Fly with Python

Learn Python fundamentals as you explore the world of autonomous drones with CodeAIR

Mission 1 - Welcome

Welcome to the CodeSpace Development Environment!

A virtual world for exploring robotics with code.

We're glad you're here!

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

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

Ready to begin your first *Mission?*

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

Objective 1 - Mission Objectives

Objectives

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

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

The *goals* to complete the Objective are below:

Goal:

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

Solution:

N/A

Objective 2 - Text Editor

Text Editor

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

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

Go ahead and *type something in!*

Goal:

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

Solution:

N/A

***Objective 3* - Tool Box**

Your Coding Toolbox

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

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

Collect 'em *ALL!*

When you see a tool, CLICK on it!

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

Access Your Tools

You can always open up your toolbox later for reference.

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

Goal:

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

Tools Found: Debugging

Solution:

N/A

***Objective 4* - Simulation Controls**

Simulation Controls

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

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

Click on the **Camera** menu below.

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

Want to *hide* these instructions?

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

Goals:

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

Solution:

N/A

Quiz 1 **- Your First Mission Quiz**

Question 1: Are you ready to learn some Python coding with your CodeAIR?

✓ Yes. This is simple!

✗ I don't think I can.

✗ It looks too complicated.

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

✓ How to open an objective

✓ How to move the camera

✗ How to run a half marathon

✗ How to control the weather

Mission 1 Complete

Welcome to CodeSpace!

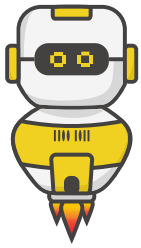
You've completed your first ***Mission.***

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

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

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

Mission 2 **- Introducing CodeAIR**



Welcome aboard!

You're about to embark on a thrilling adventure. I'll be your co-pilot as we navigate the skies of *Python* with your **CodeAIR** drone.

Why learn coding with drones?

- Flying is awesome, and the drones of the future will mostly fly themselves *autonomously!*
- Drones are used in industries like film, agriculture, exploration, and **science**.
- The ability to program a drone opens up a whole world of *automation* and **AI**.

But remember… Drones, like all tech, need coding by humans like YOU.

As you complete this *hands-on* course, you'll be mastering skills that can control not only drones but *ANY* tech you can imagine!

Objective 1 **- Behold the CodeAIR**

Firia Labs CodeAIR - *Fly with Python!*

CodeAIR is a high performance micro-drone that's fully programmable in Python.

- That means Python flies ***on the drone!*** (not on your laptop...)
- With tons of sensors, 🔧LEDs, a 🔧speaker, and 🔧buttons you can program.
- And of course, 🔧motors and props for flying!

Computer Vision - *AI Onboard*

- CodeAIR has a *camera* - but it's not meant to be an FPV (First Person View) drone.
- ...unless FPV means *"Flying Python Vision"* that is!
- Yep, your onboard **Python code** will use that camera (plus AI) to see stuff from the air!

Start with CODE!

In the "Fly with Python" curriculum, you'll learn about *quadcopter technology* and the fundamentals of software and control systems that make drone flight possible. This is *professional career* stuff, not just flying some RC planes around :-) You'll be learning principles from a variety of fields:

- Computer Science
- Electrical Engineering
- Aerospace Engineering

Goal:

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

Tools Found: BYTE LEDs, Speaker, Buttons, Motors and Props

Solution:

N/A

Objective 2 **- Static Electricity**

Careful with your CodeAIR!

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 ***CodeAIR*** by its **prop guard**, being gentle with the Connectors, LEDs, and other electronic components.
    - They're all exposed on the board so you can *really* get to know them!
2. Keep your CodeAIR in its box when not in use.
3. It's good practice to touch some grounded metal (desk, doorknob) before handling the CodeAIR 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 ON/OFF Switch!

Solution:

N/A

Quiz 1 **- Static Response**

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

✅ Touch some grounded metal

❌ Jumping jacks

❌ Clean it with wet wipes

Objective 3 **- Find the CPU**

Where does the code run?

The code you write will run on the ***CodeAIR itself!***

- After you load it on there, it doesn't need your computer anymore.
- This is no radio-controlled toy! It's fully *autonomous!*

The Main 🔧CPU (Central Processing Unit) shown here is the brain of the CodeAIR, where your Python code runs.

CodeAIR'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** CodeAIR's onboard 🔧peripherals.

- Sensors, buttons, LEDs.
- And a second dedicated *flight control* CPU, running in parallel!

The 🔧CPU is an amazing little device!

Can you find the CPU?

Goal:

- Click on the main Central Processing Unit (CPU) in the 3D Scene.

Tools Found: CPU and Peripherals

Solution:

N/A

Objective 4 **- Power Switch**

Power Switch

CodeAIR has a *slide switch* with two positions: 1=ON, and 0=OFF.

You need to set the switch to the ON (1) position for the CodeAIR to fully power-up!

- CodeAIR will still charge its battery, even with the power switch in the OFF (0) position, if you have the 🔧USB connected. *More about that later!*

Battery

CodeAIR's *battery* is a Lithium Polymer single cell (1S) pack.

- Notice how it is inserted inside the landing sled.
- You can flex the *retaining tabs* slightly downward to allow the battery to slide out.
- The white *battery connectors* are *polarized,* meaning they only fit one way. Take a close look at how the slot lines up so they fit together neatly!

Physical Interaction: *Turn it ON*

Grab your CodeAIR and set its switch to ON (1)

Goals:

- Click on the Power Switch in the 3D Scene.
- Click the white *battery connector* in the 3D scene.

Tools Found: USB

Solution:

N/A

***Objective 5* - Connect the USB**

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

⚠ Note ⚠

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

- 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 ***CodeAIR***.
2. It provides 5 volt DC power to the CodeAIR.

USB can power everything but the motors, even without a battery!

- And it *charges* the battery while plugged-in.

Physical Interaction: *Plug In*

Connect the USB cable between your computer and the CodeAIR.

Goal:

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

Tools Found: USB

Solution:

N/A

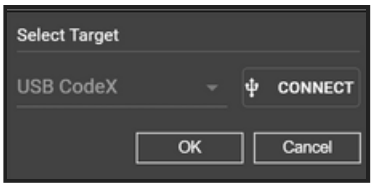
***Objective 6* - Link to CodeSpace**

Link CodeAIR 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 CodeAIR.
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 CodeAIR it will request permission to connect.

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

Physical Interaction: *Troubleshooting Connections*

If you are having trouble getting CodeSpace to recognize your CodeAIR:

1. Make sure CodeAIR's power switch is ON (set to "1").
2. Check that your USB cable is *fully* plugged in to CodeAIR.
3. Try connecting with CodeAIR's *battery unplugged.*
4. Disconnect USB, reload your browser window, then reconnect USB.

Find more troubleshooting tips at https://docs.firialabs.com/codeair/hardware_reference/Troubleshooting.html

Goal:

- Link your CodeAIR to CodeSpace.
  - **Hint:** Make sure only one CodeAIR is connected.

Solution:

N/A

Objective 7 **- 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 the 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: `Lights1`
  - If this file is already in your file system go ahead and use the *New File...* button anyway!

Solution:

N/A

Objective 8 **- 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 ("To-Do items") for you to fill in on your own!

Comments

When you write code, sometimes you'll want to add some notes - maybe to remind yourself of why you did something, or to explain things to someone else who might read your code.

- In programming languages these notes are called comments. The computer ignores them!
- ***You don't have to type the comments from the CodeTrek!***
- But you may want to add some *comments* of your own, taking notes as you learn :-)

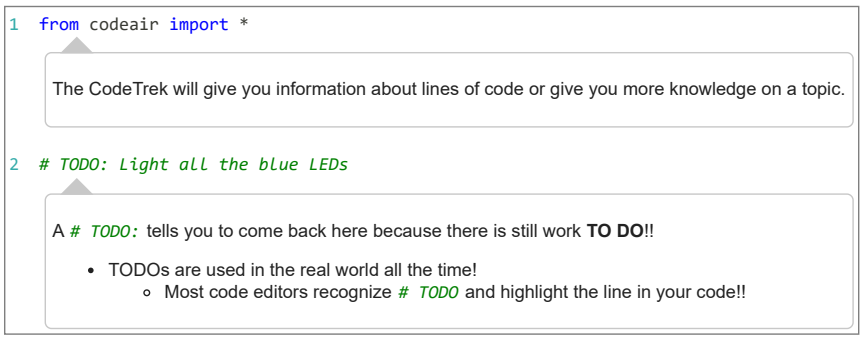
To-Do Items

A `# TODO:` in the code is a standard reminder comment.

- It tells you to come back here because there is still work **TO DO**!!
- Professional programmers often use `# TODO` comments, to mark "unfinished business".
- When you see `# TODO` in the CodeTrek, that's where YOU need to write the *actual 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.

Tools Found: Comments

Solution:

N/A

Quiz 2 - Questions TODO

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

✓ Execute your code

✗ Figure out what you were thinking

✗ Provide +5 Volt power

Question 2: Which of the following comments is a standard reminder for you to fill-in *actual code* here?

✓ `# TODO: fix this`

✗ `# good code below`

✗ `# x should be a float`

Objective 9 - Light's On

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

⚠ Notice ⚠

▶ Run It!

CodeTrek:

```
1  from codeair import *
2  leds.set(0, 50)
```

Set LED number `0` to 50%

Hint:

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

  Don't worry, you will get plenty of practice with this - and more complete explanations are in store. But to start out, it's good to just get some code running!

Goals:

- Open the CodeTrek 🚶 to see the code.
- RUN ▶ your code to light a BLUE LED on your CodeAIR.
  - Make sure your code matches the **CodeTrek**!

Tools Found: Punctuation, Syntax Highlighting, BYTE LEDs, Motors and Props

Solution:

```
1  from codeair import *
2  leds.set(0, 50)
```

Objective 10 - More Lights

Light it UP!

Okay, you're really getting hands-on now.

- It's time to roll up your sleeves and test your knowledge.

Did you pick up the *tool* for the 🔧BYTE LEDs ?

If not, go get it!

Review the two lines of Python code you wrote:

```
from codeair import *
leds.set(0, 50)
```

Even without reading the ToolBox documentation, you might guess that this sets an LED to some value.

- You could read it "Set LED zero to 50%".
- So the first number `0` is *which* LED, and the second number is 0-100% brightness!

Can you light up ALL those blue LEDs?

- There are a few ways to accomplish this!
- The most straigtforward way is just to copy/paste that `leds.set(n, 50)` seven more times, replacing `n` with the numbers `1` through `7`.

Check the 'Trek!

Your turn - get LIT!

- Please follow the "straightforward" method from the CodeTrek first, so I can check your work.
- After that feel free to try more advanced approaches if you're so inclined!

CodeTrek:

```
1  from codeair import *
2  leds.set(0, 50)
3  leds.set(1, 50)
```

Just copy the same line, changing the `n` to light LEDs 0-7.

- You can set them all to 50% as shown, or get creative!

```
4  # TODO: continue to light next six LEDs...
```

Goal:

- RUN ▶ your code to light all the blue LEDs.
  - Always check the **CodeTrek**!

Tools Found: BYTE LEDs

Solution:

```
 1  from codeair import *
 2  leds.set(0, 50)
 3  leds.set(1, 50)
 4  leds.set(2, 50)
 5  leds.set(3, 50)
 6  leds.set(4, 50)
 7  leds.set(5, 50)
 8  leds.set(6, 50)
 9  leds.set(7, 50)
10
```

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 **- Pre-Flight Check**

Welcome to Ground School

This pre-flight mission will give you a "crash course" in some coding skills you will be using to get your CodeAIR flying.

