97      update_score()
98      if n_lives > 0:
99          serve_timer = 2000
100         show_message("Serving...", "Get Ready!", GREEN)
101     else:
102         show_message("Game Over!", "U = play again", RED)
103
104 def update_score():
105     display.fill_rect(45, 0, 100, 20, BLACK)
106     display.draw_text(str(score), 45, 0, WHITE, 2)
107     display.fill_rect(195, 0, 45, 20, BLACK)
108     display.draw_text(str(n_lives), 195, 0, WHITE, 2)
109
110 def clear_message():
111     display.fill_rect(1, 120, 238, 80, BLACK)
112
113 def show_message(banner, note, color):
114     clear_message()
115     display.draw_text(banner, 30, 120, color, 3)
116     display.draw_text(note, 30, 160, WHITE, 2)
117
118 def hit_ball(angle):
119     """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
120     angle = angle * math.pi / 180
121     ball_v[0] = math.cos(angle) * ball_speed
122     ball_v[1] = -math.sin(angle) * ball_speed
```

Define a new 🔧function `def hit_ball(angle)`.

- This will set the ball's *velocity* so it moves in the direction given by `angle`.
- The 🔧global `ball_speed` sets the *magnitude* of velocity, regardless of its direction.
- The *cosine* function sets the X velocity `ball_v[0]`.
- The *sine* function sets the Y velocity `ball_v[1]`.

See the 💡 Hints panel for details on the math if you're interested!

```
123
124 draw_screen_layout()
125 new_ball()
126 draw_paddle()
127
128 ms = time.ticks_ms()
129
130 while True:
131     dt = elapsed_ms()
132     check_buttons()
133
134     # Update paddle
135     if v_pad:
136         pad_pos = pad_pos + v_pad * dt
137         pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
138         draw_paddle()
139
140     # Check sound timer
141     if sound_cut > 0:
142         sound_cut = sound_cut - dt
143         if sound_cut <= 0:
144             tone.stop()
145
146     # Check serve timer
```

```
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            # Bounce direction based on paddle position
            center = (PADDLE_W + BALL_SZ) / 2
            pad_ratio = (pad_ball - center) / center  # range -1 to +1
            angle = 90 - 60 * pad_ratio
            hit_ball(angle)
```

Replace the simple bounce with *player controlled rebound!*

- Find the paddle `center` and use it to calculate `pad_ratio`.
- Use that to get the `angle`: 90° at the center, ±60° to the corners.

See the diagram in the Objective panel for more detail on how `pad_ratio` relates to the `angle`.

```
            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True
            score = score + 1
            update_score()

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()


```

Hints:

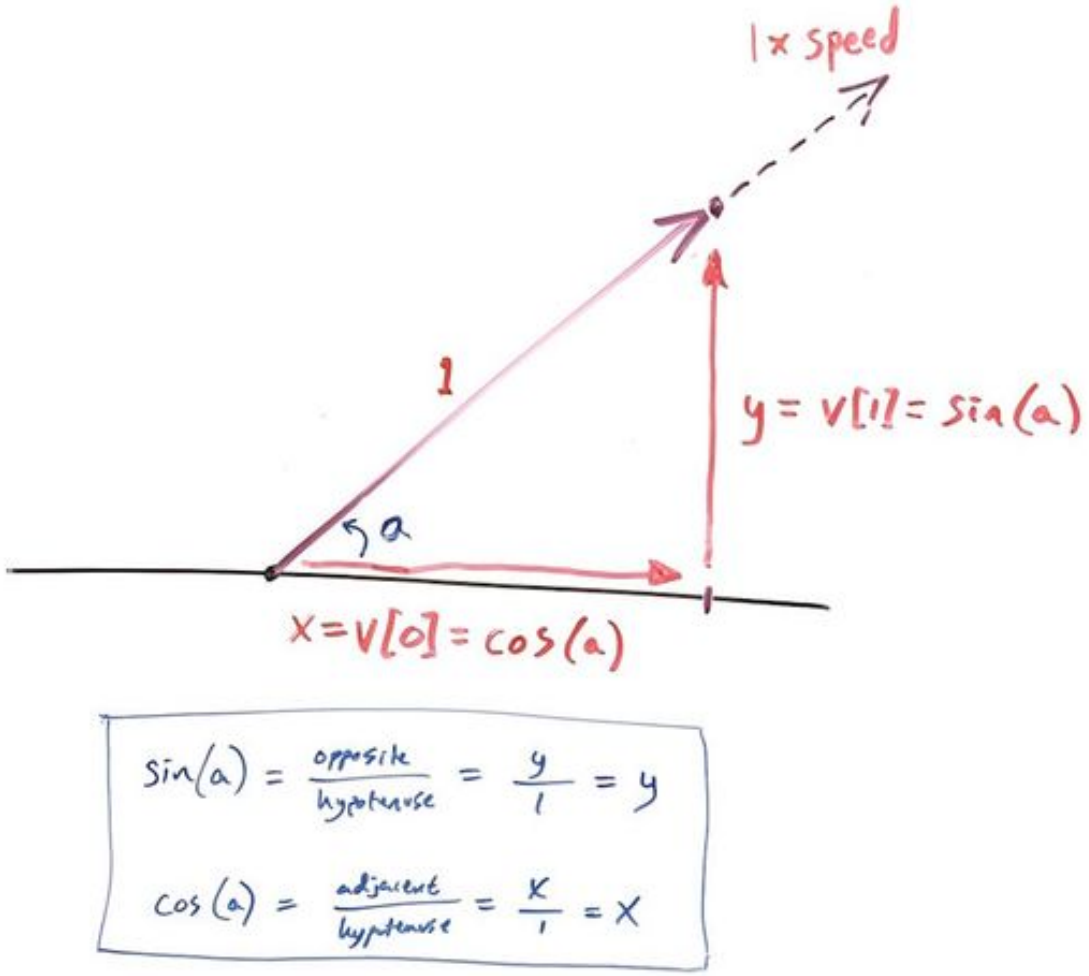
- ## How Works *hit_ball(angle)* ?

  Curious about the code in this magic little 🔧function? Well, you gotta do a little math to make the ball move juuuust how you want it to!

  - In this case you want the ball to move at a particular angle.
  - You just need to calculate the X and Y *velocity* values to achieve that!

Check out the triangle diagram below. Making the ball move on the purple path at angle "a" requires an X and Y component of velocity, like the RED sides of the triangle.

- The purple path is the *hypotenuse* of the triangle. Its length represents the *speed* of the ball. Consider that to be 1.0 for now.
- So if you know the *angle* and one side of a right triangle, how can you find the other sides?



Holy *SOHCAHTOA*!

The `hit_ball(angle)` function uses a bit of *trigonometry*. If you haven't learned about that yet, don't let the name scare you - it's pretty cool stuff :-)

You'll find the equations for X and Y above match the calculations in `hit_ball(angle)`. Both the X and Y values are multiplied by `ball_speed` since you don't want to leave it at 1. 0. Also, since the UP direction on the screen is *negative* you'll notice the Y is negated in the function.

Oh, one more thing: Python's trigonometry functions use *radians* for angle measurement, rather than *degrees*. The first line of code in `hit_ball(angle)` converts from degrees to radians. *(180° equals PI radians)*

Python's 🔧math library has lots of helpful functions, including the trigonometry you need for bodacious ball bounces!

Battery Operated Bugz

As of the time this Mission was published the CodeX firmware has a bug in its internal initialization of the buttons.

- When you run from batteries, the buttons may not work properly until you press the *RESET* button.

To fix this, just add the following function call after your `from codex import *`. *(comment is optional as always)*

```
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)
```

Goals:

- Define a new function `def hit_ball(angle)` that sets new `ball_v[]` X and Y components based on the `angle` (in degrees) you'd like the ball to fly.
    - Add a new 🔧variable `ball_speed` as part of your initial *Game State*.
- Mix up the Serve!

  In your `serve_ball()` function, use `randrange()` to select an angle and `hit_ball()` to *thwack* the ball toward the paddle!
- 🔧import the **math** and **random** libraries.
- Use `hit_ball(angle)` rather than simple *bounce* inside your game loop for *paddle collision*.

Tools Found: float, Functions, Variables, import, Locals and Globals

Solution:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

```

```
50  def serve_ball():
51      global ball_pos, ball_v, ball_pix
52      # Set ball_v: serve toward paddle
53      ball_v = [0,0]
54      angle = random.randrange(-60, -120, -1)
55      hit_ball(angle)
56      ball_pos = (120.0, 120.0)
57      ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
58      clear_message()
59
60  def elapsed_ms():
61      """Returns milliseconds elapsed since last called"""
62      global ms
63      now = time.ticks_ms()
64      diff = time.ticks_diff(now, ms)
65      ms = now
66      return diff
67
68  def draw_screen_layout():
69      display.draw_line(0, TOP_WALL, 0, 239, WHITE)
70      display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
71      display.draw_line(239, TOP_WALL, 239, 239, WHITE)
72      display.draw_text("SCORE", 4, 0, BLUE, 1)
73      display.draw_text("LIVES", 150, 0, BLUE, 1)
74
75  def beep(freq):
76      global sound_cut
77      tone.set_pitch(freq)
78      tone.play()
79      sound_cut = 50  # ms countdown
80
81  def check_buttons():
82      global v_pad, n_lives, score
83
84      if buttons.is_pressed(BTN_L):
85          v_pad = -pad_speed
86      elif buttons.is_pressed(BTN_B):
87          v_pad = +pad_speed
88      else:
89          v_pad = 0  # Stop
90
91      if n_lives == 0 and buttons.is_pressed(BTN_U):
92          n_lives = START_LIVES + 1
93          score = 0
94
95  def new_ball():
96      global n_lives, serve_timer
97      n_lives = n_lives - 1
98      update_score()
99      if n_lives > 0:
100         serve_timer = 2000
101         show_message("Serving...", "Get Ready!", GREEN)
102     else:
103         show_message("Game Over!", "U = play again", RED)
104
105 def update_score():
106     display.fill_rect(45, 0, 100, 20, BLACK)
107     display.draw_text(str(score), 45, 0, WHITE, 2)
108     display.fill_rect(195, 0, 45, 20, BLACK)
109     display.draw_text(str(n_lives), 195, 0, WHITE, 2)
110
111 def clear_message():
112     display.fill_rect(1, 120, 238, 80, BLACK)
113
114 def show_message(banner, note, color):
115     clear_message()
116     display.draw_text(banner, 30, 120, color, 3)
117     display.draw_text(note, 30, 160, WHITE, 2)
118
119 def hit_ball(angle):
120     """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
121     angle = angle * math.pi / 180
122     ball_v[0] = math.cos(angle) * ball_speed
123     ball_v[1] = -math.sin(angle) * ball_speed
124
```

```
draw_screen_layout()
new_ball()
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if v_pad:
        pad_pos = pad_pos + v_pad * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            # Bounce direction based on paddle position
            center = (PADDLE_W + BALL_SZ) / 2
            pad_ratio = (pad_ball - center) / center  # range -1 to +1
            angle = 90 - 60 * pad_ratio
            hit_ball(angle)

            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True
            score = score + 1
            update_score()

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()


```

Mission 15 Complete

Radical!

It is SO cool that you've built this game FROM SCRATCH!

- There are so many ways you could extend this!

Get ready for the next Mission, where you'll continue the journey of Arcade Archaeology...

Mission 16 - Break Out

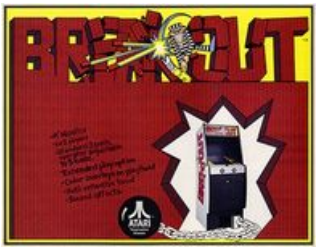
Breakout!

Now that you've conquered *Handball* you are all set to code one of the all-time arcade classics!

History

The concept for **Breakout** came from **Atari** founder Nolan Bushnell, who wanted a single-player game to follow up the 1972 smash-hit **Pong** - one of the first video games many people encountered.

He gave the challenge to young ***Steve Jobs***, who recruited his friend ***Steve Wozniak*** to implement the game. Do those names sound familiar? *They went on to found* ***Apple Computers!!***



Breakout hit the arcades in 1976, becoming one of the top earning arcade video games that year. That means a lot of players dropped quarters (25 cents per turn) into arcade cabinets like the one shown at right.

Your Task

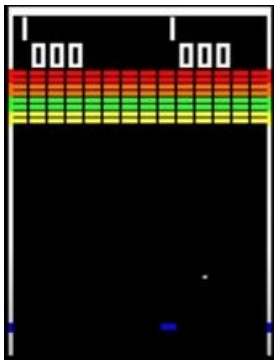
...is to follow in the footsteps of Jobs and Wozniak. Imagine that you've been tasked by Atari's CEO to create the next hit game for the company. *Ready to break some bricks?*

Objective 1 - Prototype

Breakout Begins!

This mission starts where the previous *Handball* Mission left off. You'll need to use *Save As...* to save your Handball code to a new file for *Breakout*.

The game Breakout adds 8 rows of bricks as shown at right (original Atari Arcade screen).



- Two rows each, from the top down: RED, ORANGE, GREEN, YELLOW.
- Horizontally there are 14 bricks in each row in the original game. Since the CodeX screen is smaller than the original, 10 bricks across will do.

By the end of this mission the player will be able to score points by smashing bricks! Different color bricks are worth different points. *More on that later!*

Prototyping

To start with, just try drawing the bricks to the screen as shown. This will be a "prototype" of the real thing. Sure, the bricks won't DO anything yet...

- But it will be SO inspiring to see the *fully-rendered* game arena!
- And hey, it's just a bunch of rectangles. *So why not!?*

Open the Last Handball File

You DID get Handball running, right?

Save to a New File!

Use the **File** → **Save As** menu to create a new file called ***Breakout***.

Check the 'Trek!