- But before you take to the air, you have to learn how to control this machine on the ground!

Pre-Flight Checks

A trained pilot will go through a detailed checklist before every flight.

- Gotta make sure all systems are in working order.
- That includes lighting systems, safety devices, control surfaces, engines, and navigation sensors.

You should make a habit to visually inspect CodeAIR before every flight!

Mission Targets

You have a few Objectives in this Mission!

- Continue exploring CodeAIR's lighting system - the blue 🔧LEDs and more!
- Learn about the 🔧speaker and add sounds to your drone's repertoire.
- On the aeronautics front, you will program the onboard lighting system to show the colors of the international *Aircraft Position Lighting* scheme.

Objective 1 **- Lighting Beacon**

Blink Those LEDs!

Sure, your code can light up the blue 🔧LEDs, but can you BLINK them?

- Blinking an 🔧LED is the "Hello, World" of *embedded systems programming.*
- Oh, did you know that's what you're doing? Yeah, that's writing code that goes in a tiny *microcontroller* embedded in some product that nobody realizes there's actually software running inside :-)
  - Like... AirPods, or stage lights, or a stopwatch.

Slowing it Down

By now, hopefully you've been lighting up multiple blue LEDs.

- Your program lights the LEDs and ends pretty quickly!
- And even though *your Python code is executed one line at a time,* all the LEDs seem to light up at once.

But what if you want to blink an LED on and off a few times?

Create a New File!

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

Run It!

Type the following code into the text editor and RUN it!

This *should* blink LED 0 *twice*

- Remember, you don't have to type the `#` 🔧comments.

```
from codeair import *

leds.set(0, 50)  # ON
leds.set(0, 0)   # OFF
leds.set(0, 50)  # ON
leds.set(0, 0)   # OFF
```

⚠ Note: This is not gonna blink as expected! ⚠

What's Up?

Test the code above! It doesn't blink properly because you need to *slow the computer down...*

Concept: *sleep*

When you want to control the pace of actions in your code, you have to specifically state where and how much delay you want!

- Check out the timing tool to learn how you can use Python's `time` module and the `sleep()` function to slow down the action!

Blink!

All you need to add to the test code above is a few delays.

- Pause a bit after the light turns ON...
- And don't forget to wait a moment after the light is OFF also!

Check the 'Trek!

As usual, open up the CodeTrek to guide your coding on this Objective!

CodeTrek:

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

Don't forget to import the `sleep()` function.

```
3
4  leds.set(0, 50)
5  sleep(1)
6  leds.set(0, 0)
7  sleep(1)
```

On for 1 second... Off for 1 second.

- Four *groovy* lines of Python code!

```
8
9  # TODO: Blink 3 more times!
10 # (Don't type this comment! You write the code here.)
```

Hey, a *#TODO* comment!

- I told ya there would be some of these!

Remember?

You don't need to type the comments. This is where you figure out what code needs to go here to get the job done :-)

Copy and Paste

You might want to use the editor shortcuts to reduce the typing on this one.

Goal:

- Blink a blue LED at least four times, the *hard way!*
    - (That means *no loops* if you're an advanced student :-)

Tools Found: BYTE LEDs, LED, Comments, Timing, import, Editor Shortcuts

Solution:

```
from codeair import *
from time import sleep

leds.set(0, 50)
sleep(1)
leds.set(0, 0)
sleep(1)
leds.set(0, 50)
sleep(1)
leds.set(0, 0)
sleep(1)
leds.set(0, 50)
sleep(1)
leds.set(0, 0)
sleep(1)
leds.set(0, 50)
sleep(1)
leds.set(0, 0)
sleep(1)

```

Objective 2 **- Loop de Loop**

Keep CodeAIR Flashing!

You could blink a few more times by just copying the same lines over and over.

- But it would be much better AND less typing to use an **infinite** loop!

*Yes, you need to move your **LED flashing** code inside a loop!*

Concept: *while loop*

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`!

So in Python your *infinite loop* will look something like:

```
while True:
    # LED on
    # pause
    # LED off
    # pause
```

Note two important things here:

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

Why `while True:` ?

Check out the 🔧loop tool to learn more about the `while condition:` statement.

- You'll learn to use other 🔧conditions to control how many times the loop repeats.
- But to repeat forever, just use the value `True`.

Check the 'Trek!

Modify your code to put the blinking inside a loop. Check out the 🔧editor shortcuts to learn how to easily 🔧indent a whole block of code to place it "inside" your new 🔧loop.

CodeTrek:

```
1  # TODO: Import everything from the codeair library.
2  # TODO: Import sleep from the time library.
```

You should already have the 🔧import statements from your previous code. Just make sure they're still there.

```
from codeair import *  # For `leds`
from time import sleep # For `sleep()`
```

```
3
4  while True:
5      leds.set(0, 50)
6      sleep(0.1)
7      # TODO: LED off
8      sleep(0.2)
```

All you need is four lines of code inside the `while` 🔧loop.

- They are 🔧indented - *that means they're inside the loop!*

Always make sure your indented blocks line up neatly!

```
9
```

Goal:

- Add a `while` loop to get your blue 🔧LEDs blinking continuously.

Tools Found: Loops, Indentation, bool, Editor Shortcuts, BYTE LEDs, import

Solution:

```
1  from codeair import *
2  from time import sleep
3
4  while True:
5      leds.set(0, 50)
6      sleep(0.1)
7      leds.set(0, 0)
8      sleep(0.2)
```

Quiz 1 - Blinking LEDs

Question 1: How is your Python code able to call the `sleep()` function?

✓ `from time import sleep`

- ✗ `from codeair import *`
- ✗ It is a 🔧built-in function, so it is always available
- ✗ `from sleep import delay`

Question 2: What does `sleep(1)` do?

- ✓ Pauses code execution for 1 second
- ✗ Pauses code execution for 1 microsecond
- ✗ Stops the program
- ✗ Disables all peripherals

Question 3: What line of code will turn off the last blue LED?

- ✓ `leds.set(7, 0)`
- ✗ `leds.set(8, 0)`
- ✗ `leds.set(8, OFF)`
- ✗ `leds.off(7)`

***Objective 3* - Light Cycle**

Light Cycle

Check the 'Trek!

Run It!

Observe the LEDs when you run this code.

- Can you see them continuously *cycling* every time your loop repeats?

⚠️ Uh-oh, mine's not working right either! ⚠️

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

- If you want to try fixing it now, that's cool too! You can compare your solution on the next Objective :-)

CodeTrek:

```
from codeair import *
from time import sleep

while True:
    leds.set(0, 50)
    sleep(0.1)

    leds.set(1, 50)
    sleep(0.1)

    leds.set(2, 50)
    sleep(0.1)

    # TODO: Blink LEDs 3-7
```

This Objective is to "Swoosh" all eight LEDs.

- I've shown you the first three above... *You fill-in the rest!*

Hint:

- ## Note to Advanced Students

  Once again, you need to write this code the long way.

  - The goal here is to gradually introduce topics, and show why more advanced techniques are needed!
  - The CodeTrek shows the form I'm looking for to pass this Objective.

Goal:

- Light up all eight 🔧LEDs in sequence.
  - *And repeat, inside a `while` 🔧loop!*

Tools Found: Loops, BYTE LEDs

Solution:

```
from codeair import *
from time import sleep

while True:
    leds.set(0, 50)
    sleep(0.1)

    leds.set(1, 50)
    sleep(0.1)

    leds.set(2, 50)
    sleep(0.1)

    leds.set(3, 50)
    sleep(0.1)

    leds.set(4, 50)
    sleep(0.1)

    leds.set(5, 50)
    sleep(0.1)

    leds.set(6, 50)
    sleep(0.1)

    leds.set(7, 50)
    sleep(0.1)


```

***Objective 4* - Fancy LED Fix**

🔧Debugging

Do you see what's happening with your LED animation?

- The code I started you with in the CodeTrek never turned the LEDs *OFF!*
- Right, that bug was *my* fault! *Next one's on you ;-)*

Covering Your Tracks

Okay, so a little more code needs to be added to your program.

- After you leave an LED on for a bit, you need to turn it off before lighting up the next one.
- ...or you could wait till *after* you light up the next one. Your choice!

Check the 'Trek!

Fix up your code, and let's see those lights cycling beautifully.

CodeTrek:

```
from codeair import *
from time import sleep

while True:
    leds.set(0, 50)
    sleep(0.1)
    leds.set(0, 0)
```

Set the LED brightness to `0` to turn it off.

```


    leds.set(1, 50)
    sleep(0.1)
    # TODO
```

Same deal - each LED needs to be turned off, just like LED `0` above.

```

    leds.set(2, 50)
    sleep(0.1)
    # TODO

    leds.set(3, 50)
    sleep(0.1)
    # TODO

    leds.set(4, 50)
    sleep(0.1)
    # TODO

    leds.set(5, 50)
    sleep(0.1)
    # TODO

    leds.set(6, 50)
    sleep(0.1)
    # TODO

    leds.set(7, 50)
    sleep(0.1)
    # TODO


```

Goal:

- Modify your program to turn the LEDs off as well!
  - You should see the LEDs moving continuously as your program loops.

Tools Found: Debugging, BYTE LEDs

Solution:

```
from codeair import *
from time import sleep
```

```
3
4  while True:
5      leds.set(0, 50)
6      sleep(0.1)
7      leds.set(0, 0)
8
9      leds.set(1, 50)
10     sleep(0.1)
11     leds.set(1, 0)
12
13     leds.set(2, 50)
14     sleep(0.1)
15     leds.set(2, 0)
16
17     leds.set(3, 50)
18     sleep(0.1)
19     leds.set(3, 0)
20
21     leds.set(4, 50)
22     sleep(0.1)
23     leds.set(4, 0)
24
25     leds.set(5, 50)
26     sleep(0.1)
27     leds.set(5, 0)
28
29     leds.set(6, 50)
30     sleep(0.1)
31     leds.set(6, 0)
32
33     leds.set(7, 50)
34     sleep(0.1)
35     leds.set(7, 0)
36
37
```

Quiz 2 **- Infinite Loops**

Question 1: How many times will the loop blink LED 0?

```
while True:
    leds.set(0, 50)
    sleep(1)
leds.set(0, 0)
sleep(1)
```

✓ LED 0 will stay on and not blink

✗ LED 0 will stay off and not blink

✗ Continuously - infinite loop

✗ 0 times - loop will not execute

Question 2: How many times will the loop blink LED 0?

```
while False:
    leds.set(0, 50)
    sleep(1)
    leds.set(0, 0)
    sleep(1)
```

✓ 0 times - loop will not execute

✗ Continuously - infinite loop

✗ 1 time

❌ LED 0 will stay on and not blink

***Objective 5* - Wild Blue Yonder**

Multimedia

You've seen some *lighting* features of CodeAIR, but what about ***sound***?

- There's a lot of capability in the 🔧speaker to explore!

Does a drone *really* need sound?

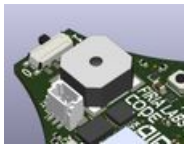
When airborne, you'll find the motors provide quite a lot of sound on their own!

- So much in fact, that the speaker has a hard time competing.
- **But** there are quite a few situations where you'll want CodeAIR to alert you before, after, or even during flight. And sound is an excellent way to do that!

Concept: *speaker*

The 🔧speaker allows you to provide a *frequency* in Hertz and a *duration* in milliseconds.

```
speaker.beep(440, 200)  # Play 440Hz tone for 200ms
```



Those frequencies can be *notes* of a musical melody, crazy sound effects, whatever you want!

Musical Melody

For your first 🔧speaker assignment you are to play a melody!

- Musical notes represent different sound *frequencies*.
- For this melody you will use the following notes. *I'm giving you these as Python 🔧constants that you can copy and paste into your code. (click the icon on the right!)*

```
# Notes used in the melody
D5 = 587
E5 = 659
F5 = 698
FS5 = 740
G5 = 784
```

And Lights, Too!

Wouldn't it be awesome to play a melody along with the lights you already have?

- When you light an LED, play a note.
- You may need to adjust the timing a little, but it sounds simple so far!

A Fitting Tune

Here's the tune you'll be playing, in Scientific Pitch Notation

E5 - G5 - G5 - F5 - E5 - D5 - E5 - F5 - F#5 - G5

- Every note should be played for a given duration (milliseconds) and after it stops take a musical "rest" using `sleep(sec)`.
- What is this melody? Try coding it first, and maybe you'll recognize it! *Spoiler Alert:* See the 💡 Hints to learn about the melody.

Sequence the notes and lights per the table below from left to right:

Melody →										
LED	0	1	1	2	3	4	5	6	7	STA
Note	E5	G5	G5	F5	E5	D5	E5	F5	FS5	G5
Duration (ms)	100	100	700	100	100	100	400	400	400	500

Melody →										
Rest (sec)	0.2	0.1	0.3	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Save to a New File!

Use the **File** → **Save As** menu to create a new file called ***Melody***.

Same Starting Code, No Loop

If you *Save-As* the code from the last Objective, you just need to use the SHIFT-TAB editor shortcut to *un-indent* the code so it's no longer inside your `while` loop. *(And delete the `while` statement of course)*

Bonus LED

The melody finishes by lighting the *STA* LED, positioned near the USB connector.

Check the 'Trek!

The CodeTrek gives you the first three notes and lights. You'll take it from there and complete the melody!

Run It!

Now sit back, watch AND listen to the show!

CodeTrek:

```
1  from codeair import *
2  from time import sleep
3
4  # Notes used in the melody
5  D5 = 587
6  E5 = 659
7  F5 = 698
8  FS5 = 740
9  G5 = 784
```

Defining the notes. Paste the constants you copied from the Objective overview. *(you did copy them, right?)*

- These are the sound frequencies of each note in Hertz (cycles per second).

```
10
11 leds.set(0, 50)        # LED 0
12 speaker.beep(E5, 100)  # Note E5, duration 100
13 sleep(0.2)             # Rest 0.2
14 leds.set(0, 0)
```

A single note.

- Each note will look like this, pretty much just adding a `speaker.beep()` to the code you already have that sequences LEDs.

```
15
16 leds.set(1, 50)        # LED 1
17 speaker.beep(G5, 100)
18 sleep(0.1)
19 speaker.beep(G5, 700)
20 sleep(0.3)
21 leds.set(1, 0)
```

Whoa!? A *double note!*

- Relax, this is the only case with two notes on a single LED.

The rest of the notes will be just like the first one:

- Add the `speaker.beep()` and adjust the `sleep()` time, and you're good!

```
22
23  # LED 2
24  # TODO
```

Your turn - Same pattern as the first note.

- Refer to the table in the overview for the LED/Note/Duration/Rest values.

```
25
26  # LED 3
27  # TODO
28
29  # LED 4
30  # TODO
31
32  # LED 5
33  # TODO
34
35  # LED 6
36  # TODO
37
38  # LED 7
39  # TODO
40
41  # LED STA
42  leds.set_status(50)
43  speaker.beep(G5, 500)
44  sleep(0.1)
45  leds.set_status(0)
```

The *Bonus* 🔧LED

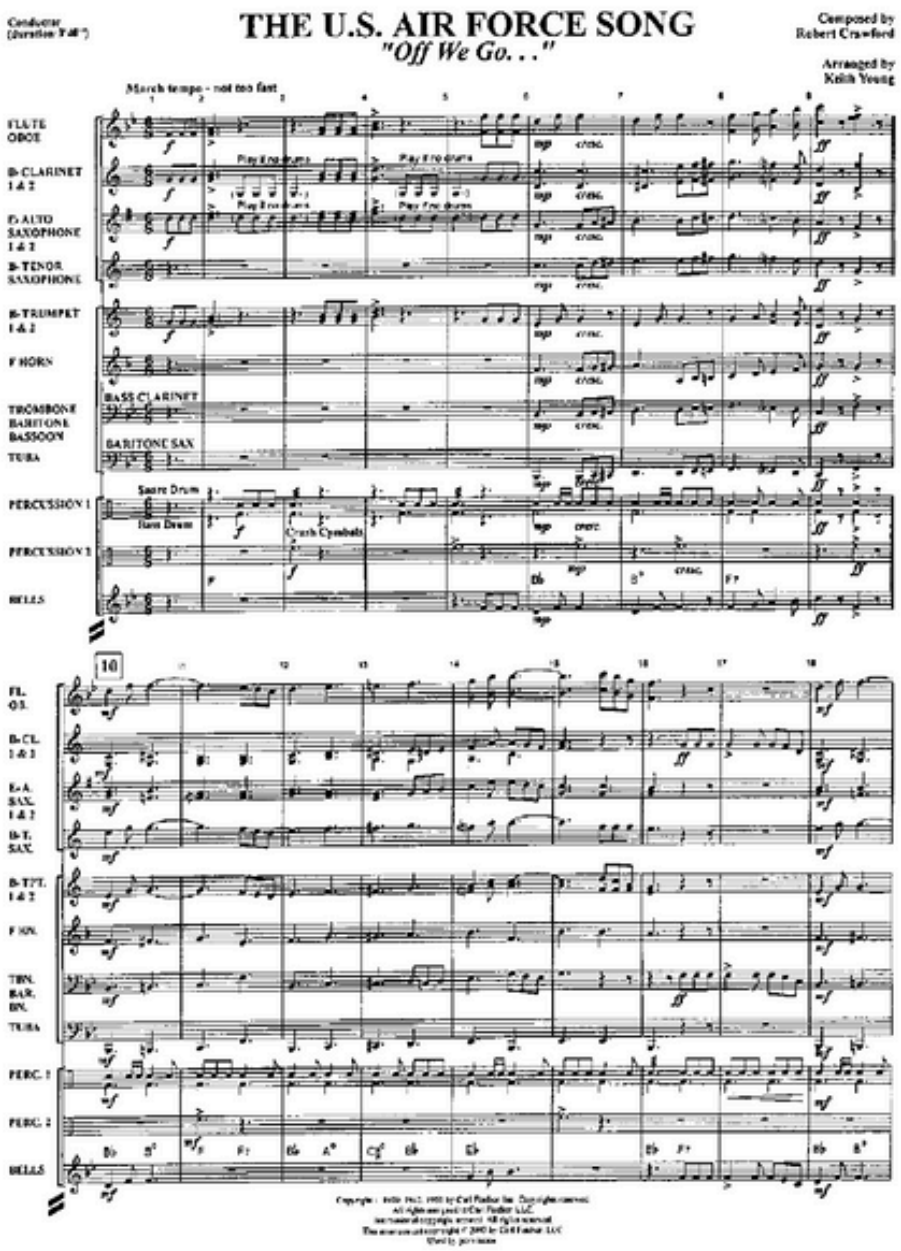
- Just needed *one* more. The cherry on top!
- Last one's on me :-)

Hint:

- ## The US Air Force

  Your melody captures the signature phrase from the official US Air Force Song:

  "Off we go, into the Wild Blue Yonder!"



Goal:

- Play the full melody as described above, complete with Light Show!

Tools Found: Speaker, Constants, Editor Shortcuts, Loops, BYTE LEDs

Solution:

```
from codeair import *
from time import sleep

# Notes used in the melody
D5 = 587
E5 = 659
F5 = 698
FS5 = 740
```

```
 9  G5 = 784
10
11
12  leds.set(0, 50)
13  speaker.beep(E5, 100)
14  sleep(0.2)
15  leds.set(0, 0)
16
17  leds.set(1, 50)
18  speaker.beep(G5, 100)
19  sleep(0.1)
20  speaker.beep(G5, 700)
21  sleep(0.3)
22  leds.set(1, 0)
23
24  leds.set(2, 50)
25  speaker.beep(F5, 100)
26  sleep(0.1)
27  leds.set(2, 0)
28
29  leds.set(3, 50)
30  speaker.beep(E5, 100)
31  sleep(0.1)
32  leds.set(3, 0)
33
34  leds.set(4, 50)
35  speaker.beep(D5, 100)
36  sleep(0.1)
37  leds.set(4, 0)
38
39  leds.set(5, 50)
40  speaker.beep(E5, 400)
41  sleep(0.1)
42  leds.set(5, 0)
43
44  leds.set(6, 50)
45  speaker.beep(F5, 400)
46  sleep(0.1)
47  leds.set(6, 0)
48
49  leds.set(7, 50)
50  speaker.beep(FS5, 400)
51  sleep(0.1)
52  leds.set(7, 0)
53
54  leds.set_status(50)
55  speaker.beep(G5, 500)
56  sleep(0.1)
57  leds.set_status(0)
```

***Objective 6* - In Living Color**

In Living Color!

Now that you have demonstrated full command over the blue 🔧LEDs, it's time to invite an even brighter and more colorful set of lights to the party!

- After all, if you want to put on an airborne *light show* you'll need really **dazzling** lights.
- How about some *really bright* LEDs you can program to any 🔧RGB Color you desire!?

Concept: *pixel LEDs*

There are 8 🔧pixel LEDs: 4 on top, and 4 below.