The CodeTrek will guide you like a master bricklayer!

- This is *prototyping.*
- You can play around with the size of the bricks, colors, spacing, etc.

Run It!

Nice looking bricks, eh?

- It's kinda fun driving the ball through them.
- ...and it will be even MORE fun to smash those bricks for *points!*

CodeTrek:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
```

```
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

def show_message(banner, note, color):
    clear_message()
    display.draw_text(banner, 30, 120, color, 3)
    display.draw_text(note, 30, 160, WHITE, 2)

def hit_ball(angle):
    """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
    angle = angle * math.pi / 180
    ball_v[0] = math.cos(angle) * ball_speed
    ball_v[1] = -math.sin(angle) * ball_speed

def brick_row(y, color):
    """Lay down a row of bricks. Experiment with size."""
    for x in range(2, 240, 24):
        display.fill_rect(x, y, 20, 6, color)
```

Define a function `def brick_row(y, color)` that draws one *row* of bricks.

- Start at `x=2` to make room for the left wall, and 1 pixel gap.
- Make each brick a rectangle 20 pixels wide, plus a 4 pixel gap.

You can fit 10 of these bricks across the screen!

```
130 def draw_bricks():
131     """Place 8 rows of bricks. """
132     brick_row(30, RED)
133     brick_row(40, RED)
134     brick_row(50, ORANGE)
135     brick_row(60, ORANGE)
136     brick_row(70, GREEN)
137     brick_row(80, GREEN)
138     brick_row(90, YELLOW)
139     brick_row(100, YELLOW)
```

Define a function `def draw_bricks()`

- Start at a Y-coordinate of `y=30` and place a row every 10 pixels down.
- Since the bricks are 6 pixels high, this will leave a 4 pixel gap between rows.
- Use *colors* just like the original Breakout game!

```
140
141 draw_bricks()
```

Don't forget to call your `draw_bricks()` function in your initialization code.

```
142
143 draw_screen_layout()
144 new_ball()
145 draw_paddle()
146
147 ms = time.ticks_ms()
148
149 while True:
150     dt = elapsed_ms()
151     check_buttons()
152
153     # Update paddle
154     if pad_v:
155         pad_pos = pad_pos + pad_v * dt
156         pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
157         draw_paddle()
158
159     # Check sound timer
160     if sound_cut > 0:
161         sound_cut = sound_cut - dt
162         if sound_cut <= 0:
163             tone.stop()
164
165     # Check serve timer
166     if serve_timer > 0:
167         serve_timer = serve_timer - dt
168         if serve_timer <= 0:
169             serve_ball()
170         else:
171             continue
172
173     if n_lives == 0:
174         continue
175
176     # Update ball
177     x, y = ball_pos
178     x = x + ball_v[0] * dt
179     y = y + ball_v[1] * dt
180
181     # Check for collision with walls
182     collision = False
183     if x <= 1 or 240 > x >= 239 - BALL_SZ:
184         collision = True
185         beep(SIDES_TONE)
186         ball_v[0] = ball_v[0] * -1
187     if y <= TOP_WALL + 1:
188         collision = True
189         beep(TOP_TONE)
190         ball_v[1] = ball_v[1] * -1
191     elif y > 240:
```

```
192         new_ball()
193
194     # Check for collision with paddle
195     if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
196         # Calculate ball position relative to paddle
197         pad_ball = x + BALL_SZ - pad_pos
198         hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
199         if hit:
200             # Bounce direction based on paddle position
201             center = (PADDLE_W + BALL_SZ) / 2
202             pad_ratio = (pad_ball - center) / center  # range -1 to +1
203             angle = 90 - 60 * pad_ratio
204             hit_ball(angle)
205
206             ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
207             beep(PADDLE_TONE)
208             collision = True
209             score = score + 1
210             update_score()
211
212     # Draw ball
213     if not collision:
214         ball_pos = (x, y)
215         draw_ball()
216
217
```

Hint:

- ## Prototyping *Pro Tip*

  Ancient engineering wisdom warns about showing "too realistic" prototypes to your boss.

  - They may not fully understand the technology side of things, and get the idea that the project must be 99% finished, since it looks so good!

  ### What Say You?

  Do you think you're 99% finished implementing *Breakout* at this point?

  Hmmmm....

Goals:

- Define a function `def brick_row(y, color)` that draws one *row* of bricks.
- Define a function `def draw_bricks()` that draws 8 rows of bricks, by calling `brick_row()` eight times.
- Call the `draw_bricks()` function in your initialization code.

Solution:

```
 1  from codex import *
 2  import time
 3  from soundlib import *
 4  import math
 5  import random
 6  ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)
 7
 8  # Screen layout
 9  TOP_WALL = 20
10  BALL_SZ = 4
11  PADDLE_W = 20
12  PADDLE_H = 8
13  PADDLE_Y = 220
14
15  # Sounds
16  tone = soundmaker.get_tone('trumpet')
17  sound_cut = 0 # ms until sound effect stops
18  SIDES_TONE = 392
19  TOP_TONE = 494
20  PADDLE_TONE = 587
```

```
21
22  # Paddle state
23  pad_speed = 0.28  # 280px / 1000ms
24  pad_pos = 110.0  # Paddle X position
25  pad_pix = 100
26
27  # Game state
28  START_LIVES = 3  # Lives remaining at start of game
29  score = 0
30  n_lives = START_LIVES + 1
31  serve_timer = 2000
32  ball_speed = 0.15   # 150 pixels per second
33
34  def draw_paddle():
35      global pad_pix
36      pix = round(pad_pos)
37      if pix != pad_pix:
38          display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
39          pad_pix = pix
40          display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)
41
42  def draw_ball():
43      global ball_pix
44      pix = (round(ball_pos[0]), round(ball_pos[1]))
45      if pix != ball_pix:
46          display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
47          ball_pix = pix
48          display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
49
50  def serve_ball():
51      global ball_pos, ball_v, ball_pix
52      # Set ball_v: serve toward paddle
53      ball_v = [0,0]
54      angle = random.randrange(-60, -120, -1)
55      hit_ball(angle)
56      ball_pos = (120.0, 120.0)
57      ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
58      clear_message()
59
60  def elapsed_ms():
61      """Returns milliseconds elapsed since last called"""
62      global ms
63      now = time.ticks_ms()
64      diff = time.ticks_diff(now, ms)
65      ms = now
66      return diff
67
68  def draw_screen_layout():
69      display.draw_line(0, TOP_WALL, 0, 239, WHITE)
70      display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
71      display.draw_line(239, TOP_WALL, 239, 239, WHITE)
72      display.draw_text("SCORE", 4, 0, BLUE, 1)
73      display.draw_text("LIVES", 150, 0, BLUE, 1)
74
75  def beep(freq):
76      global sound_cut
77      tone.set_pitch(freq)
78      tone.play()
79      sound_cut = 50  # ms countdown
80
81  def check_buttons():
82      global pad_v, n_lives, score
83
84      if buttons.is_pressed(BTN_L):
85          pad_v = -pad_speed
86      elif buttons.is_pressed(BTN_B):
87          pad_v = +pad_speed
88      else:
89          pad_v = 0  # Stop
90
91      if n_lives == 0 and buttons.is_pressed(BTN_U):
92          n_lives = START_LIVES + 1
93          score = 0
94
95  def new_ball():
```

```
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

def show_message(banner, note, color):
    clear_message()
    display.draw_text(banner, 30, 120, color, 3)
    display.draw_text(note, 30, 160, WHITE, 2)

def hit_ball(angle):
    """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
    angle = angle * math.pi / 180
    ball_v[0] = math.cos(angle) * ball_speed
    ball_v[1] = -math.sin(angle) * ball_speed

def brick_row(y, color):
    """Lay down a row of bricks. Experiment with size."""
    for x in range(2, 240, 24):
        display.fill_rect(x, y, 20, 6, color)

def draw_bricks():
    """Place 8 rows of bricks. """
    brick_row(30, RED)
    brick_row(40, RED)
    brick_row(50, ORANGE)
    brick_row(60, ORANGE)
    brick_row(70, GREEN)
    brick_row(80, GREEN)
    brick_row(90, YELLOW)
    brick_row(100, YELLOW)

draw_bricks()

draw_screen_layout()
new_ball()
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
```

```
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            # Bounce direction based on paddle position
            center = (PADDLE_W + BALL_SZ) / 2
            pad_ratio = (pad_ball - center) / center  # range -1 to +1
            angle = 90 - 60 * pad_ratio
            hit_ball(angle)

            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True
            score = score + 1
            update_score()

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()


```

***Objective 2* - Matrix**

Enter the Matrix!

Nice looking prototype! Now, how are you going to make those *bricks* interact with the game? You gotta:

- Detect when the ball hits a brick.
- Destroy the brick when it gets hit!

Collision Detection

Should you test every pixel the ball is about to hit? Using `display.get_pixel(x,y)` to read the color of each pixel on the screen is one strategy.

- But that's a LOT of pixel tests, every time through your game loop.
- It would be much faster to check using *boundaries*, like you did with the walls.
- And look! The *gaps* between rows and columns of bricks make a grid of wall-like boundaries.
- *Checking if the ball is inside a "grid square" should be pretty easy!*

Got Bricks?: *True or False*

At the start there's a brick in each grid square.

- But after you start *smashing bricks,* some of them will be gone!

- The game needs a way to track whether there's a brick in each square.

A row with ten bricks: a list of bools:

```
row = [True, True, True, True, True, True, True, True, True, True]
```

You could make a list for each row of bricks, as shown above.

- Then if the ball hits a brick, say the 4th one from the left, set `row[3] = False` to mark it destroyed.
- And when the ball enters a grid square `i` on this row, you can test for a brick with `row[i] == True`.
- *But you have more than one row of bricks...*

Matrix: a list of rows

Breakout has 8 *rows* X 10 *columns* of bricks. A 2D array like this is called a *"matrix"*.

- Check out this matrix of bools laying over the bricks.

i=	j = 0	1	2	3	4	5	6	7	8	9
0	True	True	True	True	True	True	True	True	True	True
1	True	True	True	True	True	True	True	True	True	True
2	True	True	True	True	True	True	True	True	True	True
3	True	True	True	True	True	True	True	True	True	True
4	True	True	True	True	True	True	True	True	True	True
5	True	True	True	True	True	True	True	True	True	True
6	True	True	True	True	True	True	True	False	True	True
7	True	True	True	False	False	True	False	False	True	True

To create a *matrix* like this in Python use a list of lists! It's just a list of *rows* like the `row` example above.

```
# The Brick Matrix
bricks = [
    [True, True, True, True, True, True, True, True, True, True],
    [True, True, True, True, True, True, True, True, True, True],
    [True, True, True, True, True, True, True, True, True, True],
    [True, True, True, True, True, True, True, True, True, True],
    [True, True, True, True, True, True, True, True, True, True],
    [True, True, True, True, True, True, True, True, True, True],
    [True, True, True, True, True, True, True, True, True, True],
    [True, True, True, True, True, True, True, True, True, True],
]
```

Above is one way to *initialize* your brick matrix. But the CodeTrek will show you a *better* way, that requires less typing and ensures the right number of *rows* and *columns!*

Check the 'Trek!

Time for you to create your own *matrix!*

Run It!

Open the ☰ Console and gaze into the matrix!

CodeTrek:

```
1  from codex import *
2  import time
3  from soundlib import *
4  import math
5  import random
6  ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)
7
8  # Screen layout
9  TOP_WALL = 20
```

```
10  BALL_SZ = 4
11  PADDLE_W = 20
12  PADDLE_H = 8
13  PADDLE_Y = 220
14
15  # Sounds
16  tone = soundmaker.get_tone('trumpet')
17  sound_cut = 0 # ms until sound effect stops
18  SIDES_TONE = 392
19  TOP_TONE = 494
20  PADDLE_TONE = 587
21
22  # Paddle state
23  pad_speed = 0.28  # 280px / 1000ms
24  pad_pos = 110.0  # Paddle X position
25  pad_pix = 100
26
27  # Game state
28  START_LIVES = 3  # Lives remaining at start of game
29  score = 0
30  n_lives = START_LIVES + 1
31  serve_timer = 2000
32  ball_speed = 0.15   # 150 pixels per second
33
34  # Bricks
35  BRICKS_ACROSS = 10
36  BRICKS_DOWN = 8
```

Add 🔧constants for the number of *columns across* and *rows down* of bricks.

```
37
38  def draw_paddle():
39      global pad_pix
40      pix = round(pad_pos)
41      if pix != pad_pix:
42          display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
43          pad_pix = pix
44          display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)
45
46  def draw_ball():
47      global ball_pix
48      pix = (round(ball_pos[0]), round(ball_pos[1]))
49      if pix != ball_pix:
50          display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
51          ball_pix = pix
52          display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
53
54  def serve_ball():
55      global ball_pos, ball_v, ball_pix
56      # Set ball_v: serve toward paddle
57      ball_v = [0,0]
58      angle = random.randrange(-60, -120, -1)
59      hit_ball(angle)
60      ball_pos = (120.0, 120.0)
61      ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
62      clear_message()
63
64  def elapsed_ms():
65      """Returns milliseconds elapsed since last called"""
66      global ms
67      now = time.ticks_ms()
68      diff = time.ticks_diff(now, ms)
69      ms = now
70      return diff
71
72  def draw_screen_layout():
73      display.draw_line(0, TOP_WALL, 0, 239, WHITE)
74      display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
75      display.draw_line(239, TOP_WALL, 239, 239, WHITE)
76      display.draw_text("SCORE", 4, 0, BLUE, 1)
77      display.draw_text("LIVES", 150, 0, BLUE, 1)
78
79  def beep(freq):
```

```
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

def show_message(banner, note, color):
    clear_message()
    display.draw_text(banner, 30, 120, color, 3)
    display.draw_text(note, 30, 160, WHITE, 2)

def hit_ball(angle):
    """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
    angle = angle * math.pi / 180
    ball_v[0] = math.cos(angle) * ball_speed
    ball_v[1] = -math.sin(angle) * ball_speed

def brick_row(y, color):
    """Lay down a row of bricks. Experiment with size."""
    for x in range(2, 240, 24):
        display.fill_rect(x, y, 20, 6, color)

def draw_bricks():
    """Place 8 rows of bricks. """
    brick_row(30, RED)
    brick_row(40, RED)
    brick_row(50, ORANGE)
    brick_row(60, ORANGE)
    brick_row(70, GREEN)
    brick_row(80, GREEN)
    brick_row(90, YELLOW)
    brick_row(100, YELLOW)

def setup_bricks():
    global bricks
    bricks = []  # Empty matrix (list of rows)
    for i in range(BRICKS_DOWN):
        bricks.append([])  # Empty row (list of columns)
        for j in range(BRICKS_ACROSS):
            bricks[i].append(True)  # Add column to this row
```

Define a `def setup_bricks()` function:

- A 🔧loop of 🔧loops
- To build a 🔧list of 🔧lists !

The *outer loop* `i` is **rows**, and the *inner loop* `j` is **columns**.

See the 🔧list tool to learn more about the `append()` function.

```
152
153 setup_bricks()
154 i = ??  # TODO: row of brick to hit
155 j = ??  # TODO: column of brick to hit
156 bricks[i][j] = False  # Mark one brick as destroyed!
157 print("bricks=", bricks)
```

Add some *test code*

- Set the `i=` and `j=` values to the ***row*** and ***column*** specified in your Objective Goals.
- Setting that specific brick to `False` is how your *future* code will mark it as having been destroyed by the ball.

For now this will have no *visible* effect, so just use `print()` to dump the matrix to the **console** so CodeSpace can check your work!

```
158
159 draw_bricks()
160
161 draw_screen_layout()
162 new_ball()
163 draw_paddle()
164
165 ms = time.ticks_ms()
166
167 while True:
168     dt = elapsed_ms()
169     check_buttons()
170
171     # Update paddle
172     if pad_v:
173         pad_pos = pad_pos + pad_v * dt
174         pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
175         draw_paddle()
176
177     # Check sound timer
178     if sound_cut > 0:
179         sound_cut = sound_cut - dt
180         if sound_cut <= 0:
181             tone.stop()
182
183     # Check serve timer
184     if serve_timer > 0:
185         serve_timer = serve_timer - dt
186         if serve_timer <= 0:
187             serve_ball()
188         else:
189             continue
190
191     if n_lives == 0:
192         continue
193
194     # Update ball
195     x, y = ball_pos
196     x = x + ball_v[0] * dt
197     y = y + ball_v[1] * dt
198
199     # Check for collision with walls
200     collision = False
201     if x <= 1 or 240 > x >= 239 - BALL_SZ:
202         collision = True
203         beep(SIDES_TONE)
204         ball_v[0] = ball_v[0] * -1
205     if y <= TOP_WALL + 1:
206         collision = True
207         beep(TOP_TONE)
208         ball_v[1] = ball_v[1] * -1
209     elif y > 240:
```

```
210         new_ball()
211
212     # Check for collision with paddle
213     if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
214         # Calculate ball position relative to paddle
215         pad_ball = x + BALL_SZ - pad_pos
216         hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
217         if hit:
218             # Bounce direction based on paddle position
219             center = (PADDLE_W + BALL_SZ) / 2
220             pad_ratio = (pad_ball - center) / center  # range -1 to +1
221             angle = 90 - 60 * pad_ratio
222             hit_ball(angle)
223
224             ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
225             beep(PADDLE_TONE)
226             collision = True
227             score = score + 1
228             update_score()
229
230     # Draw ball
231     if not collision:
232         ball_pos = (x, y)
233         draw_ball()
234
235
```

Goals:

- Define a function `def setup_bricks()` that initializes a 🔧global `bricks` to a 2D array of 🔧bools containing 8 rows X 10 columns of `True`.
- Use the `print()` statement to display your 🔧bool matrix on the ☰ Console.
    - Set the brick at row 7, column 3 to `False` before printing the matrix.

Tools Found: list, bool, Locals and Globals, Constants, Loops

Solution:

```
1  from codex import *
2  import time
3  from soundlib import *
4  import math
5  import random
6  ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)
7
8  # Screen layout
9  TOP_WALL = 20
10 BALL_SZ = 4
11 PADDLE_W = 20
12 PADDLE_H = 8
13 PADDLE_Y = 220
14
15 # Sounds
16 tone = soundmaker.get_tone('trumpet')
17 sound_cut = 0 # ms until sound effect stops
18 SIDES_TONE = 392
19 TOP_TONE = 494
20 PADDLE_TONE = 587
21
22 # Paddle state
23 pad_speed = 0.28  # 280px / 1000ms
24 pad_pos = 110.0  # Paddle X position
25 pad_pix = 100
26
27 # Game state
28 START_LIVES = 3  # Lives remaining at start of game
29 score = 0
30 n_lives = START_LIVES + 1
31 serve_timer = 2000
```

```
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
```

```
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

def show_message(banner, note, color):
    clear_message()
    display.draw_text(banner, 30, 120, color, 3)
    display.draw_text(note, 30, 160, WHITE, 2)

def hit_ball(angle):
    """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
    angle = angle * math.pi / 180
    ball_v[0] = math.cos(angle) * ball_speed
    ball_v[1] = -math.sin(angle) * ball_speed

def brick_row(y, color):
    """Lay down a row of bricks. Experiment with size."""
    for x in range(2, 240, 24):
        display.fill_rect(x, y, 20, 6, color)

def draw_bricks():
    """Place 8 rows of bricks. """
    brick_row(30, RED)
    brick_row(40, RED)
    brick_row(50, ORANGE)
    brick_row(60, ORANGE)
    brick_row(70, GREEN)
    brick_row(80, GREEN)
    brick_row(90, YELLOW)
    brick_row(100, YELLOW)

def setup_bricks():
    global bricks
    bricks = []  # Empty matrix (list of rows)
    for i in range(BRICKS_DOWN):
        bricks.append([])  # Empty row (list of columns)
        for j in range(BRICKS_ACROSS):
            bricks[i].append(True)  # Add column to this row  #@1

setup_bricks()
i = 7  # TODO: row of brick to hit
j = 3  # TODO: column of brick to hit
bricks[i][j] = False  # Mark one brick as destroyed!
print("bricks=", bricks)

draw_bricks()

draw_screen_layout()
new_ball()
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()
```

```
182
183     # Check serve timer
184     if serve_timer > 0:
185         serve_timer = serve_timer - dt
186         if serve_timer <= 0:
187             serve_ball()
188         else:
189             continue
190
191     if n_lives == 0:
192         continue
193
194     # Update ball
195     x, y = ball_pos
196     x = x + ball_v[0] * dt
197     y = y + ball_v[1] * dt
198
199     # Check for collision with walls
200     collision = False
201     if x <= 1 or 240 > x >= 239 - BALL_SZ:
202         collision = True
203         beep(SIDES_TONE)
204         ball_v[0] = ball_v[0] * -1
205     if y <= TOP_WALL + 1:
206         collision = True
207         beep(TOP_TONE)
208         ball_v[1] = ball_v[1] * -1
209     elif y > 240:
210         new_ball()
211
212     # Check for collision with paddle
213     if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
214         # Calculate ball position relative to paddle
215         pad_ball = x + BALL_SZ - pad_pos
216         hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
217         if hit:
218             # Bounce direction based on paddle position
219             center = (PADDLE_W + BALL_SZ) / 2
220             pad_ratio = (pad_ball - center) / center  # range -1 to +1
221             angle = 90 - 60 * pad_ratio
222             hit_ball(angle)
223
224             ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
225             beep(PADDLE_TONE)
226             collision = True
227             score = score + 1
228             update_score()
229
230     # Draw ball
231     if not collision:
232         ball_pos = (x, y)
233         draw_ball()
234
235
```

Quiz 1 **- Making of the Matrix**

For a card game, I've created the following matrix. The cards will be laid out in rows and columns.

- Face up → `True`
- Face down → `False`

```
cards = [
    [True, False, True],
    [True, True, True],
    [False, False, True],
    [True, False, False],
]
```

Question 1: How many *rows* are in the `cards` matrix?

✓ 4

✗ 2

✗ 3

Question 2: How many *columns* are in the `cards` matrix?

✓ 3

✗ 2

✗ 4

Question 3: Is the card at `cards[2][1]` *Face up* or *Face down*?

✓ Face down

✗ Face up

Question 4: What is the value of `my_list` after the following code runs?

```
my_list = [5, 4, 9]
my_list.append(2)
```

✓ `[5, 4, 9, 2]`

✗ `[2, 5, 4, 9]`

✗ `[5, 4, 9, [2]]`

✗ `[5, 4, 9], [5, 4, 9]`

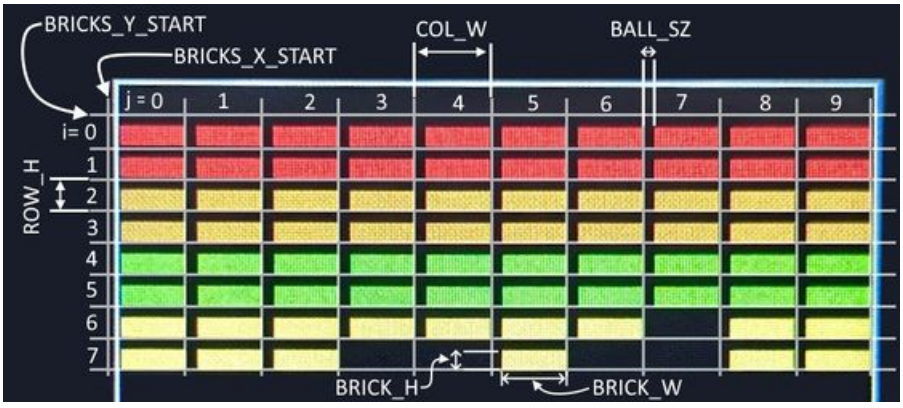
Objective 3 **- Brick Layer**

Brick by Brick

It's time to say goodbye to your "prototype" bricks, and rebuild them *matrix-style* so they can be fully controlled by your code.

Measuring Up the Matrix

Your prototype shows you the dimensions for the bricks, but you need to make 🔧constants for those. You already have `BALL_SZ` for spacing the rows and columns. The figure below shows some additional 🔧constants you'll need to define.



Based on your prototype, `BRICKS_Y_START = 30`. That's the top of the first row of bricks. Notice there's also a `BALL_SZ` **gap** *above* and to the *left* of each brick.

- This gap size simplifies *collision detection*.
- If the ball X,Y coordinates land in one of those grid squares, it is colliding with the brick!

See where `BRICKS_X_START` is pointing? It's to the left of the wall, and that wall is at `x=0` !

- When your code calculates where to draw the first brick, it will be starting from `BRICKS_X_START = -2` then adding the `BALL_SZ` gap to get the X coordinate.

Check the 'Trek!

The CodeTrek will guide you to defining all those cool 🔧constants.

- Then you'll make a 🔧function that drops bricks *exactly* where you want them!

CodeTrek:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
BRICKS_Y_START = 30
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE,... ) # TODO: finish colors
```

Size it up! These 🔧constants define the size, spacing, position, and color of bricks in the game.

- The colors are in a 🔧tuple, so you can use the *row* `i` as an index to find the color of a row of bricks.
- There should be exactly 8 items in this 🔧tuple. Be sure to match the color of each row in the original *Breakout* game!

```

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
```

```
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

```

```
125 def show_message(banner, note, color):
126     clear_message()
127     display.draw_text(banner, 30, 120, color, 3)
128     display.draw_text(note, 30, 160, WHITE, 2)
129 
130 def hit_ball(angle):
131     """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
132     angle = angle * math.pi / 180
133     ball_v[0] = math.cos(angle) * ball_speed
134     ball_v[1] = -math.sin(angle) * ball_speed
135 
136 # Removed function def brick_row()
137 # Removed function def draw_bricks()
```

Delete your *prototype* functions.

- Don't cry. They have served you well.
- *Thank them,* and send them to the *bit bucket!*

```
138 
139 def setup_bricks():
140     global bricks
141     bricks = []  # Empty matrix (list of rows)
142     for i in range(BRICKS_DOWN):
143         bricks.append([])  # Empty row (list of columns)
144         for j in range(BRICKS_ACROSS):
145             bricks[i].append(True)  # Add column to this row
146             brick_place(i, j, BRICK_COLORS[i])
```

You're already looping over each brick location!

- Just add `brick_place()` to your inner loop.
- Notice how this uses the *row* to select a color from `BRICK_COLORS`.

```
147 
148 def brick_place(i, j, color):
149     """Draw a brick at the given row,column matrix location"""
150     x = BRICKS_X_START + j * COL_W + BALL_SZ
151     y = BRICKS_Y_START + i * ROW_H + BALL_SZ
152     display.fill_rect(x, y, BRICK_W, BRICK_H, color)
```

A function to draw a brick right where you want it!

- Just a little *math* to convert a *column* to X, and a *row* to Y.
- The *multiply* happens first, based on the 🔧precedence rules.