- They're numbered 0-7
- You can set them to any color with `pixels.set(n, color)`

Ex: *The following code sets pixel 0 to RED*

```
from codeair 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*.
- Use the color `BLACK` to turn the pixel OFF.

There are many more advanced pixel LED features you'll learn to use later.

Ready to Test Them?

Run It!

Goals:

- Get colorful. I want to see some pixels get lit up!
- Experiment with the code! Try an LED combination and run the code.
  - Then make some changes and run again. *Do this at least three times.*

Tools Found: BYTE LEDs, RGB Colors, RGB "pixel" LEDs

Solution:

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

***Objective 7* - Sky Lights**

Sky Lights

For your "Light Show Finale" the lights need to dance around CodeAIR!

- For that purpose a new Python tool will help.
  - *For* the win?

Concept: *for loop*

So far you have been using the `while` loop as your repeating workhorse.

- The `while` loop will always be your go-to for general purpose looping tasks. But sometimes your needs are more *specialized*.

Another Loop - the `for` loop

Often times when looping you are going through a *sequence* of some kind.

- This is called iterating.

Example:

```
from codeair import *

for color in (RED, GREEN, BLUE):
    pixels.set(0, color)
```

Can you guess what the above does? *Give it a try!*

For a change, try range

The range function is another awesome Python built-in function you'll often use with `for` loops:

Example:

```
from codeair import *

# Flash all 8 pixels in sequence
for n in range(8):
    pixels.set(n, GREEN)
    sleep(0.2)
    pixels.set(n, BLACK)
    sleep(0.2)
```

The above is the same as if you wrote `for n in (0, 1, 2, 3, 4, 5, 6, 7)` but much easier to type!

Run It!

Go ahead and try the examples above.

- Try adding more colors, or looping over a smaller range.

Finale

Armed with new looping tools, you can make your light show more dazzling than ever!

- You will be looping through colors
  - And for each color, looping through all eight LEDs

That's one `for` loop *nested* inside another `for` loop!

Check the 'Trek!

Surprise! There's not a lot of code there :-)

- More powerful coding tools let you do **more** with *less!*

Customize it!

- You could try more RGB Colors
- And since the numerical order of the pixels is a little odd, make your own sequence using the constants, for example: `(TOP_FRONT_LEFT, TOP_FRONT_RIGHT, TOP_REAR_RIGHT, TOP_REAR_LEFT)`

CodeTrek:

```
1  from codeair import *
2  from time import sleep
3
4  while True:
5      for color in (RED, GREEN, BLUE):
6          for n in range(8):
7              pixels.set(n, color)
8              sleep(0.05)
9
10
```

This is the whole solution!

- There's even a bonus *outer* `while` loop to keep the show running forever!

Goal:

- Dazzle me with colorful, flashing, cycling lights!

Tools Found: Loops, Iterable, Ranges, Built-In Functions, RGB Colors, RGB "pixel" LEDs, Constants

Solution:

```
from codeair import *
from time import sleep

while True:
    for color in (RED, GREEN, BLUE):
        for n in range(8):
            pixels.set(n, color)
            sleep(0.05)


```

Quiz 3 - More Loops

Question 1: What line of code defines a 🔧constant?

✓ `D5 = 587`

✗ `from time import sleep`

✗ `for n in range(8)`

✗ `pixels.set(n, color)`

Question 2: How many times will the sound beep?

```
from codeair import *
from time import sleep

for n in range(1, 2, 3):
    speaker.beep(440, 200)
    sleep(0.2)
```

✓ 3 times

✗ 2 times

✗ 1 time

✗ There is an error in the code.

Question 3: How many times will the pixel blink?

```
from codeair import *
from time import sleep

for n in range(5):
    pixels.set(0, BLUE)
    sleep(0.5)
    pixels.set(0, BLACK)
    sleep(0.5)
```

✓ 5 times

✗ 4 times

✗ Infinite

❌ There is an error in the code.

Question 4: How many total times will ***any*** pixel change color?

```
from codeair import *
from time import sleep

for color in (RED, WHITE, BLUE):
    for n in range(8):
        pixels.set(n, color)
        sleep(0.25)
```

✅ 24

❌ 8

❌ 3

❌ 16

***Objective 8* - Aero Lights**

Aeronautical Navigation Lighting

Ever notice the lights on aircraft at night? ...or on marine vessels for that matter?

- There's an international standard color scheme to indicate the orientation of the craft!

Drones Too?

Indoor drones aren't required to fly these colors, but it *can* be very helpful to visually identify orientation from across the room!

- Larger outdoor drones have FAA required anti-collision lights (more on that below).
- And while not strictly required for UAVs (Unmanned Aerial Vehicles), standard color navigation lights ensure that manned aircraft can see and avoid such drones.

Concept: *Standard Navigation Lights*

These are *solid* (not flashing) lights positioned as follows:

- Green on the right (starboard side)
- Red on the left (port side)
- White on the tail

Alright then, ready to light up CodeAIR properly?

Create a New File!

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

Navigation Lights for CodeAIR

Can you use the 🔧pixel LEDs to implement this lighting scheme?

- To assist, use the 🔧constants defined in the `codeair` module.

```
# TOP_FRONT_RIGHT, TOP_FRONT_LEFT, TOP_REAR_RIGHT, TOP_REAR_LEFT
# BOTTOM_FRONT_RIGHT, BOTTOM_FRONT_LEFT, BOTTOM_REAR_RIGHT, BOTTOM_REAR_LEFT
```

Run It!

Go ahead, start by typing in the code below.

- Test this out, and then add the other Navigation Lights.
- Be sure you have RED, GREEN, and WHITE lights - *TOP and BOTTOM!*

```
from codeair import *
pixels.set(TOP_FRONT_LEFT, RED)
```

Anti-Collision Lights

Check the 'Trek!

Run It!

Check your lighting! Would other pilots understand what's happening with your drone and be able to navigate around it?

- Even UAVs have to be mindful of international standards!

CodeTrek:

```
1  from codeair import *
2  from time import sleep
3
4  while True:
5      # Standard Navigation Lights (solid)
6      # Left = Portside
7      pixels.set(BOTTOM_FRONT_LEFT, RED)
8      pixels.set(TOP_FRONT_LEFT, RED)
9
10     # Right = Starboard
11     pixels.set(BOTTOM_FRONT_RIGHT, GREEN)
12     pixels.set(TOP_FRONT_RIGHT, GREEN)
13
14     # Back = Aft
15     pixels.set(BOTTOM_REAR_LEFT, WHITE)
16     pixels.set(TOP_REAR_LEFT, WHITE)
17     pixels.set(BOTTOM_REAR_RIGHT, WHITE)
18     pixels.set(TOP_REAR_RIGHT, WHITE)
```

Your basic, friendly neighborhood "Standard Navigation Lights".

- No mystery here... except why the `while` loop? *More on that next...*

```
19
20     # Anti-Collision Strobe Lights
21     sleep(1.0)
22     # TODO: bright whites...
23     sleep(0.02)
```

Anti-Collision Strobe Lights

- Once per second use the `pixels.fill(WHITE, brightness=50)` function to flash a bright `WHITE` pulse of light from all the 🔧pixels.

This 🔧loop sets the Standard Navigation Lights, then after 1 second blips
a very short pulse of bright `WHITE`, then restores Standard Lights again, and so on.

```
24
```

Goal:

- Show the Red/Green/White **Standard Navigation Lights** on all eight 🔧pixel LEDs.
    - Also implement **Anti-Collision Lights** as a once-per-second `WHITE` *strobe*.

Tools Found: RGB "pixel" LEDs, Constants, Loops

Solution:

```
from codeair import *
from time import sleep

while True:
    pixels.set(BOTTOM_FRONT_LEFT, RED)
    pixels.set(TOP_FRONT_LEFT, RED)
    pixels.set(BOTTOM_FRONT_RIGHT, GREEN)
    pixels.set(TOP_FRONT_RIGHT, GREEN)

    pixels.set(BOTTOM_REAR_LEFT, WHITE)
    pixels.set(TOP_REAR_LEFT, WHITE)
    pixels.set(BOTTOM_REAR_RIGHT, WHITE)
    pixels.set(TOP_REAR_RIGHT, WHITE)
    sleep(1.0)
    pixels.fill(WHITE, brightness=50)
    sleep(0.02)

```

Mission 3 Complete

Brilliant Lighting!

You've Learned So Much!

- Cycling those blue LEDs like a Python coding boss!
- Mastering the 🔧speaker with an aeronautically-approved melody!
- And adhering to international lighting standards. Awesome.

With Meaning

Hey, you're not just "pretending" to develop embedded software for a UAV here.

- You are *doing it!*
- CodeAIR is not gonna fly properly unless you make it so.
- And you're writing code the same way professional engineers do it, even on the most sophisticated drones!

Congrats on getting this far. Keep going on your journey!

Remix Plz!?

Oh, and one more thing. Take some time to remix what you just did.

- You need to really understand this stuff. Make changes to your code, experiment!
- From here on out, do NOT type in code if you don't at least THINK you know what it does!

Mission 4 **- Flight Safety**

Flight Safety

Ready to get those motors running?

- This Mission will get you there.
- But with *power* comes *responsibility!*

Quadcopter Safety Guidelines

CodeAIR is designed to be a durable and safe nano-quadcopter for use in close-proximity indoor environments. However, even small drones require caution to ensure safety. *Reckless or improper usage can lead to injury!*

- **Protective Gear** - *Always* wear face and eye protection, such as safety glasses, when operating CodeAIR or any powered device with moving parts.
- **Propeller Safety** - While CodeAIR's propellers are designed to minimize injury risks, you *must* avoid any contact with moving parts, especially around your face and eyes.
- **Supervised Use** - CodeAIR is safe for students but should always be used under adult supervision to reinforce safety protocols.
- **Flight Zone** - Ensure CodeAIR operates within a designated, clear area with no obstacles or bystanders within reach of the drone's flight path.

Mission Targets

In this mission you will code a set of *Safety Procedures.* CodeAIR will be flying autonomously under the control of the Python code *you* write and load into it. Just like the "Lights and Sounds" projects you have already completed, once the code is running you can just stand back and watch!

This mission will provide:

- A procedure to "Arm" CodeAIR reliably, so it never takes off "accidentally".
- A clear warning indicator, alerting people to stand back prior to takeoff.
- Understanding the quadcopter power system - motors and propellers.

Objective 1 **- Arm**

Safety Interlocks

Before you enable the motors on CodeAIR, you must code some *safety interlocks.*

- When making a product (including writing software) you need to think through the "user experience" or UX.
- Say your product is an autonomous drone that maps the dimensions of a room. *How would a user start the drone?*

Pre-Flight Steps

It could be surprising or even dangerous if CodeAIR started spinning propellers at full speed and jumped into the air immediately when the code runs. *So the way your "Light Show" code runs right away could be problematic!*

A much safer plan is for your drone to wait until a button is pressed. Then it can generate warning beeps from the 🔧speaker as well as some flashing warning lights on the 🔧pixel LEDs to show that the drone is "armed" for takeoff.

- After that, a **second** button press could confirm that the user is truly *ready to take off.*

Sounds pretty simple, but you're going to need a few new Python concepts to get this done!

Concept: *Button Input*

CodeAIR has two 🔧buttons you can *read* from your Python code: **B0 and B1**

Check for a 🔧button *B0* press with the following code:

```
buttons.was_pressed(BTN_0)  # True if button was was_pressed
```

Arm Button

Concept: *Branching 'if' statement*

CodeTrek:

```
from codeair import *
from time import sleep

while True:
```

The whole program 🔧loops *interminably!*

```

    # Wait for first "ARM" button press
    while True:
        # Blink
        leds.set(0, 50)  # LED near B0
        sleep(0.1)
        leds.set(0, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break
```

Okay, *another* `while` loop.

- This time you're blinking blue 🔧pixel LED `0` as you've done before.
- But wait... *what's this?*

  Every time around the 🔧loop you're checking for a button press!

- And if it's pressed: **BLAMMO! -** 🔧break right outta here.

```

    # Armed!
    pixels.fill(YELLOW)
```

Show the user that CodeAIR is ***ARMED!***

Note:

Check your 🔧indentation, people!

- I'm serious about this.
- See the nice *guidelines* showing the indentation levels.

Make your indentation neat and pretty!

```

    # Wait for second "CONFIRM" or "DISARM" button press
    while True:
        # Blink
        leds.set(7, 50)  # LED near B1
        sleep(0.1)
        leds.set(7, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break # Disarm
```

This looks *almost exactly* like the first 🔧loop.

- Only difference is now you're blinking LED 7, near button **B1**.

Soon you will add code here to check for B1, letting the user "CONFIRM" the launch.

- But for now, just check B0 → if it's pressed, *disarm* the drone.

```
30
31      # Disarmed
32      pixels.off()
```

Turn all the 🔧pixels off, the *easy* way!