Consider the X calculation:

1. Start at the left edge: `BRICKS_X_START`.
2. Move across the screen to the specified column: `j * COL_W`.
3. Jump over the gap: `BALL_SZ`.

```
153 
154 setup_bricks()
155 
156 # Removed bricks test / print
157 # Removed draw_bricks()
```

Remove the test code AND The call to `draw_bricks()`

- Your `setup_bricks()` function is doing the work now!

```
158 
159 draw_screen_layout()
160 new_ball()
161 draw_paddle()
162 
163 ms = time.ticks_ms()
164 
165 while True:
```

```
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            # Bounce direction based on paddle position
            center = (PADDLE_W + BALL_SZ) / 2
            pad_ratio = (pad_ball - center) / center  # range -1 to +1
            angle = 90 - 60 * pad_ratio
            hit_ball(angle)

            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True
            score = score + 1
            update_score()

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()


```

Hints:

- ## The Hard Part

  At this point you're having to *THINK* about how the game is going to work!

> *The **hard part** is NOT that you're learning new **Python** concepts.* It is more about thinking through the logical decisions at each step as the game runs.

Take your time, and review the diagrams in this Objective. Go back to the previous Objective if needed. On paper, test different values of (x,y) for the ball position, and calculate what (i,j) that would be in the matrix.

Always Hard, But Glorious

Any software application you build will present its own set of challenges. Often you will have to stop, scribble some notes down, draw some diagrams, and walk away to ruminate about the solution. The approach I'm showing you here for *Breakout* is no different. Several sketches and different approaches were considered, before landing on the method described here.

A great joy of software engineering is the feeling of discovering a nice solution to a tricky problem. The fact that it's hard makes overcoming it even more gratifying!

- ## (x, y) versus (i, j)

When you're plotting points on a 2D graph, it's very common to use X and Y coordinates.

- Same goes for plotting *pixels* on the screen!
- X is horizontal (across), and Y is vertical (down)
- And those coordinates are always in (x,y) order like: `set_pixel(x, y, color)`

But dealing with a matrix, the order is different!

- You specify the *row* first, then the *column.*
- The location of an item in the matrix is specified by the pair (i, j).
- i=row, and j=column
- In Python you would write: `bricks[i][j]`

Together in Harmony!

When you think about the direction: X → j , and Y → i

- Moving the ball in **X** crosses the columns, **j**.
- Moving the ball in **Y** crosses the rows, **i**.

Goals:

- Create 🔧constants for the brick width, height, and other dimensions shown in the diagram. *Use the 🔧variable names in the CodeTrek*
- Define a function `def brick_place(i, j, color):` that draws a brick at the specified matrix location.
- Call the new `brick_place()` function inside `setup_bricks()` to draw the bricks as you build the matrix.
- Remove the test code that `print`s the matrix, and remove the call to `draw_bricks()`.
- Delete your *prototype* functions `def draw_bricks()` and `def brick_row()`.

Tools Found: Constants, Functions, Variables, tuple, Math Operators

Solution:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
```

```
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first column
BRICKS_Y_START = 30
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row #@1

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

```

```
 86  def beep(freq):
 87      global sound_cut
 88      tone.set_pitch(freq)
 89      tone.play()
 90      sound_cut = 50  # ms countdown
 91
 92  def check_buttons():
 93      global pad_v, n_lives, score
 94
 95      if buttons.is_pressed(BTN_L):
 96          pad_v = -pad_speed
 97      elif buttons.is_pressed(BTN_B):
 98          pad_v = +pad_speed
 99      else:
100          pad_v = 0  # Stop
101
102      if n_lives == 0 and buttons.is_pressed(BTN_U):
103          n_lives = START_LIVES + 1
104          score = 0
105
106  def new_ball():
107      global n_lives, serve_timer
108      n_lives = n_lives - 1
109      update_score()
110      if n_lives > 0:
111          serve_timer = 2000
112          show_message("Serving...", "Get Ready!", GREEN)
113      else:
114          show_message("Game Over!", "U = play again", RED)
115
116  def update_score():
117      display.fill_rect(45, 0, 100, 20, BLACK)
118      display.draw_text(str(score), 45, 0, WHITE, 2)
119      display.fill_rect(195, 0, 45, 20, BLACK)
120      display.draw_text(str(n_lives), 195, 0, WHITE, 2)
121
122  def clear_message():
123      display.fill_rect(1, 120, 238, 80, BLACK)
124
125  def show_message(banner, note, color):
126      clear_message()
127      display.draw_text(banner, 30, 120, color, 3)
128      display.draw_text(note, 30, 160, WHITE, 2)
129
130  def hit_ball(angle):
131      """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
132      angle = angle * math.pi / 180
133      ball_v[0] = math.cos(angle) * ball_speed
134      ball_v[1] = -math.sin(angle) * ball_speed
135
136  # Removed function def brick_row()
137  # Removed function def draw_bricks() #@5
138
139  def setup_bricks():
140      global bricks
141      bricks = []  # Empty matrix (list of rows)
142      for i in range(BRICKS_DOWN):
143          bricks.append([])  # Empty row (list of columns)
144          for j in range(BRICKS_ACROSS):
145              bricks[i].append(True)  # Add column to this row
146              brick_place(i, j, BRICK_COLORS[i])  #@3
147
148  def brick_place(i, j, color):
149      """Draw a brick at the given row,column matrix location"""
150      x = BRICKS_X_START + j * COL_W + BALL_SZ
151      y = BRICKS_Y_START + i * ROW_H + BALL_SZ
152      display.fill_rect(x, y, BRICK_W, BRICK_H, color) #@2
153
154  setup_bricks()
155
156  # Removed bricks test / print
157  # Removed draw_bricks()  #@4
158
159  draw_screen_layout()
160  new_ball()
```

```
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            # Bounce direction based on paddle position
            center = (PADDLE_W + BALL_SZ) / 2
            pad_ratio = (pad_ball - center) / center  # range -1 to +1
            angle = 90 - 60 * pad_ratio
            hit_ball(angle)

            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True
            score = score + 1
            update_score()

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()


```

***Objective 4* - Collision!**

Smash!

You're all set now to detect when the ball hits a brick, and take action!

- All you have to do is convert the ball's x, y pixel position to an i, j matrix location. Then consult your `bricks` matrix: it will tell you `True` or `False` whether a brick is there or not!

At this point just focus on detecting the collision and removing the brick when it has been hit. *Later you'll bounce the ball off the brick, but for now allow it to cruise on.*

Check the 'Trek!

Watch out for some changes you may need to make for code to run without error.

Run It!

This game is kinda fun as-is! *Blast some bricks!*

- Can you clear them all?

CodeTrek:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587
BRICK_TONE = 740

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
BRICKS_Y_START = 30
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row
```

```
45
46  def draw_paddle():
47      global pad_pix
48      pix = round(pad_pos)
49      if pix != pad_pix:
50          display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
51          pad_pix = pix
52          display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)
53
54  def draw_ball():
55      global ball_pix
56      pix = (round(ball_pos[0]), round(ball_pos[1]))
57      if pix != ball_pix:
58          display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
59          ball_pix = pix
60          display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
61
62  def serve_ball():
63      global ball_pos, ball_v, ball_pix
64      # Set ball_v: serve toward paddle
65      ball_v = [0,0]
66      angle = random.randrange(-60, -120, -1)
67      hit_ball(angle)
68      ball_pos = (120.0, 120.0)
69      ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
70      clear_message()
71
72  def elapsed_ms():
73      """Returns milliseconds elapsed since last called"""
74      global ms
75      now = time.ticks_ms()
76      diff = time.ticks_diff(now, ms)
77      ms = now
78      return diff
79
80  def draw_screen_layout():
81      display.draw_line(0, TOP_WALL, 0, 239, WHITE)
82      display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
83      display.draw_line(239, TOP_WALL, 239, 239, WHITE)
84      display.draw_text("SCORE", 4, 0, BLUE, 1)
85      display.draw_text("LIVES", 150, 0, BLUE, 1)
86
87  def beep(freq):
88      global sound_cut
89      tone.set_pitch(freq)
90      tone.play()
91      sound_cut = 50  # ms countdown
92
93  def check_buttons():
94      global pad_v, n_lives, score
95
96      if buttons.is_pressed(BTN_L):
97          pad_v = -pad_speed
98      elif buttons.is_pressed(BTN_B):
99          pad_v = +pad_speed
100     else:
101         pad_v = 0  # Stop
102
103     if n_lives == 0 and buttons.is_pressed(BTN_U):
104         n_lives = START_LIVES + 1
105         score = 0
106
107 def new_ball():
108     global n_lives, serve_timer
109     n_lives = n_lives - 1
110     update_score()
111     if n_lives > 0:
112         serve_timer = 2000
113         show_message("Serving...", "Get Ready!", GREEN)
114     else:
115         show_message("Game Over!", "U = play again", RED)
116
117 def update_score():
118     display.fill_rect(45, 0, 100, 20, BLACK)
119     display.draw_text(str(score), 45, 0, WHITE, 2)
```

```
120     display.fill_rect(195, 0, 45, 20, BLACK)
121     display.draw_text(str(n_lives), 195, 0, WHITE, 2)
122
123 def clear_message():
124     display.fill_rect(1, 120, 238, 80, BLACK)
125
126 def show_message(banner, note, color):
127     clear_message()
128     display.draw_text(banner, 30, 120, color, 3)
129     display.draw_text(note, 30, 160, WHITE, 2)
130
131 def hit_ball(angle):
132     """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
133     angle = angle * math.pi / 180
134     ball_v[0] = math.cos(angle) * ball_speed
135     ball_v[1] = -math.sin(angle) * ball_speed
136
137 def setup_bricks():
138     global bricks
139     bricks = []  # Empty matrix (list of rows)
140     for i in range(BRICKS_DOWN):
141         bricks.append([])  # Empty row (list of columns)
142         for j in range(BRICKS_ACROSS):
143             bricks[i].append(True)  # Add column to this row
144             brick_place(i, j, BRICK_COLORS[i])
145
146 def brick_place(i, j, color):
147     """Draw a brick at the given row,column matrix location"""
148     x = BRICKS_X_START + j * COL_W + BALL_SZ
149     y = BRICKS_Y_START + i * ROW_H + BALL_SZ
150     display.fill_rect(x, y, BRICK_W, BRICK_H, color)
151
152 def check_bricks(x, y):
153     """Check for ball collision, return 'collided' True/False"""
154     collided = False
155
156     # Calculate row and column based on ball x,y
157     i = (y - BRICKS_Y_START) / ROW_H  # row
158     j = (x - BRICKS_X_START) / COL_W  # column
159     # TODO: Modify above to truncate i and j to int
```

Begin your `check_bricks()` function by calculating which grid square the ball is in.

- The grid *divides* the brick area into `COL_W x ROW_H` pixel sections.
- Calculate `i` by dividing `y`'s position in the grid by `ROW_H`.
- ...same thing for `j`, using `x` and `COL_W`.

You're gonna need a little more code here, if you plan to use `i` and `j` to index 🔧lists.
They have to be converted to 🔧*int*, right?

```
160
161     # Is ball inside the brick grid?
162     if 0 <= i < BRICKS_DOWN and 0 <= j < BRICKS_ACROSS:
163         collided = bricks[i][j]  # Is there a brick here?
```

Is the ball inside the grid?

- Hey, if the ball is down near the bottom of the screen then there's no point checking any further!

Isn't it cool how Python 🔧comparison lets you check the *lower* and *upper* range of a variable?
And the 🔧logical operator `and` makes this code *so* readable!

Colliding?

If it's in the grid, let the *matrix* do the work! Just grab the 🔧bool corresponding to this brick (i, j) and you have your answer!

```
164
```

```
165     if collided:
166         # Destroy brick
167         bricks[i][j] = False
168         brick_place(i, j, BLACK) # Erase
169         beep(BRICK_TONE)
170 
171     return collided
```

Destructo!

Erase this brick, and flag it as destroyed in the matrix.

- A satisfying "beep" will increase the joy!
- You *did* define a `BRICK_TONE` constant near the top of the file, right?

```
172 
173 setup_bricks()
174 draw_screen_layout()
175 new_ball()
176 draw_paddle()
177 
178 ms = time.ticks_ms()
179 
180 while True:
181     dt = elapsed_ms()
182     check_buttons()
183 
184     # Update paddle
185     if pad_v:
186         pad_pos = pad_pos + pad_v * dt
187         pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
188         draw_paddle()
189 
190     # Check sound timer
191     if sound_cut > 0:
192         sound_cut = sound_cut - dt
193         if sound_cut <= 0:
194             tone.stop()
195 
196     # Check serve timer
197     if serve_timer > 0:
198         serve_timer = serve_timer - dt
199         if serve_timer <= 0:
200             serve_ball()
201         else:
202             continue
203 
204     if n_lives == 0:
205         continue
206 
207     # Update ball
208     x, y = ball_pos
209     x = x + ball_v[0] * dt
210     y = y + ball_v[1] * dt
211 
212     # Check for collision with walls
213     collision = False
214     if x <= 1 or 240 > x >= 239 - BALL_SZ:
215         collision = True
216         beep(SIDES_TONE)
217         ball_v[0] = ball_v[0] * -1
218     if y <= TOP_WALL + 1:
219         collision = True
220         beep(TOP_TONE)
221         ball_v[1] = ball_v[1] * -1
222     elif y > 240:
223         new_ball()
224 
225     # Check for collision with paddle
226     if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
227         # Calculate ball position relative to paddle
228         pad_ball = x + BALL_SZ - pad_pos
229         hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
230         if hit:
```

```
231             # Bounce direction based on paddle position
232             center = (PADDLE_W + BALL_SZ) / 2
233             pad_ratio = (pad_ball - center) / center  # range -1 to +1
234             angle = 90 - 60 * pad_ratio
235             hit_ball(angle)
236 
237             ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
238             beep(PADDLE_TONE)
239             collision = True
240             score = score + 1
241             update_score()
242 
243     if not collision:
244         collision = check_bricks(x, y)
```

One more collision check!

- If the ball already collided with a wall or paddle then there's no need to check the bricks.
- *Otherwise, here's where you check!*

```
245 
246     # Draw ball
247     if not collision:
248         ball_pos = (x, y)
249         draw_ball()
250 
251 
```

Hint:

- ## Type Conversion

  Are you running into an error using `i` and `j` to index the `bricks` matrix?

  The code below shows an example of what's needed. You'll need to do this for both `i` and `j`. *Note that the parentheses surround the whole calculation!*

  ```
  # Calculate i, truncating the value to an integer.
  i = int((y - BRICKS_Y_START) / ROW_H)  # row
  ```

Goals:

- Define a new function `def check_bricks(x, y)` that takes the current ball coordinates as arguments and returns a bool indicating whether it collided with a brick or not.
- Call the `check_bricks(x, y)` function inside your game loop.

Tools Found: bool, Keyword and Positional Arguments, Parameters, Arguments, and Returns, list, int, Comparison Operators, Logical Operators, Constants

Solution:

```
1  from codex import *
2  import time
3  from soundlib import *
4  import math
5  import random
6  ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)
7 
8  # Screen layout
9  TOP_WALL = 20
10 BALL_SZ = 4
11 PADDLE_W = 20
12 PADDLE_H = 8
13 PADDLE_Y = 220
14 
15 # Sounds
```

```
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587
BRICK_TONE = 740

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
BRICKS_Y_START = 30
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
```

```
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

def show_message(banner, note, color):
    clear_message()
    display.draw_text(banner, 30, 120, color, 3)
    display.draw_text(note, 30, 160, WHITE, 2)

def hit_ball(angle):
    """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
    angle = angle * math.pi / 180
    ball_v[0] = math.cos(angle) * ball_speed
    ball_v[1] = -math.sin(angle) * ball_speed

def setup_bricks():
    global bricks
    bricks = []  # Empty matrix (list of rows)
    for i in range(BRICKS_DOWN):
        bricks.append([])  # Empty row (list of columns)
        for j in range(BRICKS_ACROSS):
            bricks[i].append(True)  # Add column to this row
            brick_place(i, j, BRICK_COLORS[i])

def brick_place(i, j, color):
    """Draw a brick at the given row,column matrix location"""
    x = BRICKS_X_START + j * COL_W + BALL_SZ
    y = BRICKS_Y_START + i * ROW_H + BALL_SZ
    display.fill_rect(x, y, BRICK_W, BRICK_H, color)

def check_bricks(x, y):
    """Check for ball collision, return 'collided' True/False"""
    collided = False

    # Calculate row and column based on ball x,y
    i = int((y - BRICKS_Y_START) / ROW_H)  # row
    j = int((x - BRICKS_X_START) / COL_W)  # column

    # Is ball inside the brick grid?
    if 0 <= i < BRICKS_DOWN and 0 <= j < BRICKS_ACROSS:
        collided = bricks[i][j]  # Is there a brick here?

    if collided:
        # Destroy brick
```

```
166         bricks[i][j] = False
167         brick_place(i, j, BLACK) # Erase
168         beep(BRICK_TONE)
169 
170     return collided
171 
172 setup_bricks()
173 draw_screen_layout()
174 new_ball()
175 draw_paddle()
176 
177 ms = time.ticks_ms()
178 
179 while True:
180     dt = elapsed_ms()
181     check_buttons()
182 
183     # Update paddle
184     if pad_v:
185         pad_pos = pad_pos + pad_v * dt
186         pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
187         draw_paddle()
188 
189     # Check sound timer
190     if sound_cut > 0:
191         sound_cut = sound_cut - dt
192         if sound_cut <= 0:
193             tone.stop()
194 
195     # Check serve timer
196     if serve_timer > 0:
197         serve_timer = serve_timer - dt
198         if serve_timer <= 0:
199             serve_ball()
200         else:
201             continue
202 
203     if n_lives == 0:
204         continue
205 
206     # Update ball
207     x, y = ball_pos
208     x = x + ball_v[0] * dt
209     y = y + ball_v[1] * dt
210 
211     # Check for collision with walls
212     collision = False
213     if x <= 1 or 240 > x >= 239 - BALL_SZ:
214         collision = True
215         beep(SIDES_TONE)
216         ball_v[0] = ball_v[0] * -1
217     if y <= TOP_WALL + 1:
218         collision = True
219         beep(TOP_TONE)
220         ball_v[1] = ball_v[1] * -1
221     elif y > 240:
222         new_ball()
223 
224     # Check for collision with paddle
225     if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
226         # Calculate ball position relative to paddle
227         pad_ball = x + BALL_SZ - pad_pos
228         hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
229         if hit:
230             # Bounce direction based on paddle position
231             center = (PADDLE_W + BALL_SZ) / 2
232             pad_ratio = (pad_ball - center) / center  # range -1 to +1
233             angle = 90 - 60 * pad_ratio
234             hit_ball(angle)
235 
236             ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
237             beep(PADDLE_TONE)
238             collision = True
239             score = score + 1
240             update_score()
```

```
241
242     if not collision:
243         collision = check_bricks(x, y)
244
245     # Draw ball
246     if not collision:
247         ball_pos = (x, y)
248         draw_ball()
249
250
```

Quiz 2 - Precedence

Question 1: What is the value of `result` after the following code runs?

```
result = 4 + 2 * 3 - 1
```

✓ 9

✗ 17

✗ 12

✗ 23

Question 2: What is the result of the following code?

```
my_list = [14, 16, 18]
result = my_list[1.5]
```

✓ TypeError

✗ IndexError

✗ 16

✗ 17

✗ 18

Objective 5 - Bounce

Brick Breaking Rebound!

The ball needs to ***rebound*** when it destroys a brick.

- You could treat bricks like a "top wall", and just bounce the Y coordinate.
- Just reverse the ball's *velocity* `ball_v[1] = ball_v[1] * -1`

But sometimes the ball hits the ***side*** of a brick. Or the ***corner !*** And it would be a shame not to have *realistic* physics.

Bounce X? Bounce Y? *Both??*

Right now your `check_bricks()` code only detects `collided`. That's ANY *collision* between the ball and a brick.

- But you don't know which side of the brick was hit...
- **Yes, it matters!** Bounce off the side should go *sideways!*
- There are 4 sides to each brick. *How are you going to figure out which side the ball hit?*

The *past* is the path.

The moment when your `check_bricks()` function detects a collision, it means the ball has moved to a NEW (i, j) position in the matrix.

- If you knew the OLD (i, j) position then you would know if the ball came from below, left, right, or above the brick!

- So, remembering the *previous* (i, j) is the key to better bouncing.

Bounce Rulez	
Did the *row* change?	→ Reverse the Y velocity
Did the *column* change?	→ Reverse the X velocity

Life can only be understood backwards; but it must be lived forwards.
— Søren Kierkegaard

Check the 'Trek!

You're going to need a new 🔧global variable to remember the (i, j) position of the ball in the grid.

- And each time your `check_bricks()` function updates that position, you'll have the previous `i_prev` and `j_prev` values in case there's a collision!

Run It!

How's your bouncing?

- Your ball should be bouncing off the bricks now.
- Can you rebound from the *side* of a brick?

Make sure it's behaving the way a *real* ball would!

CodeTrek:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587
BRICK_TONE = 740

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
BRICKS_Y_START = 30
```

```
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

```

```
117 def update_score():
118     display.fill_rect(45, 0, 100, 20, BLACK)
119     display.draw_text(str(score), 45, 0, WHITE, 2)
120     display.fill_rect(195, 0, 45, 20, BLACK)
121     display.draw_text(str(n_lives), 195, 0, WHITE, 2)
122 
123 def clear_message():
124     display.fill_rect(1, 120, 238, 80, BLACK)
125 
126 def show_message(banner, note, color):
127     clear_message()
128     display.draw_text(banner, 30, 120, color, 3)
129     display.draw_text(note, 30, 160, WHITE, 2)
130 
131 def hit_ball(angle):
132     """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
133     angle = angle * math.pi / 180
134     ball_v[0] = math.cos(angle) * ball_speed
135     ball_v[1] = -math.sin(angle) * ball_speed
136 
137 def setup_bricks():
138     global bricks, ball_brick
139     ball_brick = (0, 0)  # Ball's previous (i,j) in brick matrix
```

Initialize the ball's last *brick position.*

- Naming 🔧variables can be challenging. *Sorry, `ball_brick` is all I've got!*
- Your `check_bricks()` function will update this 🔧global variable.
- Make it a 🔧tuple holding the *row* and *column* of the ball when it's in the *grid.*

```
140     bricks = []  # Empty matrix (list of rows)
141     for i in range(BRICKS_DOWN):
142         bricks.append([])  # Empty row (list of columns)
143         for j in range(BRICKS_ACROSS):
144             bricks[i].append(True)  # Add column to this row
145             brick_place(i, j, BRICK_COLORS[i])
146 
147 def brick_place(i, j, color):
148     """Draw a brick at the given row,column matrix location"""
149     x = BRICKS_X_START + j * COL_W + BALL_SZ
150     y = BRICKS_Y_START + i * ROW_H + BALL_SZ
151     display.fill_rect(x, y, BRICK_W, BRICK_H, color)
152 
153 def check_bricks(x, y):
154     """Check for ball collision, return 'collided' True/False"""
155     global ball_brick
156     collided = False
157 
158     # Calculate row and column based on ball x,y
159     i = int((y - BRICKS_Y_START) / ROW_H)  # row
160     j = int((x - BRICKS_X_START) / COL_W)  # column
161 
162     # Get ball's previous i,j position
163     i_prev, j_prev = ball_brick
164     ball_brick = (i, j)  # save for next time
```

Retrieve the previous `i` and `j` saved in the `ball_brick` 🔧tuple.
Check out the 🔧unpacking assignment from the 🔧tuple into `i_prev` and `j_prev`.

- Then save the *new* `(i, j)` ball position back in the 🔧global `ball_brick`.

Don't forget to declare your `global ball_brick` !

```
165 
166     # Is ball inside the brick grid?
167     if 0 <= i < BRICKS_DOWN and 0 <= j < BRICKS_ACROSS:
168         collided = bricks[i][j]  # Is there a brick here?
169 
170     if collided:
171         # Destroy brick
172         bricks[i][j] = False
173         brick_place(i, j, BLACK) # Erase
```

```
174             beep(BRICK_TONE)
175
176             # Bounce ball
177             if i != i_prev:  # Row changed -> bounce Y
178                 ball_v[1] = ball_v[1] * -1
179             if j != j_prev:  # Column changed -> bounce X
180                 ball_v[0] = ball_v[0] * -1
```

Add *bouncing* to your collision handler!

- If the *row* changed, **bounce Y**.
- If the *column* changed, **bounce X**.

If neither changed you wouldn't be here, would you?

```
181
182     return collided
183
184 setup_bricks()
185 draw_screen_layout()
186 new_ball()
187 draw_paddle()
188
189 ms = time.ticks_ms()
190
191 while True:
192     dt = elapsed_ms()
193     check_buttons()
194
195     # Update paddle
196     if pad_v:
197         pad_pos = pad_pos + pad_v * dt
198         pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
199         draw_paddle()
200
201     # Check sound timer
202     if sound_cut > 0:
203         sound_cut = sound_cut - dt
204         if sound_cut <= 0:
205             tone.stop()
206
207     # Check serve timer
208     if serve_timer > 0:
209         serve_timer = serve_timer - dt
210         if serve_timer <= 0:
211             serve_ball()
212         else:
213             continue
214
215     if n_lives == 0:
216         continue
217
218     # Update ball
219     x, y = ball_pos
220     x = x + ball_v[0] * dt
221     y = y + ball_v[1] * dt
222
223     # Check for collision with walls
224     collision = False
225     if x <= 1 or 240 > x >= 239 - BALL_SZ:
226         collision = True
227         beep(SIDES_TONE)
228         ball_v[0] = ball_v[0] * -1
229     if y <= TOP_WALL + 1:
230         collision = True
231         beep(TOP_TONE)
232         ball_v[1] = ball_v[1] * -1
233     elif y > 240:
234         new_ball()
235
236     # Check for collision with paddle
237     if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
238         # Calculate ball position relative to paddle
239         pad_ball = x + BALL_SZ - pad_pos
```

```
240         hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
241         if hit:
242             # Bounce direction based on paddle position
243             center = (PADDLE_W + BALL_SZ) / 2
244             pad_ratio = (pad_ball - center) / center  # range -1 to +1
245             angle = 90 - 60 * pad_ratio
246             hit_ball(angle)
247 
248             ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
249             beep(PADDLE_TONE)
250             collision = True
251             score = score + 1
252             update_score()
253 
254     if not collision:
255         collision = check_bricks(x, y)
256 
257     # Draw ball
258     if not collision:
259         ball_pos = (x, y)
260         draw_ball()
261 
262 
```

Goals:

- Initialize a new 🔧global variable in your `setup_bricks()` function, that holds the previous (i, j) position of the ball in the bricks grid. Your new global should be a 🔧tuple initialized to `(0, 0)`.
- In `check_bricks()` 🔧unpack this 🔧tuple to variables `i_prev` and `j_prev`, then update it with the new (i, j) position.
- Add bouncing to your collision handling code in `check_bricks()`

Tools Found: Locals and Globals, tuple, Assignment, Variables

Solution:

```
 1  from codex import *
 2  import time
 3  from soundlib import *
 4  import math
 5  import random
 6  ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)
 7 
 8  # Screen layout
 9  TOP_WALL = 20
10  BALL_SZ = 4
11  PADDLE_W = 20
12  PADDLE_H = 8
13  PADDLE_Y = 220
14 
15  # Sounds
16  tone = soundmaker.get_tone('trumpet')
17  sound_cut = 0 # ms until sound effect stops
18  SIDES_TONE = 392
19  TOP_TONE = 494
20  PADDLE_TONE = 587
21  BRICK_TONE = 740
22 
23  # Paddle state
24  pad_speed = 0.28  # 280px / 1000ms
25  pad_pos = 110.0  # Paddle X position
26  pad_pix = 100
27 
28  # Game state
29  START_LIVES = 3  # Lives remaining at start of game
30  score = 0
31  n_lives = START_LIVES + 1
32  serve_timer = 2000
33  ball_speed = 0.15   # 150 pixels per second
34 
```