```
33
```

Hints:

- ## Struggling?

  There are a lot of new concepts here, but the CodeTrek has your back on this one. All the code is given to you there.

  - If you are having trouble, compare your code to the CodeTrek.
  - Pay close attention to 🔧indentation.
  - If there are error messages, check closely on or near the line number where the error occurs.

- ## Bouncy Buttons?

  Yes, they are!

  - You'll tackle that in the next Objective.

- ## Need More Help?

  Ask an LLM! Here's the prompt I provided to ChatGPT to get the annotated code below:

  > "Can you provide an annotated version (comments inline with code) that explains this code to a very new Python programmer?"

  Here's a line-by-line explanation of the code, providing comments to guide a new Python programmer:

```python
from codeair import *  # Import everything from the codeair library, which has controls for CodeAIR.
from time import sleep  # Import the sleep function to pause the code for a specific time.

# Main loop that will keep the code running
while True:

    # Inner loop to wait for the first button press to "ARM" the drone.
    while True:
        # Blink the blue LED near button B0
        leds.set(0, 50)  # Turn on the LED near B0 at a brightness of 5   0.
        sleep(0.1)  # Pause for 0.1 seconds (a short blink).
        leds.set(0, 0)  # Turn off the LED (making it blink off).
        sleep(0.2)  # Pause for 0.2 seconds before the next blink.

        # Check if button B0 has been pressed
        if buttons.was_pressed(BTN_0):  # True if button B0 was pressed
            break  # Exit the inner loop if button B0 is pressed, which means the drone is now "armed".

    # Code to run when the drone is "armed" (after the first button press)
    pixels.fill(YELLOW)  # Set all pixel LEDs to yellow as a warning that the drone is armed.

    # Another loop to wait for the second button press to either "CONFIRM" (takeoff) or "DISARM"
    while True:
        # Blink the LED near button B1 to indicate the drone is armed and waiting for confirmation.
        leds.set(7, 50)  # Turn on the LED near B1 at brightness 5   0.
        sleep(0.1)  # Pause for 0.1 seconds.
        leds.set(7, 0)  # Turn off the LED, making it blink off.
        sleep(0.2)  # Pause for 0.2 seconds before the next blink.

        # Check if button B0 is pressed again to disarm the drone.
        if buttons.was_pressed(BTN_0):  # True if button B0 was pressed again.
            break  # Exit the inner loop if button B0 is pressed again, disarming the drone.

    # Code to run when the drone is disarmed (after the second button press)
    pixels.off()  # Turn off all the pixel LEDs to show the drone is no longer armed.
```

Goal:

- Run and test the code from the CodeTrek. *Arm and Disarm your drone!*

Tools Found: Speaker, RGB "pixel" LEDs, Buttons, Loops, BYTE LEDs, bool, Branching, Indentation, Break and Continue

Solution:

```
from codeair import *
from time import sleep

while True:

    # Wait for first "ARM" button press
    while True:
        # Blink
        leds.set(0, 50)  # LED near B0
        sleep(0.1)
        leds.set(0, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break

    # Armed!
    pixels.fill(YELLOW)

    # Wait for second "CONFIRM" or "DISARM" button press
    while True:
        # Blink
        leds.set(7, 50)  # LED near B1
        sleep(0.1)
        leds.set(7, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            # Disarm
            break

    # Disarmed
    pixels.off()

```

***Objective 2* - Debounce**

Debouncing the Button

You have encountered a *classic* electronics and robotics problem.

- At a microscopic level, the metal contacts of a button or switch often bounce a few times before coming to rest. So you might detect *two or more bounces* depending on how fast you're checking!
- Ah yes, engineers had to deal with this even before *I* was born!

You will solve this problem with code! But first, a bit more about the 🔧button functions.

Concept: *was_pressed() Back Story*

Consider how `buttons.was_pressed(BTN_0)` works. It actually does two things:

1. Return `True` if a button has been pressed.
    - Button presses are monitored by a 🔧CPU *interrupt handler*.
2. Reset the internal status of the button to `False`.
    - It won't return `True` unless the button is pressed again after the last `was_pressed(BTN_0)`.

Bad Bounce!

When the **button bounces**, here's the sequence:

1. User presses button... *now in slooow moootiiooon...*
2. First Contact!
3. `was_pressed(BTN_0)` → `True` *# we detected the first press!*
4. It's all good. The internal status of the button is reset to `False`.
5. **Bounce!!**
6. The CPU *interrupt handler* saves the *was_pressed* status.

Oh No! ...Next time around the loop when we call `was_pressed(BTN_0)` it will remember this bounce :-(

Concept: *Debounce*

Debouncing a button is quite easy:

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

You can use `sleep(0.1)` for step 2. *But what about **step 3?***

- How do you do you reset the internal button press status?

Easy! Just call `buttons.was_pressed(BTN_0)` again.

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

Check the 'Trek!

Run It!

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

CodeTrek:

```
1  from codeair import *
2  from time import sleep
3
4  # Repeat this test program forever
5  while True:
6
7      # Wait for first "ARM" button press
8      while True:
9          # Blink
10         leds.set(0, 50)  # LED near B0
11         sleep(0.1)
12         leds.set(0, 0)
13         sleep(0.2)
14
15         if buttons.was_pressed(BTN_0):
16             break
17
18     # Armed!
19     pixels.fill(YELLOW)
20
21     # Debounce
```

Debounce Here

Just add these two sweet lines of Python code! Remember, no need to type in my 🔧comments.

```
# Debounce
sleep(0.1)  # Wait for bouncing to stop
```

```
    buttons.was_pressed()  # Clear internal state
```

Note:

Above I used `buttons.was_pressed()` without specifying `BTN_0`. That's not a mistake!

- When you call it this way it reads BOTH 🔧buttons at once.
- That will be helpful later when you're checking **B1** also.

```
22
23      # Wait for second "CONFIRM" or "DISARM" button press
24      while True:
25          # Blink
26          leds.set(7, 50)  # LED near B1
27          sleep(0.1)
28          leds.set(7, 0)
29          sleep(0.2)
30
31          if buttons.was_pressed(BTN_0):
32              break # Disarm
33
34      # Disarmed
35      pixels.off()
36
37      # Debounce
```

Debounce Here

You guessed it. *Just like before!*

```
38
```

Goal:

- Modify your code to eliminate the "double-press" bug.
  - No more *contact-bounce!*

Tools Found: Buttons, CPU and Peripherals, Comments

Solution:

```
1   from codeair import *
2   from time import sleep
3
4   # Repeat this test program forever
5   while True:
6
7       # Wait for first "ARM" button press
8       while True:
9           # Blink
10          leds.set(0, 50)  # LED near B0
11          sleep(0.1)
12          leds.set(0, 0)
13          sleep(0.2)
14
15          if buttons.was_pressed(BTN_0):
16              break
17
18      # Armed!
19      pixels.fill(YELLOW)
20
21      # Debounce
22      sleep(0.1)
23      buttons.was_pressed()
24
25      # Wait for second "CONFIRM" or "DISARM" button press
26      while True:
27          # Blink
28          leds.set(7, 50)  # LED near B1
```

```
29          sleep(0.1)
30          leds.set(7, 0)
31          sleep(0.2)
32
33          if buttons.was_pressed(BTN_0):
34              break # Disarm
35
36      # Disarmed
37      pixels.off()
38
39      # Debounce
40      sleep(0.1)
41      buttons.was_pressed()
42
```

Quiz 1 **- Buttons!**

Question 1: What does the `break` statement do?

✓ Breaks out of a loop

✗ Causes the code to stop

✗ Jumps over the next line of code

✗ Crashes the program

Question 2: What command checks to see if B0 was pressed?

✓ `buttons.was_pressed(BTN_0)`

✗ `buttons.is_pressed(B0)`

✗ `buttons(BTN_0, pressed)`

✗ `buttons.pressed(B0)`

Question 3: What is the purpose of the code:

```
while True:
    if buttons.was_pressed(BTN_0):
        break
```

✓ Pause the code until B0 is pressed.

✗ Pause the code when B0 is pressed.

✗ End the program when B0 is pressed.

✗ Loops the button press continuously.

Question 4: What code will debounce a button?

✓ `ERROR: Invalid Code Block!! sleep(0.1) buttons.was_pressed()

ERROR: Invalid Code Block!!

✗ `ERROR: Invalid Code Block!! while True: if buttons.was_pressed(BTN_0): break

ERROR: Invalid Code Block!!

✗ `buttons.debounce(BTN_0)`

❌ `buttons.was_pressed() = False`

***Objective 3* - Countdown**

Countdown to Launch

Now that you have the ARM process all sorted, it's time to add 🔧button **B1** to confirm the LAUNCH.

- Once it is confirmed, sound a warning alert using the 🔧speaker.
- Also flash the 🔧pixel LEDs `RED` so the user knows to *STAND CLEAR!*

Check the 'Trek!

You'll be adding another 🔧if condition: with a button check for **B1**.

- Oh, and this is where you add sounds and an awesome WARNING **countdown**!
- The additional code should be pretty familiar to you by now.

Run It!

This is getting *exciting!*

CodeTrek:

```
1  from codeair import *
2  from time import sleep
3
4  # Repeat this test program forever
5  while True:
6
7      # Wait for first "ARM" button press
8      while True:
9          # Blink
10         leds.set(0, 50)  # LED near B0
11         sleep(0.1)
12         leds.set(0, 0)
13         sleep(0.2)
14
15         if buttons.was_pressed(BTN_0):
16             break
17
18     # Armed!
19     pixels.fill(YELLOW)
20     speaker.beep(??, 100)  # TODO: fill in frequency
21     speaker.beep(??, 50)   # TODO: fill in frequency
```

All About the UX

User experience, that is.

- Add a little 2-tone confirmation beep to give the user some audible feedback after *arming.*
- *You need to choose the frequencies of the two beeps!*
  - Might I suggest 500Hz and 1000 Hz?

```
22
23     # Debounce
24     sleep(0.1)
25     buttons.was_pressed()
26
27     # Wait for second "CONFIRM" or "DISARM" button press
28     while True:
29         # Blink
```

```
30          leds.set(7, 50)  # LED near B1
31          sleep(0.1)
32          leds.set(7, 0)
33          sleep(0.2)
34
35          if buttons.was_pressed(BTN_0):
36              break  # Disarm
37
38          elif buttons.was_pressed(BTN_1):
39              # Confirmed! Start countdown...
```

Add the check for `BTN_1`

Notice this looks a lot like the `if` check for `BTN_0`.

- But here it's `elif` - short for *"else if"*.
- Check the branching tool for more details on that.

```
40              for i in range(4):
41                  pixels.off()
42                  sleep(0.5)
43                  pixels.fill(RED)
44                  # TODO: beep at 800Hz for 500ms
```

The Countdown!

A neat little `for` loop.

- In this case, you just need to flash and beep four times.
- The variable `i` keeps track of the count 0, 1, 2, 3 (but otherwise you aren't using it!)

TODO!

You need to add a `speaker.beep(800, 500)` here.