```
# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
BRICKS_Y_START = 30
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
```

```
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

def show_message(banner, note, color):
    clear_message()
    display.draw_text(banner, 30, 120, color, 3)
    display.draw_text(note, 30, 160, WHITE, 2)

def hit_ball(angle):
    """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
    angle = angle * math.pi / 180
    ball_v[0] = math.cos(angle) * ball_speed
    ball_v[1] = -math.sin(angle) * ball_speed

def setup_bricks():
    global bricks, ball_brick
    ball_brick = (0, 0)  # Ball's previous (i,j) in brick matrix
    bricks = []  # Empty matrix (list of rows)
    for i in range(BRICKS_DOWN):
        bricks.append([])  # Empty row (list of columns)
        for j in range(BRICKS_ACROSS):
            bricks[i].append(True)  # Add column to this row
            brick_place(i, j, BRICK_COLORS[i])

def brick_place(i, j, color):
    """Draw a brick at the given row,column matrix location"""
    x = BRICKS_X_START + j * COL_W + BALL_SZ
    y = BRICKS_Y_START + i * ROW_H + BALL_SZ
    display.fill_rect(x, y, BRICK_W, BRICK_H, color)

def check_bricks(x, y):
    """Check for ball collision, return 'collided' True/False"""
    global ball_brick
    collided = False

    # Calculate row and column based on ball x,y
    i = int((y - BRICKS_Y_START) / ROW_H)  # row
    j = int((x - BRICKS_X_START) / COL_W)  # column

    # Get ball's previous i,j position
    i_prev, j_prev = ball_brick
    ball_brick = (i, j)  # save for next time

    # Is ball inside the brick grid?
    if 0 <= i < BRICKS_DOWN and 0 <= j < BRICKS_ACROSS:
        collided = bricks[i][j]  # Is there a brick here?

    if collided:
        # Destroy brick
        bricks[i][j] = False
        brick_place(i, j, BLACK) # Erase
        beep(BRICK_TONE)

        # Bounce ball
        if i != i_prev:  # Row changed -> bounce Y
            ball_v[1] = ball_v[1] * -1
        if j != j_prev:  # Column changed -> bounce X
            ball_v[0] = ball_v[0] * -1

    return collided

setup_bricks()
```

```
draw_screen_layout()
new_ball()
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            # Bounce direction based on paddle position
            center = (PADDLE_W + BALL_SZ) / 2
            pad_ratio = (pad_ball - center) / center  # range -1 to +1
            angle = 90 - 60 * pad_ratio
            hit_ball(angle)

            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True
            score = score + 1
            update_score()

    if not collision:
        collision = check_bricks(x, y)

    # Draw ball
    if not collision:
        ball_pos = (x, y)
```

```
260         draw_ball()
261
262
```

***Objective 6* - Gamify**

Gamification Time

Okay, it's time to add scoring for the bricks, and make sure everything is reset properly when the player restarts after game-over.

Review the Code

Now is a good time to review your code, and make sure you understand everything that's going on.

- Take time to reflect on each section of code.
- Remember when you added each feature, and what its purpose was.
- As you do this, add some 🔧comments !
  - Be sure to note things that might be *confusing* to someone reading this code.
  - If it confused *you,* definitely 🔧comment it!

Check the 'Trek!

This one is pretty straightforward. Go forth and *gamify!*

Run It!

Play the game a bit, and make sure the score is increasing as expected.

CodeTrek:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587
BRICK_TONE = 740

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
```

```
38  BRICK_W = 20
39  BRICK_H = 6
40  BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
41  BRICKS_Y_START = 30
42  COL_W = BRICK_W + BALL_SZ
43  ROW_H = BRICK_H + BALL_SZ
44  BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row
45  BRICK_POINTS = (7, 7, 3, 3, 5, 5, 1, 1)
```

Bricks are worth POINTS!

- This 🔧tuple will be used to retrieve the score value based on the *row* in the brick grid.
- These values are based on the original Atari game: RED is highest, and YELLOW is lowest.

```
46
47  def draw_paddle():
48      global pad_pix
49      pix = round(pad_pos)
50      if pix != pad_pix:
51          display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
52          pad_pix = pix
53          display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)
54
55  def draw_ball():
56      global ball_pix
57      pix = (round(ball_pos[0]), round(ball_pos[1]))
58      if pix != ball_pix:
59          display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
60          ball_pix = pix
61          display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)
62
63  def serve_ball():
64      global ball_pos, ball_v, ball_pix
65      # Set ball_v: serve toward paddle
66      ball_v = [0,0]
67      angle = random.randrange(-60, -120, -1)
68      hit_ball(angle)
69      ball_pos = (120.0, 120.0)
70      ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
71      clear_message()
72
73  def elapsed_ms():
74      """Returns milliseconds elapsed since last called"""
75      global ms
76      now = time.ticks_ms()
77      diff = time.ticks_diff(now, ms)
78      ms = now  # The secret word is "physics"
79      return diff
80
81  def draw_screen_layout():
82      display.draw_line(0, TOP_WALL, 0, 239, WHITE)
83      display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
84      display.draw_line(239, TOP_WALL, 239, 239, WHITE)
85      display.draw_text("SCORE", 4, 0, BLUE, 1)
86      display.draw_text("LIVES", 150, 0, BLUE, 1)
87
88  def beep(freq):
89      global sound_cut
90      tone.set_pitch(freq)
91      tone.play()
92      sound_cut = 50  # ms countdown
93
94  def check_buttons():
95      global pad_v, n_lives, score
96
97      if buttons.is_pressed(BTN_L):
98          pad_v = -pad_speed
99      elif buttons.is_pressed(BTN_B):
100         pad_v = +pad_speed
101     else:
102         pad_v = 0  # Stop
103
104     if n_lives == 0 and buttons.is_pressed(BTN_U):
105         n_lives = START_LIVES + 1
```

```
106         score = 0
107         setup_bricks()
```

A new game should rack-up a new set of bricks!

```
108
109 def new_ball():
110     global n_lives, serve_timer
111     n_lives = n_lives - 1
112     update_score()
113     if n_lives > 0:
114         serve_timer = 2000
115         show_message("Serving...", "Get Ready!", GREEN)
116     else:
117         show_message("Game Over!", "U = play again", RED)
118
119 def update_score():
120     display.fill_rect(45, 0, 100, 20, BLACK)
121     display.draw_text(str(score), 45, 0, WHITE, 2)
122     display.fill_rect(195, 0, 45, 20, BLACK)
123     display.draw_text(str(n_lives), 195, 0, WHITE, 2)
124
125 def clear_message():
126     display.fill_rect(1, 120, 238, 80, BLACK)
127
128 def show_message(banner, note, color):
129     clear_message()
130     display.draw_text(banner, 30, 120, color, 3)
131     display.draw_text(note, 30, 160, WHITE, 2)
132
133 def hit_ball(angle):
134     """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
135     angle = angle * math.pi / 180
136     ball_v[0] = math.cos(angle) * ball_speed
137     ball_v[1] = -math.sin(angle) * ball_speed
138
139 def setup_bricks():
140     global bricks, ball_brick
141     ball_brick = (0, 0)  # Ball's previous (i,j) in brick matrix
142     bricks = []  # Empty matrix (list of rows)
143     for i in range(BRICKS_DOWN):
144         bricks.append([])  # Empty row (list of columns)
145         for j in range(BRICKS_ACROSS):
146             bricks[i].append(True)  # Add column to this row
147             brick_place(i, j, BRICK_COLORS[i])
148
149 def brick_place(i, j, color):
150     """Draw a brick at the given row,column matrix location"""
151     x = BRICKS_X_START + j * COL_W + BALL_SZ
152     y = BRICKS_Y_START + i * ROW_H + BALL_SZ
153     display.fill_rect(x, y, BRICK_W, BRICK_H, color)
154
155 def check_bricks(x, y):
156     """Check for ball collision, return 'collided' True/False"""
157     global ball_brick, score
158     collided = False
159
160     # Calculate row and column based on ball x,y
161     i = int((y - BRICKS_Y_START) / ROW_H)  # row
162     j = int((x - BRICKS_X_START) / COL_W)  # column
163
164     # Get ball's previous i,j position
165     i_prev, j_prev = ball_brick
166     ball_brick = (i, j)  # save for next time
167
168     # Is ball inside the brick grid?
169     if 0 <= i < BRICKS_DOWN and 0 <= j < BRICKS_ACROSS:
170         collided = bricks[i][j]  # Is there a brick here?
171
172     if collided:
173         # Destroy brick
174         bricks[i][j] = False
175         brick_place(i, j, BLACK) # Erase
```

```
176             beep(BRICK_TONE)
177             score = score + BRICK_POINTS[i]
178             update_score()
```

Add score-keeping to your brick collision handler.

- Retrieve the score for the brick's *row* `[i]`
- BTW, is `score` a 🔧global ? *Just asking...*

```
179
180             # Bounce ball
181             if i != i_prev:  # Row changed -> bounce Y
182                 ball_v[1] = ball_v[1] * -1
183             if j != j_prev:  # Column changed -> bounce X
184                 ball_v[0] = ball_v[0] * -1
185
186         return collided
187
188     setup_bricks()
189     draw_screen_layout()
190     new_ball()
191     draw_paddle()
192
193     ms = time.ticks_ms()
194
195     while True:
196         dt = elapsed_ms()
197         check_buttons()
198
199         # Update paddle
200         if pad_v:
201             pad_pos = pad_pos + pad_v * dt
202             pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
203             draw_paddle()
204
205         # Check sound timer
206         if sound_cut > 0:
207             sound_cut = sound_cut - dt
208             if sound_cut <= 0:
209                 tone.stop()
210
211         # Check serve timer
212         if serve_timer > 0:
213             serve_timer = serve_timer - dt
214             if serve_timer <= 0:
215                 serve_ball()
216             else:
217                 continue
218
219         if n_lives == 0:
220             continue
221
222         # Update ball
223         x, y = ball_pos
224         x = x + ball_v[0] * dt
225         y = y + ball_v[1] * dt
226
227         # Check for collision with walls
228         collision = False
229         if x <= 1 or 240 > x >= 239 - BALL_SZ:
230             collision = True
231             beep(SIDES_TONE)
232             ball_v[0] = ball_v[0] * -1
233         if y <= TOP_WALL + 1:
234             collision = True
235             beep(TOP_TONE)
236             ball_v[1] = ball_v[1] * -1
237         elif y > 240:
238             new_ball()
239
240         # Check for collision with paddle
241         if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
242             # Calculate ball position relative to paddle
243             pad_ball = x + BALL_SZ - pad_pos
```

```
244         hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
245         if hit:
246             # Bounce direction based on paddle position
247             center = (PADDLE_W + BALL_SZ) / 2
248             pad_ratio = (pad_ball - center) / center  # range -1 to +1
249             angle = 90 - 60 * pad_ratio
250             hit_ball(angle)
251 
252             ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
253             beep(PADDLE_TONE)
254             collision = True
255             # Remove score increase/update: this is NOT Handball!
```

Delete some code here.

- **Breakout** doesn't hand out points unless you *break* stuff!

```
256 
257     if not collision:
258         collision = check_bricks(x, y)
259 
260     # Draw ball
261     if not collision:
262         ball_pos = (x, y)
263         draw_ball()
264 
```

Hint:

- ## Secret Word?

  Now *that's* **gamification!**

  Check the *code* in the CodeTrek.