```
45
46              # Launch!
47              break
```

Here's where you will add the **motor spin-up** code in the next Objective!

- For now after the countdown your code just hits the `break` statement, going back to the top: *disarmed and waiting.*

```
48
49      # Disarmed
50      pixels.off()
51
52      # Debounce
53      sleep(0.1)
54      buttons.was_pressed()
55
```

Goals:

- Add the `elif` statement and countdown `for` loop.
- Add a double-beep on ARM, and warning beep prior to launch.

Tools Found: Buttons, Speaker, RGB "pixel" LEDs, Branching, Loops, Variables

Solution:

```
1  from codeair import *
2  from time import sleep
3
4  # Repeat this test program forever
5  while True:
6
```

```
    # Wait for first "ARM" button press
    while True:
        # Blink
        leds.set(0, 50)  # LED near B0
        sleep(0.1)
        leds.set(0, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break

    # Armed!
    pixels.fill(YELLOW)
    speaker.beep(500, 100)
    speaker.beep(1000, 50)

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

    # Wait for second "CONFIRM" or "DISARM" button press
    while True:
        # Blink
        leds.set(7, 50)  # LED near B1
        sleep(0.1)
        leds.set(7, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break  # Disarm

        elif buttons.was_pressed(BTN_1):
            # Confirmed! Start countdown...
            for i in range(4):
                pixels.off()
                sleep(0.5)
                pixels.fill(RED)
                speaker.beep(800, 500)

            # Launch!

            break

    # Disarmed
    pixels.off()

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

```

***Objective 4* - Motor Test**

Motor Test

With the *safety interlocks* fully in place and tested, it's time to spin up those motors!

- This step will only be at "test speed", so you can first confirm 🔧motor operation without taking off into the air.

Test Spin!!

The 🔧flight module has a `motor_test()` 🔧function that's perfect for this kind of testing. The motors will spin-up, but not fast enough to lift off :-)

Check the 'Trek!

A new 🔧flight 🔧module is joining the party.

- You'll learn a lot about that module in future Missions!

Run It!

It's all coming together now. The *safety* code makes this a *breeze!*

⚠ **Note:** The **battery** must be connected to run the motors! ⚠

Physical Interaction: *Watch Your Fingers*

Investigate the motors as they run!

- Can you confirm the thrust is all downward?
- Are the 🔧propellers spinning the same direction? Check the toolbox to see why *not!*

CodeTrek:

```
1  from codeair import *
2  from time import sleep
3  from flight import *
```

`import` everything from the 🔧flight module.

```
4
5  # Repeat this test program forever
6  while True:
7
8      # Wait for first "ARM" button press
9      while True:
10         # Blink
11         leds.set(0, 50)  # LED near B0
12         sleep(0.1)
13         leds.set(0, 0)
14         sleep(0.2)
15
16         if buttons.was_pressed(BTN_0):
17             break
18
19     # Armed!
20     pixels.fill(YELLOW)
21     speaker.beep(500, 100)
22     speaker.beep(1000, 50)
23
24     # Debounce
25     sleep(0.1)
26     buttons.was_pressed()
27
28     # Wait for second "CONFIRM" or "DISARM" button press
29     while True:
30         # Blink
31         leds.set(7, 50)  # LED near B1
32         sleep(0.1)
33         leds.set(7, 0)
34         sleep(0.2)
35
36         if buttons.was_pressed(BTN_0):
37             break  # Disarm
38
39         elif buttons.was_pressed(BTN_1):
40             # Confirmed! Start countdown...
41             for i in range(4):
42                 pixels.off()
43                 sleep(0.5)
44                 pixels.fill(RED)
45                 speaker.beep(800, 500)
46
47             # Launch!
48             pixels.fill(GREEN)
```

```
            motor_test(True)
            sleep(3)
            motor_test(False)
            break
```

Motor Test

This will run the motors for 3 seconds.

- And fill the pixel LEDs with `GREEN` while they run.

```


    # Disarmed
    pixels.off()

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

```

Goals:

- Import the `flight` module.
- Run the code and test those motors!

Tools Found: Motors and Props, Flight Module, Functions, import, RGB "pixel" LEDs

Solution:

```
from codeair import *
from time import sleep
from flight import *

# Repeat this test program forever
while True:

    # Wait for first "ARM" button press
    while True:
        # Blink
        leds.set(0, 50)  # LED near B0
        sleep(0.1)
        leds.set(0, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break

    # Armed!
    pixels.fill(YELLOW)
    speaker.beep(500, 100)
    speaker.beep(1000, 50)

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

    # Wait for second "CONFIRM" or "DISARM" button press
    while True:
        # Blink
        leds.set(7, 50)  # LED near B1
        sleep(0.1)
        leds.set(7, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break  # Disarm

        elif buttons.was_pressed(BTN_1):
            # Confirmed! Start countdown...
```

```
41              for i in range(4):
42                  pixels.off()
43                  sleep(0.5)
44                  pixels.fill(RED)
45                  speaker.beep(800, 500)
46
47              # Launch!
48              pixels.fill(GREEN)
49              motor_test(True)
50              sleep(3)
51              motor_test(False)
52              break
53
54      # Disarmed
55      pixels.off()
56
57      # Debounce
58      sleep(0.1)
59      buttons.was_pressed()
60
```

***Objective 5* - Functions**

Custom Tools

The *safety* check you have developed in this Mission is a very useful tool.

- *When might it be useful?*
- Any time you want to let the user start up the drone with a button-press!

Software Engineering

Making "reusable components" is a major goal of Software Engineering.

- Consider your **"Button Arm"** code. You wouldn't want to have to start over and write that from scratch every time you needed it!
- As you've seen, it takes effort to get the code just right.

First Steps to Reusability

You've already experienced Python's reusability features:

- When you `from codeair import *` you're re-using a Python 🔧module that contains code for `buttons`, `leds`, and more!
- You are using 🔧functions like `motor_test()` and `sleep()`. Those are just chunks of code someone else wrote, so you don't have to!

Concept: *Functions*

When you write some code that you'd like to use over and over again, you should put it in a 🔧function.

Here's how you would `def`ine a function that 🔧returns `0` for BTN_0, and `1` for BTN_1. Say you decide to name the new function `any_button()` :

```
def any_button():
    if buttons.was_pressed(BTN_0):
        return 0
    elif buttons.was_pressed(BTN_1):
        return 1
```

Once it's defined, you can *call* the 🔧function whenever needed:

```
if any_button() == 1:
    # Action when BTN_1 is pressed
```

Your First Function

The **"Button Arm"** code will be an excellent function to use whenever you need a safe way to start flying.

- You are going to need this 🔧function later!

Check the 'Trek!

The CodeTrek will show you a few small changes to *package* your code into a function.

- *AFTER* the function is defined you can *call* it as part of your motor test!

CodeTrek:

```
from codeair import *
from time import sleep
from flight import *

def button_arm():
    do_launch = False
```

1. Replace your `while` loop with a function `def`.
2. Add a `do_launch` variable, and *initialize* it to `False`. You'll set this to `True` below if the user *confirms* with **B1**.

```

    # Wait for first "ARM" button press
    while True:
        # Blink
        leds.set(0, 50)  # LED near B0
        sleep(0.1)
        leds.set(0, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break

    # Armed!
    pixels.fill(YELLOW)
    speaker.beep(500, 100)
    speaker.beep(1000, 50)

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

    # Wait for second "CONFIRM" or "DISARM" button press
    while True:
        # Blink
        leds.set(7, 50)  # LED near B1
        sleep(0.1)
        leds.set(7, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break  # Disarm

        elif buttons.was_pressed(BTN_1):
            # Confirmed! Start countdown...
            for i in range(4):
                pixels.off()
                sleep(0.5)
                pixels.fill(RED)
                speaker.beep(800, 500)

            # Launch!
            do_launch = True
            break
```

User confirmed launch!

1. ***Cut*** your motor test code, and ***paste*** it at the bottom of the file.
2. Replace it with `do_launch = True` as shown here.

```
50
51      # Disarmed
52      pixels.off()
53
54      # Debounce
55      sleep(0.1)
56      buttons.was_pressed()
57
58      return do_launch
```

The **last** thing your new 🔧function does is 🔧return a value.

- In this case, `True` means CONFIRMED and `False` means DISARMED.

```
59
60  # Now that the function is defined, here's the test program:
61  while True:
62      if button_arm():
63          pixels.fill(GREEN)
64          motor_test(True)
65          sleep(3)
66          motor_test(False)
```

Test Program

Notice, you are now OUTSIDE the 🔧function!

- Check the 🔧indentation to be sure.

Just a simple loop, checking whether `button_arm()` is `True` or not.

```
67          pixels.off()
68
```

Goals:

- Convert your big `while` 🔧loop to a *reusable* 🔧function called `button_arm()`.
- After defining your new function, call it from a test program:
  - A `while` loop that uses `button_arm()` to check for *launch confirmation.*
  - ...and spins up those 🔧motors!

Tools Found: import, Functions, Parameters, Arguments, and Returns, Loops, Motors and Props, Variables, Indentation

Solution:

```
1  from codeair import *
2  from time import sleep
3  from flight import *
4
5  def button_arm():
6      do_launch = False
7
8      # Wait for first "ARM" button press
9      while True:
10         # Blink
11         leds.set(0, 50)  # LED near B0
12         sleep(0.1)
13         leds.set(0, 0)
14         sleep(0.2)
15
16         if buttons.was_pressed(BTN_0):
17             break
18
19     # Armed!
20     pixels.fill(YELLOW)
21     speaker.beep(500, 100)
```

```
    speaker.beep(1000, 50)

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

    # Wait for second "CONFIRM" or "DISARM" button press
    while True:
        # Blink
        leds.set(7, 50)  # LED near B1
        sleep(0.1)
        leds.set(7, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break  # Disarm

        elif buttons.was_pressed(BTN_1):
            # Confirmed! Start countdown...
            for i in range(4):
                pixels.off()
                sleep(0.5)
                pixels.fill(RED)
                speaker.beep(800, 500)

            # Launch!
            do_launch = True
            break

    # Disarmed
    pixels.off()

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

    return do_launch

# Now that the function is defined, here's the test program:
while True:
    if button_arm():
        pixels.fill(GREEN)
        motor_test(True)
        sleep(3)
        motor_test(False)
        pixels.off()

```

Quiz 2 **- Default Quiz**

Question 1: What is printed?

```
x = 5
if x < 5:
    print('Hello')
elif x > 5:
    print('World')
```

✓ Nothing is printed

✗ Hello

✗ World

✗ Hello World

Question 2: What does `==` mean in `if choice == 1`?

✓ Returns `True` if `choice` is the same as `1`

❌ Assigns `1` to the variable `choice`

❌ Selects either `choice` or `1`

❌ It causes an error

Question 3: What line of code will call this function?