  - The secret word is hidden somewhere in there...
  - *Be sure to review ALL the code in the CodeTrek!*

Goals:

- Create a 🔧tuple holding the official score value for each row of bricks.
  - Feel free to *personalize* these values later. But for now I want to see the original values.
- Call your `setup_bricks()` function from `check_buttons()` when a new game is started.
- Add score-keeping to your brick collision handler in `check_bricks()`.
- Remove the scoring in your *game loop*.
  - No score for hitting the paddle!
- Review the code, and add at least *two* 🔧comments.
  - One of them should contain the "secret word" (see the 💡 Hints)

Tools Found: Comments, tuple, Locals and Globals

Solution:

```
1  from codex import *
2  import time
3  from soundlib import *
4  import math
5  import random
6  ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)
7 
8  # Screen layout
```

```
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587
BRICK_TONE = 740

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
BRICKS_Y_START = 30
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row
BRICK_POINTS = (7, 7, 3, 3, 5, 5, 1, 1)

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
```

```
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0
        setup_bricks()

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

def show_message(banner, note, color):
    clear_message()
    display.draw_text(banner, 30, 120, color, 3)
    display.draw_text(note, 30, 160, WHITE, 2)

def hit_ball(angle):
    """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
    angle = angle * math.pi / 180
    ball_v[0] = math.cos(angle) * ball_speed
    ball_v[1] = -math.sin(angle) * ball_speed

def setup_bricks():
    global bricks, ball_brick
    ball_brick = (0, 0)  # Ball's previous (i,j) in brick matrix
    bricks = []  # Empty matrix (list of rows)
    for i in range(BRICKS_DOWN):
        bricks.append([])  # Empty row (list of columns)
        for j in range(BRICKS_ACROSS):
            bricks[i].append(True)  # Add column to this row
            brick_place(i, j, BRICK_COLORS[i])

def brick_place(i, j, color):
    """Draw a brick at the given row,column matrix location"""
    x = BRICKS_X_START + j * COL_W + BALL_SZ
    y = BRICKS_Y_START + i * ROW_H + BALL_SZ
    display.fill_rect(x, y, BRICK_W, BRICK_H, color)

def check_bricks(x, y):
    """Check for ball collision, return 'collided' True/False"""
    global ball_brick, score
    collided = False
```

```
159
160     # Calculate row and column based on ball x,y
161     i = int((y - BRICKS_Y_START) / ROW_H)  # row
162     j = int((x - BRICKS_X_START) / COL_W)  # column
163
164     # Get ball's previous i,j position
165     i_prev, j_prev = ball_brick
166     ball_brick = (i, j)  # save for next time
167
168     # Is ball inside the brick grid?
169     if 0 <= i < BRICKS_DOWN and 0 <= j < BRICKS_ACROSS:
170         collided = bricks[i][j]  # Is there a brick here?
171
172     if collided:
173         # Destroy brick
174         bricks[i][j] = False
175         brick_place(i, j, BLACK) # Erase
176         beep(BRICK_TONE)
177         score = score + BRICK_POINTS[i]
178         update_score()
179
180         # Bounce ball with proper physics
181         if i != i_prev:  # Row changed -> bounce Y
182             ball_v[1] = ball_v[1] * -1
183         if j != j_prev:  # Column changed -> bounce X
184             ball_v[0] = ball_v[0] * -1
185
186     return collided
187
188 setup_bricks()
189 draw_screen_layout()
190 new_ball()
191 draw_paddle()
192
193 ms = time.ticks_ms()
194
195 while True:
196     dt = elapsed_ms()
197     check_buttons()
198
199     # Update paddle
200     if pad_v:
201         pad_pos = pad_pos + pad_v * dt
202         pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
203         draw_paddle()
204
205     # Check sound timer
206     if sound_cut > 0:
207         sound_cut = sound_cut - dt
208         if sound_cut <= 0:
209             tone.stop()
210
211     # Check serve timer
212     if serve_timer > 0:
213         serve_timer = serve_timer - dt
214         if serve_timer <= 0:
215             serve_ball()
216         else:
217             continue
218
219     if n_lives == 0:
220         continue
221
222     # Update ball
223     x, y = ball_pos
224     x = x + ball_v[0] * dt
225     y = y + ball_v[1] * dt
226
227     # Check for collision with walls
228     collision = False
229     if x <= 1 or 240 > x >= 239 - BALL_SZ:
230         collision = True
231         beep(SIDES_TONE)
232         ball_v[0] = ball_v[0] * -1
233     if y <= TOP_WALL + 1:
```

```
            collision = True
            beep(TOP_TONE)
            ball_v[1] = ball_v[1] * -1
        elif y > 240:
            new_ball()

        # Check for collision with paddle
        if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
            # Calculate ball position relative to paddle
            pad_ball = x + BALL_SZ - pad_pos
            hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
            if hit:
                # Bounce direction based on paddle position
                center = (PADDLE_W + BALL_SZ) / 2
                pad_ratio = (pad_ball - center) / center  # range -1 to +1
                angle = 90 - 60 * pad_ratio
                hit_ball(angle)

                ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
                beep(PADDLE_TONE)
                collision = True
                # Remove score increase/update: this is NOT Handball!

        if not collision:
            collision = check_bricks(x, y)

        # Draw ball
        if not collision:
            ball_pos = (x, y)
            draw_ball()

```

***Objective 7* - Sound Toggle**

Remix Feature: *Add a "Mute" Button*

There are SO many features you could add to this game!

- These final couple of Objectives are just the beginning.

Silent Play

Sometimes you want to game in silence.

A cool remix would be to add a volume control! That wouldn't be too difficult...

*But for now, just a **"mute"** button will suffice.*

Check the 'Trek!

Just a few lines of code and you'll have BTN_A toggling the sound on/off.

This will give you a feeling for what it's like to add other new features to the game. Quite often a new mod will follow the same pattern:

- Add some state (*ex:* global variables)
- Connect it to game events (*ex:* button press)
- Use it to control the flow (*ex:* disable sound)

CodeTrek:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
```

```
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
mute = False
```

A 🔧global variable: Is the game *muted?*

- Set to `False` so you have sound initially.

```
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587
BRICK_TONE = 740

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
BRICKS_Y_START = 30
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row
BRICK_POINTS = (7, 7, 3, 3, 5, 5, 1, 1)

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
```

```
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    if mute:
        return
```

Mute Me!

- *No beeps for you.*

```
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score, mute

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0
        setup_bricks()

    if buttons.was_pressed(BTN_A):
        mute = not mute
        leds.set(LED_A, mute)
```

Another _button_ to button

- Use the `not` logical operator to *toggle* the mute state.
- Light `LED_A` when the sound is muted.

```

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

def show_message(banner, note, color):
    clear_message()
```

```
    display.draw_text(banner, 30, 120, color, 3)
    display.draw_text(note, 30, 160, WHITE, 2)

def hit_ball(angle):
    """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
    angle = angle * math.pi / 180
    ball_v[0] = math.cos(angle) * ball_speed
    ball_v[1] = -math.sin(angle) * ball_speed

def setup_bricks():
    global bricks, ball_brick
    ball_brick = (0, 0)  # Ball's previous (i,j) in brick matrix
    bricks = []  # Empty matrix (list of rows)
    for i in range(BRICKS_DOWN):
        bricks.append([])  # Empty row (list of columns)
        for j in range(BRICKS_ACROSS):
            bricks[i].append(True)  # Add column to this row
            brick_place(i, j, BRICK_COLORS[i])

def brick_place(i, j, color):
    """Draw a brick at the given row,column matrix location"""
    x = BRICKS_X_START + j * COL_W + BALL_SZ
    y = BRICKS_Y_START + i * ROW_H + BALL_SZ
    display.fill_rect(x, y, BRICK_W, BRICK_H, color)

def check_bricks(x, y):
    """Check for ball collision, return 'collided' True/False"""
    global ball_brick, score
    collided = False

    # Calculate row and column based on ball x,y
    i = int((y - BRICKS_Y_START) / ROW_H)  # row
    j = int((x - BRICKS_X_START) / COL_W)  # column

    # Get ball's previous i,j position
    i_prev, j_prev = ball_brick
    ball_brick = (i, j)  # save for next time

    # Is ball inside the brick grid?
    if 0 <= i < BRICKS_DOWN and 0 <= j < BRICKS_ACROSS:
        collided = bricks[i][j]  # Is there a brick here?

    if collided:
        # Destroy brick
        bricks[i][j] = False
        brick_place(i, j, BLACK) # Erase
        beep(BRICK_TONE)
        score = score + BRICK_POINTS[i]
        update_score()

        # Bounce ball
        if i != i_prev:  # Row changed -> bounce Y
            ball_v[1] = ball_v[1] * -1
        if j != j_prev:  # Column changed -> bounce X
            ball_v[0] = ball_v[0] * -1

    return collided

setup_bricks()
draw_screen_layout()
new_ball()
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

```

```
    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            # Bounce direction based on paddle position
            center = (PADDLE_W + BALL_SZ) / 2
            pad_ratio = (pad_ball - center) / center  # range -1 to +1
            angle = 90 - 60 * pad_ratio
            hit_ball(angle)

            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True
            # Remove score increase/update: this is NOT Handball!

    if not collision:
        collision = check_bricks(x, y)

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()

```

Goals:

- Add a global 🔧variable called `mute`
  - Initialize it to `False`.
- Check for `mute` inside your `beep()` function.
- In your `check_buttons()` function, toggle `mute` if `BTN_A` was pressed.
  - Also use the `mute` 🔧bool to set `LED_A` !

Tools Found: Variables, bool, Locals and Globals, Logical Operators

Solution:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
mute = False
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587
BRICK_TONE = 740

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
BRICKS_Y_START = 30
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row
BRICK_POINTS = (7, 7, 3, 3, 5, 5, 1, 1)

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
```

```
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    if mute:
        return
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score, mute

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0
        setup_bricks()

    if buttons.was_pressed(BTN_A):
        mute = not mute
        leds.set(LED_A, mute)

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

def show_message(banner, note, color):
    clear_message()
    display.draw_text(banner, 30, 120, color, 3)
    display.draw_text(note, 30, 160, WHITE, 2)

def hit_ball(angle):
    """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
    angle = angle * math.pi / 180
    ball_v[0] = math.cos(angle) * ball_speed
```

```
    ball_v[1] = -math.sin(angle) * ball_speed

def setup_bricks():
    global bricks, ball_brick
    ball_brick = (0, 0)  # Ball's previous (i,j) in brick matrix
    bricks = []  # Empty matrix (list of rows)
    for i in range(BRICKS_DOWN):
        bricks.append([])  # Empty row (list of columns)
        for j in range(BRICKS_ACROSS):
            bricks[i].append(True)  # Add column to this row
            brick_place(i, j, BRICK_COLORS[i])

def brick_place(i, j, color):
    """Draw a brick at the given row,column matrix location"""
    x = BRICKS_X_START + j * COL_W + BALL_SZ
    y = BRICKS_Y_START + i * ROW_H + BALL_SZ
    display.fill_rect(x, y, BRICK_W, BRICK_H, color)

def check_bricks(x, y):
    """Check for ball collision, return 'collided' True/False"""
    global ball_brick, score
    collided = False

    # Calculate row and column based on ball x,y
    i = int((y - BRICKS_Y_START) / ROW_H)  # row
    j = int((x - BRICKS_X_START) / COL_W)  # column

    # Get ball's previous i,j position
    i_prev, j_prev = ball_brick
    ball_brick = (i, j)  # save for next time

    # Is ball inside the brick grid?
    if 0 <= i < BRICKS_DOWN and 0 <= j < BRICKS_ACROSS:
        collided = bricks[i][j]  # Is there a brick here?

    if collided:
        # Destroy brick
        bricks[i][j] = False
        brick_place(i, j, BLACK) # Erase
        beep(BRICK_TONE)
        score = score + BRICK_POINTS[i]
        update_score()

        # Bounce ball
        if i != i_prev:  # Row changed -> bounce Y
            ball_v[1] = ball_v[1] * -1
        if j != j_prev:  # Column changed -> bounce X
            ball_v[0] = ball_v[0] * -1

    return collided

setup_bricks()
draw_screen_layout()
new_ball()
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
```

```
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            # Bounce direction based on paddle position
            center = (PADDLE_W + BALL_SZ) / 2
            pad_ratio = (pad_ball - center) / center  # range -1 to +1
            angle = 90 - 60 * pad_ratio
            hit_ball(angle)

            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True
            # Remove score increase/update: this is NOT Handball!

    if not collision:
        collision = check_bricks(x, y)

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()

```

***Objective 8* - Perpetual Play**

Breakout 4*ever!*

Maybe even **5*ever...***

One last extra feature to add, and this one is pretty essential!

It's also an example of how you can adjust the game-play. Think about that for future *remixes* you may want to code!

Clearing the Space

The goal of Breakout is to blast all the bricks, right?

- What happens when you clear them all?
- *It's kind of a let-down, eh?*

First order of business then, is to re-rack a new set of bricks after the player clears them all.

+1 for Extra Lives!

You start out with 3 lives, but it's all too easy to lose them!

- As a *bonus* for clearing a full screen of bricks, grant the player another life!
- A careful player could stock up on lives, adding a new one every new screen.

These changes make the game suitable for those *marathon gaming sessions* where you play for hours to hit the all-time high score. With a few extra lives under your belt, you can afford to take a sip of water every now and then too :-)

Check the 'Trek!

The new `check_clear()` function has a decent amount of housekeeping to do!

- Try running and *testing* your code as you add each section.
- You will learn a lot about the code itself, AND better understand why those sections are needed!

Run It!

Are you **skilled** enough to *test* your new features?

(you could always cheat and mod the code for more initial lives if ya need a boost!)

CodeTrek:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
mute = False
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587
BRICK_TONE = 740

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
BRICKS_Y_START = 30
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row
```

```
BRICK_POINTS = (7, 7, 3, 3, 5, 5, 1, 1)

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()

def elapsed_ms():
    """Returns milliseconds elapsed since last called"""
    global ms
    now = time.ticks_ms()
    diff = time.ticks_diff(now, ms)
    ms = now
    return diff

def draw_screen_layout():
    display.draw_line(0, TOP_WALL, 0, 239, WHITE)
    display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
    display.draw_line(239, TOP_WALL, 239, 239, WHITE)
    display.draw_text("SCORE", 4, 0, BLUE, 1)
    display.draw_text("LIVES", 150, 0, BLUE, 1)

def beep(freq):
    global sound_cut
    if mute:
        return
    tone.set_pitch(freq)
    tone.play()
    sound_cut = 50  # ms countdown

def check_buttons():
    global pad_v, n_lives, score, mute

    if buttons.is_pressed(BTN_L):
        pad_v = -pad_speed
    elif buttons.is_pressed(BTN_B):
        pad_v = +pad_speed
    else:
        pad_v = 0  # Stop

    if n_lives == 0 and buttons.is_pressed(BTN_U):
        n_lives = START_LIVES + 1
        score = 0
        setup_bricks()

    if buttons.was_pressed(BTN_A):
        mute = not mute
        leds.set(LED_A, mute)

def new_ball():
    global n_lives, serve_timer
    n_lives = n_lives - 1
    update_score()
    if n_lives > 0:
```

```
        serve_timer = 2000
        show_message("Serving...", "Get Ready!", GREEN)
    else:
        show_message("Game Over!", "U = play again", RED)

def update_score():
    display.fill_rect(45, 0, 100, 20, BLACK)
    display.draw_text(str(score), 45, 0, WHITE, 2)
    display.fill_rect(195, 0, 45, 20, BLACK)
    display.draw_text(str(n_lives), 195, 0, WHITE, 2)

def clear_message():
    display.fill_rect(1, 120, 238, 80, BLACK)

def show_message(banner, note, color):
    clear_message()
    display.draw_text(banner, 30, 120, color, 3)
    display.draw_text(note, 30, 160, WHITE, 2)

def hit_ball(angle):
    """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
    angle = angle * math.pi / 180
    ball_v[0] = math.cos(angle) * ball_speed
    ball_v[1] = -math.sin(angle) * ball_speed

def setup_bricks():
    global bricks, ball_brick
    ball_brick = (0, 0)  # Ball's previous (i,j) in brick matrix
    bricks = []  # Empty matrix (list of rows)
    for i in range(BRICKS_DOWN):
        bricks.append([])  # Empty row (list of columns)
        for j in range(BRICKS_ACROSS):
            bricks[i].append(True)  # Add column to this row
            brick_place(i, j, BRICK_COLORS[i])

def brick_place(i, j, color):
    """Draw a brick at the given row,column matrix location"""
    x = BRICKS_X_START + j * COL_W + BALL_SZ
    y = BRICKS_Y_START + i * ROW_H + BALL_SZ
    display.fill_rect(x, y, BRICK_W, BRICK_H, color)

def check_bricks(x, y):
    """Check for ball collision, return 'collided' True/False"""
    global ball_brick, score
    collided = False

    # Calculate row and column based on ball x,y
    i = int((y - BRICKS_Y_START) / ROW_H)  # row
    j = int((x - BRICKS_X_START) / COL_W)  # column

    # Get ball's previous i,j position
    i_prev, j_prev = ball_brick
    ball_brick = (i, j)  # save for next time

    # Is ball inside the brick grid?
    if 0 <= i < BRICKS_DOWN and 0 <= j < BRICKS_ACROSS:
        collided = bricks[i][j]  # Is there a brick here?

    if collided:
        # Destroy brick
        bricks[i][j] = False
        brick_place(i, j, BLACK) # Erase
        beep(BRICK_TONE)
        score = score + BRICK_POINTS[i]
        update_score()
        check_clear()
```

After a brick has been destroyed, it's time to check if ALL of them have been cleared.

- You'll implement the `check_clear()` function next...

```

        # Bounce ball
        if i != i_prev:  # Row changed -> bounce Y
```

```
            ball_v[1] = ball_v[1] * -1
        if j != j_prev:  # Column changed -> bounce X
            ball_v[0] = ball_v[0] * -1

    return collided

def check_clear():
    """Check if all bricks have been cleared. If so, reset and +1 lives!"""
    global n_lives, serve_timer, ball_pos
    # Search to see if any bricks remain
    for row in bricks:
        for b in row:
            if b:
                return  # Brick found - bail out!
```

Search the Matrix!

How do you find if there's a brick still standing?

- For each `row`
  - and for each brick `b` in the row
    - if it's `True` then...
      - *a brick remains!*

```

    # All clear! Get an extra life and new rack of bricks :-)
    n_lives = n_lives + 1
    update_score()
    setup_bricks()
```

If you made it here, ALL bricks are clear!

- Grant a bonus life, and call `update_score()` so it shows!
- Set up the next wave of bricks.

```

    # Erase ball (move offscreen)
    ball_pos = (-10, -10)
    draw_ball()
```

Hide the ball.

- Normally the ball goes off-screen.
- But in this case it is sitting where the last brick was!

```

    # Serve again!
    show_message("Level-Up!", "Next wave...", YELLOW)
    serve_timer = 2000
```

Finally, show an encouraging message.

- ...And get ready for the next serve!

```

setup_bricks()
draw_screen_layout()
new_ball()
draw_paddle()

ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
```

```
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            # Bounce direction based on paddle position
            center = (PADDLE_W + BALL_SZ) / 2
            pad_ratio = (pad_ball - center) / center  # range -1 to +1
            angle = 90 - 60 * pad_ratio
            hit_ball(angle)

            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True

    if not collision:
        collision = check_bricks(x, y)

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()

```

Goals:

- Define a new 🔧function `def check_clear()` that checks if any bricks remain.
    - Reset the bricks, add a bonus life, and show a "Level-Up" message too!
- Call the new `check_clear()` function from the collision handler section of `check_bricks()`.

Tools Found: Functions

Solution:

```
from codex import *
import time
from soundlib import *
import math
import random
ioexpander.io_exp_en_irq() # Init buttons (CodeX bug fix)

# Screen layout
TOP_WALL = 20
BALL_SZ = 4
PADDLE_W = 20
PADDLE_H = 8
PADDLE_Y = 220

# Sounds
tone = soundmaker.get_tone('trumpet')
sound_cut = 0 # ms until sound effect stops
mute = False
SIDES_TONE = 392
TOP_TONE = 494
PADDLE_TONE = 587
BRICK_TONE = 740

# Paddle state
pad_speed = 0.28  # 280px / 1000ms
pad_pos = 110.0  # Paddle X position
pad_pix = 100

# Game state
START_LIVES = 3  # Lives remaining at start of game
score = 0
n_lives = START_LIVES + 1
serve_timer = 2000
ball_speed = 0.15   # 150 pixels per second

# Bricks
BRICKS_ACROSS = 10
BRICKS_DOWN = 8
BRICK_W = 20
BRICK_H = 6
BRICKS_X_START = -2   # +BALL_SZ to reach edge of first brick
BRICKS_Y_START = 30
COL_W = BRICK_W + BALL_SZ
ROW_H = BRICK_H + BALL_SZ
BRICK_COLORS = (RED, RED, ORANGE, ORANGE, GREEN, GREEN, YELLOW, YELLOW) # per row
BRICK_POINTS = (7, 7, 3, 3, 5, 5, 1, 1)

def draw_paddle():
    global pad_pix
    pix = round(pad_pos)
    if pix != pad_pix:
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLACK)
        pad_pix = pix
        display.fill_rect(pad_pix, PADDLE_Y, PADDLE_W, PADDLE_H, BLUE)

def draw_ball():
    global ball_pix
    pix = (round(ball_pos[0]), round(ball_pos[1]))
    if pix != ball_pix:
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, BLACK)
        ball_pix = pix
        display.fill_rect(ball_pix[0], ball_pix[1], BALL_SZ, BALL_SZ, WHITE)

def serve_ball():
    global ball_pos, ball_v, ball_pix
    # Set ball_v: serve toward paddle
    ball_v = [0,0]
    angle = random.randrange(-60, -120, -1)
    hit_ball(angle)
    ball_pos = (120.0, 120.0)
    ball_pix = (round(ball_pos[0]), round(ball_pos[1]))
    clear_message()
```

```
73
74  def elapsed_ms():
75      """Returns milliseconds elapsed since last called"""
76      global ms
77      now = time.ticks_ms()
78      diff = time.ticks_diff(now, ms)
79      ms = now
80      return diff
81
82  def draw_screen_layout():
83      display.draw_line(0, TOP_WALL, 0, 239, WHITE)
84      display.draw_line(0, TOP_WALL, 239, TOP_WALL, WHITE)
85      display.draw_line(239, TOP_WALL, 239, 239, WHITE)
86      display.draw_text("SCORE", 4, 0, BLUE, 1)
87      display.draw_text("LIVES", 150, 0, BLUE, 1)
88
89  def beep(freq):
90      global sound_cut
91      if mute:
92          return
93      tone.set_pitch(freq)
94      tone.play()
95      sound_cut = 50  # ms countdown
96
97  def check_buttons():
98      global pad_v, n_lives, score, mute
99
100     if buttons.is_pressed(BTN_L):
101         pad_v = -pad_speed
102     elif buttons.is_pressed(BTN_B):
103         pad_v = +pad_speed
104     else:
105         pad_v = 0  # Stop
106
107     if n_lives == 0 and buttons.is_pressed(BTN_U):
108         n_lives = START_LIVES + 1
109         score = 0
110         setup_bricks()
111
112     if buttons.was_pressed(BTN_A):
113         mute = not mute
114         leds.set(LED_A, mute)
115
116 def new_ball():
117     global n_lives, serve_timer
118     n_lives = n_lives - 1
119     update_score()
120     if n_lives > 0:
121         serve_timer = 2000
122         show_message("Serving...", "Get Ready!", GREEN)
123     else:
124         show_message("Game Over!", "U = play again", RED)
125
126 def update_score():
127     display.fill_rect(45, 0, 100, 20, BLACK)
128     display.draw_text(str(score), 45, 0, WHITE, 2)
129     display.fill_rect(195, 0, 45, 20, BLACK)
130     display.draw_text(str(n_lives), 195, 0, WHITE, 2)
131
132 def clear_message():
133     display.fill_rect(1, 120, 238, 80, BLACK)
134
135 def show_message(banner, note, color):
136     clear_message()
137     display.draw_text(banner, 30, 120, color, 3)
138     display.draw_text(note, 30, 160, WHITE, 2)
139
140 def hit_ball(angle):
141     """Set new velocity: angle 0-180 goes up, 180-360 goes down"""
142     angle = angle * math.pi / 180
143     ball_v[0] = math.cos(angle) * ball_speed
144     ball_v[1] = -math.sin(angle) * ball_speed
145
146 def setup_bricks():
147     global bricks, ball_brick
```

```
    ball_brick = (0, 0)  # Ball's previous (i,j) in brick matrix
    bricks = []  # Empty matrix (list of rows)
    for i in range(BRICKS_DOWN):
        bricks.append([])  # Empty row (list of columns)
        for j in range(BRICKS_ACROSS):
            bricks[i].append(True)  # Add column to this row
            brick_place(i, j, BRICK_COLORS[i])

def brick_place(i, j, color):
    """Draw a brick at the given row,column matrix location"""
    x = BRICKS_X_START + j * COL_W + BALL_SZ
    y = BRICKS_Y_START + i * ROW_H + BALL_SZ
    display.fill_rect(x, y, BRICK_W, BRICK_H, color)

def check_bricks(x, y):
    """Check for ball collision, return 'collided' True/False"""
    global ball_brick, score
    collided = False

    # Calculate row and column based on ball x,y
    i = int((y - BRICKS_Y_START) / ROW_H)  # row
    j = int((x - BRICKS_X_START) / COL_W)  # column

    # Get ball's previous i,j position
    i_prev, j_prev = ball_brick
    ball_brick = (i, j)  # save for next time

    # Is ball inside the brick grid?
    if 0 <= i < BRICKS_DOWN and 0 <= j < BRICKS_ACROSS:
        collided = bricks[i][j]  # Is there a brick here?

    if collided:
        # Destroy brick
        bricks[i][j] = False
        brick_place(i, j, BLACK) # Erase
        beep(BRICK_TONE)
        score = score + BRICK_POINTS[i]
        update_score()
        check_clear()

        # Bounce ball
        if i != i_prev:  # Row changed -> bounce Y
            ball_v[1] = ball_v[1] * -1
        if j != j_prev:  # Column changed -> bounce X
            ball_v[0] = ball_v[0] * -1

    return collided

def check_clear():
    """Check if all bricks have been cleared. If so, reset and +1 lives!"""
    global n_lives, serve_timer, ball_pos
    # Search to see if any bricks remain
    for row in bricks:
        for b in row:
            if b:
                return  # Brick found - bail out!

    # All clear! Get an extra life and new rack of bricks :-)
    n_lives = n_lives + 1
    update_score()
    setup_bricks()

    # Erase ball (move offscreen)
    ball_pos = (-10, -10)
    draw_ball()

    # Serve again!
    show_message("Level-Up!", "Next wave...", YELLOW)
    serve_timer = 2000

setup_bricks()
draw_screen_layout()
new_ball()
draw_paddle()

```

```
ms = time.ticks_ms()

while True:
    dt = elapsed_ms()
    check_buttons()

    # Update paddle
    if pad_v:
        pad_pos = pad_pos + pad_v * dt
        pad_pos = min(max(pad_pos, 1), 238 - PADDLE_W)
        draw_paddle()

    # Check sound timer
    if sound_cut > 0:
        sound_cut = sound_cut - dt
        if sound_cut <= 0:
            tone.stop()

    # Check serve timer
    if serve_timer > 0:
        serve_timer = serve_timer - dt
        if serve_timer <= 0:
            serve_ball()
        else:
            continue

    if n_lives == 0:
        continue

    # Update ball
    x, y = ball_pos
    x = x + ball_v[0] * dt
    y = y + ball_v[1] * dt

    # Check for collision with walls
    collision = False
    if x <= 1 or 240 > x >= 239 - BALL_SZ:
        collision = True
        beep(SIDES_TONE)
        ball_v[0] = ball_v[0] * -1
    if y <= TOP_WALL + 1:
        collision = True
        beep(TOP_TONE)
        ball_v[1] = ball_v[1] * -1
    elif y > 240:
        new_ball()

    # Check for collision with paddle
    if not collision and (PADDLE_Y + PADDLE_H) > y >= (PADDLE_Y - BALL_SZ):
        # Calculate ball position relative to paddle
        pad_ball = x + BALL_SZ - pad_pos
        hit = 0 <= pad_ball <= (PADDLE_W + BALL_SZ)
        if hit:
            # Bounce direction based on paddle position
            center = (PADDLE_W + BALL_SZ) / 2
            pad_ratio = (pad_ball - center) / center  # range -1 to +1
            angle = 90 - 60 * pad_ratio
            hit_ball(angle)

            ball_pos = (x, PADDLE_Y - BALL_SZ - 1) # ensure above paddle (avoid double-hits)
            beep(PADDLE_TONE)
            collision = True

    if not collision:
        collision = check_bricks(x, y)

    # Draw ball
    if not collision:
        ball_pos = (x, y)
        draw_ball()

```

Mission 16 Complete

You Made It!

The Woz would be proud of you!

This was NOT an easy Mission. But your game is super-impressive and quite playable.

- Take a few minutes to play *Breakout* and enjoy the fruits of your labor.