```
def any_button():
    if buttons.was_pressed(BTN_0):
        return 0
    elif buttons.was_pressed(BTN_1):
        return 1
```

✅ `if any_button() == 1:`

❌ `any_button()`

❌ `call any_button()`

❌ `def any_button():`

Question 4: What will be printed if B1 is pressed:

```
def any_button():
    if buttons.was_pressed(BTN_0):
        return 0
    elif buttons.was_pressed(BTN_1):
        return 1

if any_button() == 0:
    print('Hello')
else:
    print('World')
```

✅ World

❌ Hello

❌ 1

❌ Nothing will be printed

***Objective 6* - Torque**

Quadcopter Physics

One more bit of *quadcopter physics* you need to understand before we go further.

- Watch the 🔧propellers carefully just as they stop, and you'll notice something interesting.
  - The BLACK propellers rotate clockwise (CW)
  - The RED propellers rotate counterclockwise (CCW)

This is critical to the flight of these machines, but why?

Concept: *Torque*

A rotational force, which is what your 🔧motors produce, is called ***Torque***.

And when you produce torque in one direction, there is naturally an opposing force in the *opposite* direction.

- If you were on skates and spinning a big propeller over your head, your body would spin the other direction.
- And when a power drill bit spins up, you have to hold tight to keep the handle from rotating the opposite direction!

Newton's 3rd Law

Check out the toolbox 🔧motors entry for more information on the forces at play here.

- The bottom line is: two of the propellers need to rotate the *opposite* direction!
- By doing this, the forces cancel-out. *You must bring balance to the force!*

Prove It!

It's said that if all the propellers went the same direction, the drone would just rotate uncontrollably in the *opposite* direction.

- Can you prove that, with a ground-based *test?*
- A short pulse of *the right two* 🔧propellers should make the drone rotate briefly in the opposite direction, right?

⚠️ Warning: Grown Up Motor Tests! ⚠️

- CodeAIR is designed to put YOU in full control. *I'm trusting you here.*
- In case you're wondering: YES, you could set the motor speed much higher and it would likely spin uncontrollably into the air.

  Please don't do that! Be responsible with your drone, and safe with yourself and fellow humans.

Check the 'Trek!

The CodeTrek shows how to replace your motor test with a "torque test" that uses the low-level *parameter* system of the 🔧flight module to enable just the two RED (CCW) motors briefly.

Physical Interaction: *Slippery Slope*

After you've loaded and tested this code, unplug the USB and place it on a very smooth desk or other surface.

- Can you see the how the body of CodeAIR rotates?
- Was Newton right about the "opposite" direction of the force?

CodeTrek:

```
from codeair import *
from time import sleep
from flight import *

def button_arm():
    do_launch = False

    # Wait for first "ARM" button press
    while True:
        # Blink
        leds.set(0, 50)  # LED near B0
        sleep(0.1)
        leds.set(0, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break

    # Armed!
```

```
    pixels.fill(YELLOW)
    speaker.beep(500, 100)
    speaker.beep(1000, 50)

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

    # Wait for second "CONFIRM" or "DISARM" button press
    while True:
        # Blink
        leds.set(7, 50)  # LED near B1
        sleep(0.1)
        leds.set(7, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break  # Disarm

        elif buttons.was_pressed(BTN_1):
            # Confirmed! Start countdown...
            for i in range(4):
                pixels.off()
                sleep(0.5)
                pixels.fill(RED)
                speaker.beep(800, 500)

            # Launch!
            do_launch = True
            break

    # Disarmed
    pixels.off()

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

    return do_launch

# Now that the function is defined, here's the test program:
while True:
    if button_arm():
        pixels.fill(GREEN)
        # BRIEF pulse of RED (CCW) motors
        set_param('motorPowerSet.m2', 30000)
        set_param('motorPowerSet.m3', 30000)
        set_param('motorPowerSet.enable', 1)
        sleep(0.2)
        set_param('motorPowerSet.enable', 0)
        pixels.off()
```

Motor Pulse

Copy and paste this code, *replacing* your motor test between `pixels.fill(GREEN)` and `pixels.off()`.

```
        # BRIEF pulse of RED (CCW) motors
        set_param('motorPowerSet.m2', 30000)
        set_param('motorPowerSet.m3', 30000)
        set_param('motorPowerSet.enable', 1)
        sleep(0.2)
        set_param('motorPowerSet.enable', 0)
```

Goal:

- Modify your code to run the Torque test!
    - Use the `set_param()` functions as shown in the CodeTrek

Tools Found: Motors and Props, Flight Module

Solution:

```python
from codeair import *
from time import sleep
from flight import *

def button_arm():
    do_launch = False

    # Wait for first "ARM" button press
    while True:
        # Blink
        leds.set(0, 50)  # LED near B0
        sleep(0.1)
        leds.set(0, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break

    # Armed!
    pixels.fill(YELLOW)
    speaker.beep(500, 100)
    speaker.beep(1000, 50)

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

    # Wait for second "CONFIRM" or "DISARM" button press
    while True:
        # Blink
        leds.set(7, 50)  # LED near B1
        sleep(0.1)
        leds.set(7, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break  # Disarm

        elif buttons.was_pressed(BTN_1):
            # Confirmed! Start countdown...
            for i in range(4):
                pixels.off()
                sleep(0.5)
                pixels.fill(RED)
                speaker.beep(800, 500)

            # Launch!
            do_launch = True
            break

    # Disarmed
    pixels.off()

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

    return do_launch

# Now that the function is defined, here's the test program:
while True:
    if button_arm():
        pixels.fill(GREEN)
        # BRIEF pulse of RED (CCW) motors
        set_param('motorPowerSet.m2', 30000)
        set_param('motorPowerSet.m3', 30000)
        set_param('motorPowerSet.enable', 1)
        sleep(0.2)
        set_param('motorPowerSet.enable', 0)
        pixels.off()


```

Mission 4 Complete

Safety Protocol Complete

Nice Work!

- You now have startup code you can use for future Missions.
- You've learned about the risks, and how to safely operate CodeAIR.
- And you have a hands-on understanding of the quadcopter power plant!
- What's more, your Python coding skills have grown immensely!

Be Safe Out There

Safe flying is *your* responsibility!

Mission 5 **- Hovering Flight**

Take Flight

In this Mission you'll get CodeAIR flying, and begin your journey into the world of *Sensor-Based Navigation.*

Standing on the Shoulders of *Code*

You will begin by learning about *custom* 🔧*modules* - taking the work you did in the last Mission building a *launch safety system* and putting it to good use!

You'll end by mastering an ***Escape Room*** challenge, where CodeAIR must use its FORWARD 🔧laser ranger to avoid walls and seek the exit.

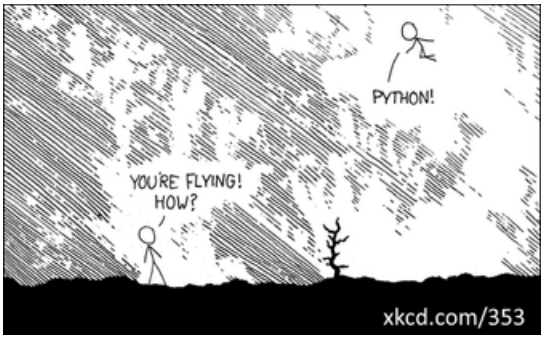
Mission Targets

There's much to cover in this Mission -

- Using *console* output 🔧print() statement in Python.
- Flying with the 🔧MotionCommander interface.
- Blocking versus Non-Blocking 🔧functions.
- Measuring precise distances with CodeAIR's 🔧laser rangers.
- Working with 🔧variables in Python.
- *...and much more!*

Objective 1 **- Modular**

Defying Gravity!



It's time to get this thing off the ground.

- But, you surely know by now, there is gonna be some coding involved!
- At right is a classic Python meme from XKCD: `import antigravity`
- *Python makes it so simple!*

Alas, there's no *antigravity* module here. But you do have another important 🔧import to attend to.

Importing the Safety Protocol

First off, do you still have your **safety.py** code handy?

- Rather than adding to that, how about making it a 🔧module?
- Then you can `import` it whenever you need it!

Concept: *Custom Modules*

Creating a Python module can be as simple as placing a code file like "foo.py" in the same folder as your program, and then just typing `import foo`.

- Notice, you don't need the file extension (.py) in the `import` statement.

So if you already have a file called "safety.py" loaded on CodeAIR[1] you can write `import safety` and use the 🔧functions defined therein.

---

1. CodeSpace will automatically keep files named with the .py extension loaded on CodeAIR.

File → Open "safety.py"

Open up your safety.py program, if it's not already open.

- If for some reason you don't have the file, I've provided it below to copy.
- You *will* need to edit your version to **remove** the test code after the function. Otherwise the test code will run when you `import` it, which is *definitely not* what you want!
- For more details on custom modules check the Hints.

Check the 'Trek!

There are a couple of small changes you need to make to your *safety.py* program, to be sure it's a nice, well-behaved module.

Open the ≡ Console

See that "hamburger" icon at the bottom right of your window?

- Gimme dat! *...I'm jokin'*
- But seriously, click the ≡ Console icon to open a window so you can see print() output from your Python program.
- While CodeAIR is connected via USB, your code can `print()` messages there!

Run It!

To "install" this module, you'll need to click the ▶ RUN button.

- When you do so, check the ≡ Console to see the "Loaded..." message.

CodeTrek:

```
1  """Safety Module - provide functions for safe CodeAIR operation."""
2  from codeair import *
```

A documentation string ("docstring") at the top of the file.

- Related to comments, these strings don't affect how your program runs, but they are *essential* for folks trying to understand your code later!

```
3  from time import sleep
4  from flight import *
5
6  def button_arm():
7      do_launch = False
8
9      # Wait for first "ARM" button press
10     while True:
11         # Blink
12         leds.set(0, 50)  # LED near B0
13         sleep(0.1)
14         leds.set(0, 0)
15         sleep(0.2)
16
17         if buttons.was_pressed(BTN_0):
18             break
19
20     # Armed!
21     pixels.fill(YELLOW)
22     speaker.beep(500, 100)
23     speaker.beep(1000, 50)
24
25     # Debounce
26     sleep(0.1)
27     buttons.was_pressed()
28
29     # Wait for second "CONFIRM" or "DISARM" button press
30     while True:
31         # Blink
```

```
32          leds.set(7, 50)  # LED near B1
33          sleep(0.1)
34          leds.set(7, 0)
35          sleep(0.2)
36
37          if buttons.was_pressed(BTN_0):
38              break  # Disarm
39
40          elif buttons.was_pressed(BTN_1):
41              # Confirmed! Start countdown...
42              for i in range(4):
43                  pixels.off()
44                  sleep(0.5)
45                  pixels.fill(RED)
46                  speaker.beep(800, 500)
47
48              # Launch!
49              do_launch = True
50              break
51
52      # Disarmed
53      pixels.off()
54
55      # Debounce
56      sleep(0.1)
57      buttons.was_pressed()
58
59      return do_launch
60
61  # Print to console if running standalone
62  if __name__ == '__main__':
63      print("Loaded safety.py")
64
```

1. Remove the test code at the bottom of the file.
2. Add this strange `if` statement. This will `print()` a message to the 🔧console when you first RUN the module on CodeAIR.

The 🔧condition `__name__ == '__main__'` will be `True` when this program (safety.py) is running as the "main" program on CodeAIR. That will *not* be the case when you `import` it from other programs!

Hints:

- **Custom Modules**

  In Python, importing a file as a 🔧module is like telling your program to use code from another file. Think of it as borrowing functions or settings from a "helper file" that you can use in your main program. Here's how it works:

  1. **Creating a Module**: If you want to create a module, you just need to write your Python code in a separate file (let's say you name it "tools.py") and save it in the same folder as your main program. This file can contain any Python code you want to reuse, like 🔧functions or 🔧variables.
  2. **Using the Module**: Once you have your "tools.py" file ready, you can bring its contents into your main program by typing `import tools`. This tells Python to load everything in "tools.py" so you can use it in your program. For example, if "tools.py" has a function called `measure_distance`, after you import it, you can use it by writing `tools.measure_distance()` in your main code.
  3. **No File Extension Needed in Import**: When importing, you only need the file name without the `.py` extension. For instance, if your file is named "safety.py," you would import it by writing `import safety` — not `import safety.py`.
  4. **Loading Files onto CodeAIR**: In CodeSpace, any `.py` file that you RUN will automatically be retained in CodeAIR's flash filesystem. This means that any programs you have run with a `.py` extension will be ready for `import` by future programs.

  This setup allows you to keep your code organized by separating reusable pieces into different files, making it easier to manage and reuse them in different programs.

- **Loading Files on CodeAIR**

  As mentioned in the Objective overview, the simple act of running a file named with the standard `.py` extension will signal CodeSpace that the file should be *persisted* on the CodeAIR filesystem.

If you open your OS file browser, and view the "flash drive" that appears when CodeAIR is connected, you will see the files which have been loaded in this way.

- The currently loaded "main program" - the one that runs when CodeAIR boots up - will be called "main.py", regardless of what it is named in CodeSpace!
- If you named it with a `.py` extension in CodeSpace, it will ALSO be saved to CodeAIR by its proper Python filename, making it available for later import.

Note:

When you view the filesystem with your OS file browser, it often won't immediately show files which have been written by Python or CodeSpace via WebUSB.

- Computers don't really expect flash drives to write themselves!
- Unplug CodeAIR, reconnect, refresh,... the files are there.

You can also load files *outside* of CodeSpace. More information on that here:

- Working with files

Goal:

- Run the "safety.py" program to install it onto CodeAIR.
  - I'm looking for "Loaded safety.py" on the console!

Tools Found: import, Functions, Print Function, Motors and Props, Comments, bool

Solution:

```
"""Safety Module - provide functions for safe CodeAIR operation."""
from codeair import *
from time import sleep
from flight import *

def button_arm():
    do_launch = False

    # Wait for first "ARM" button press
    while True:
        # Blink
        leds.set(0, 50)  # LED near B0
        sleep(0.1)
        leds.set(0, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break

    # Armed!
    pixels.fill(YELLOW)
    speaker.beep(500, 100)
    speaker.beep(1000, 50)

    # Debounce
    sleep(0.1)
    buttons.was_pressed()

    # Wait for second "CONFIRM" or "DISARM" button press
    while True:
        # Blink
        leds.set(7, 50)  # LED near B1
        sleep(0.1)
        leds.set(7, 0)
        sleep(0.2)

        if buttons.was_pressed(BTN_0):
            break  # Disarm

```

```
40          elif buttons.was_pressed(BTN_1):
41              # Confirmed! Start countdown...
42              for i in range(4):
43                  pixels.off()
44                  sleep(0.5)
45                  pixels.fill(RED)
46                  speaker.beep(800, 500)
47
48              # Launch!
49              do_launch = True
50              break
51
52      # Disarmed
53      pixels.off()
54
55      # Debounce
56      sleep(0.1)
57      buttons.was_pressed()
58
59      return do_launch
60
61  # Print to console if running standalone
62  if __name__ == '__main__':
63      print("Loaded safety.py")
64
```

***Objective 2* - Hover**

Hover!

With your safety code in place, it's time to take flight!

- *You are going to be amazed at how simple the code is now.*

Flight Sensors

CodeAIR uses *sensors* for autonomous flight.

- For altitude, a 🔧pressure sensor and 🔧laser rangers are used.
- For tracking and holding position, an optical 🔧flow sensor is used.

Air Space

For best results when flying, be sure to have:

- **Good lighting** - nice bright room lighting, no harsh shadows.
- **Floor space** - take off from *floor,* with no obstacles within a 1 meter radius.

Make sure the floor your drone flies over has some "pattern" so the 🔧flow sensor can maintain a stable horizontal position. *(refer to the toolbox for more details)*

Concept: *Motion Commander*

The flight module provides the `fly` object, exposing a high-level flight control interface called MotionCommander.

Behind the scenes, MotionCommander sends *velocity setpoints* to CodeAIR's flight controller, and it uses its *sensors* to try to maintain stable flight while executing commands like:

```
fly.take_off(height_meters)
fly.steady(seconds)
fly.turn_right(degrees)
fly.land()
```

Note:

There is no `sleep()` allowed while flying!

- The `fly.steady()` function allows your code to pause while keeping the flight controller running.

Create a New File!

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

Check the 'Trek!

Run It!

Make a few *test flights...*

CodeTrek:

```
1  from codeair import *
2  from flight import *
3  from safety import *
```

Import your shiny new module!

- This assume, of course, that you loaded "safety.py" onto CodeAIR in the last Objective.

```
4
5  # Repeat the flight test
6  while True:
7
8      # Safety-check button press
9      if button_arm():
10         # Begin flight!
11         pixels.fill(GREEN)
```

It Begins!

If the code reaches this line, it means `button_arm()` returned `True`.

- That means the user has armed AND confirmed *launch.*
- *(And hopefully they're standing clear!)*

```
12         fly.take_off(1.0)
13         pixels.fill(BLUE)
14         fly.steady(3.0)
15         pixels.fill(YELLOW)
16         fly.land()
17         pixels.off()
```

The Whole Shebang

This is the whole "hover" program.

- Notice the pixel LEDs are used to show which stage of flight your code is in.
- So just three `MotionCommander` commands:

```
fly.take_off(1.0)  # ascend to 1 meter altitude
fly.steady(3.0)    # hover for 3 seconds
fly.land()         # descend to the floor
```

```
18
```

Goals:

- Run the Hover code, calling `fly.take_off()` and `fly.land()`.
- Use your new `safety` module in an `import`.

Tools Found: Barometric Pressure Sensor, Laser Range Sensors, Optical Flow Sensor, Flight Module, MotionCommander Flight Interface, import, Parameters, Arguments, and Returns, RGB "pixel" LEDs

Solution:

```
from codeair import *
from flight import *
from safety import *

# Repeat the flight test
while True:

    # Safety-check button press
    if button_arm():
        # Begin flight!
        pixels.fill(GREEN)
        fly.take_off(1.0)
        pixels.fill(BLUE)
        fly.steady(3.0)
        pixels.fill(YELLOW)
        fly.land()
        pixels.off()

```

***Objective 3* - Moving Forward**

Moving Forward

You're flying now! Ready to build on what you've learned?

- Oh, but before you *move forward,* there's something you should know.
- To start with you'll only be using the **blocking** subset of MotionCommander.

Concept: *Blocking vs. Non-Blocking Functions*

What does "blocking" mean?

- Functions that block your code from continuing until they finish are called ***blocking*** functions.
- `sleep(seconds)` is a classic *blocking* function. Your code can't continue until it finishes.
- The MotionCommander functions you used for hovering are ***blocking***.

But there are also a set of ***non-blocking*** functions in MotionCommander.

- Say you want to start moving forward, and then while still moving continuously check some sensors, or blink LEDs, or play sounds, etc.
- The **non-blocking** functions *start* a movement, then return *immediately*.
  - You must then send *another* command to change or stop the movement!

More Motion

Check the 'Trek!

CodeTrek:

```
from codeair import *
from flight import *
from safety import *

# Repeat the flight test
while True:

    # Safety-check button press
    if button_arm():
        # Begin flight!
        pixels.fill(GREEN)
        fly.take_off(1.0)

        pixels.fill(BLUE)
        fly.forward(2.0)
```

Once you're airborne, try flying forward before landing.

- Experiment with different distances, but be sure you have enough room!
- *Remember, these distances are in **meters**.*

```

        pixels.fill(YELLOW)
        fly.land()

        pixels.off()

```

Goal:

- Add a call to `fly.forward(distance)` and run some test flights.

Tools Found: MotionCommander Flight Interface, Functions, Parameters, Arguments, and Returns

Solution:

```
from codeair import *
from flight import *
from safety import *

# Repeat the flight test
while True:

    # Safety-check button press
    if button_arm():
        # Begin flight!
        pixels.fill(GREEN)
        fly.take_off(1.0)

        pixels.fill(BLUE)
        fly.forward(2.0)

        pixels.fill(YELLOW)
```

```
18          fly.land()
19
20          pixels.off()
21
```

***Objective 4* - Quadcopter Sensors**

Quadcopter Sensors: *OODA!*

"Observe, Orient, Decide, Act!" - John Boyd, Colonel USAF

Take a look at CodeAIR.

No wings, no tail, this can't be stable!

- How can this thing fly?

Stable Quadcopter Flight Requires *Sensors!*

Sensors and electronics. *Autonomous* flight means those sensors have to connect to an onboard computer 🔧CPU.

- Like a well-trained fighter pilot, the computer runs a continuous OODA 🔧loop.

CodeAIR's 🔧laser rangers are a key component in keeping the drone flying at a desired *altitude*. The 🔧flight module lets you read data from all the flight control sensors, so accessing range data is as simple as:

```
get_data(RANGERS)  # returns (fwd, up, down) distance in mm
```

Create a New File!

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

Check the 'Trek!

Run It!

Run it, and 🔧print some 🔧laser ranger values to the *Console!*

Physical Interaction: *Hold Your Drone*

You'll need to point those rangers in different directions to achieve the Goals of this Objective!

- Watch the `(fwd, up, down)` values stream by on the **Console** while you handle CodeAIR.

CodeTrek:

```
1  from flight import *
```

For this test, only the flight module is needed.

```
2  while True:
3      r = get_data(RANGERS)
4      print(r)
```

A simple `while` loop.

- First save `get_data()`'s return value in a variable.
- Then use the print() statement to display it on the **Console**.

Don't worry, you'll get to explore "variables" more deeply soon!

Goals:

- Watch the **UP** ranger measure down to 0 and up to 1000 mm or more!
- Watch the **FWD** ranger measure down to 0 and up to 1000 mm or more!
- Watch the **DOWN** ranger measure down to 0 and up to 1000 mm or more!

Tools Found: CPU and Peripherals, Loops, Laser Range Sensors, Flight Module, API, Print Function, Variables

Solution:

```
1  from flight import *
2  while True:
3      r = get_data(RANGERS)
4      print(r)
```

Quiz 1 - CodeAIR sensors

Question 1: Which of the following are steps to using a custom module?

✓ Load the source code file on CodeAIR.

✓ Import the source code file.

✗ Define a function for the module.

✗ Read CodeAIR's sensors.

Question 2: Which of the following are features of a blocking function?

✓ Code execution is paused while the function runs.

✗ The function returns immediately after starting execution.

✓ Code doesn't continue until the function finishes.

✗ Another command is needed to stop the function.

Question 3: What sensor is used to keep the drone flying at a desired altitude?

✓ Laser rangers

- ❌ Pressure sensor
- ❌ Optical flow sensor
- ❌ Distance sensor

***Objective 5* - Back Off!**

Back Off!

Hands off my drone, dude!

- How would you like to put those 🔧laser rangers to good use?
- There's been talk of some shenanigans going on around here, but with technology and some clever coding you can *protect* CodeAIR from nefarious hands.

Defensive Lasers

Your challenge is to use your UP 🔧ranger to detect if someone gets too close to your drone.

- How close is too close? *That's for YOU to decide, and **code!***
- And I think *you* know what to do when an intruder is detected. *Shadow... and **flame!***
- Sensors? You're going to start with just the UP ranger, but you can always add more sensors to make the security even better!

Concept: *Variables*

The code below creates a 🔧variable named `r`, and sets it to whatever `get_data()` returns:

```
r = get_data(RANGERS)
```

In the last Objective, you used `print(r)` to show `r`'s value on the console.

Just a Name

A 🔧variable is a name that you attach to an object so your code can work with it.

- That object can be any data: like a *number*, *text*, or even a *3-🔧tuple* like the 🔧ranger data!

You'll need to understand variables to implement your *security* code. Read through the 🔧variables tool entry so you're up to speed!

Detecting an Intruder

Once you have the 🔧ranger data in a variable, how do you tell if someone is too close?

- First you need to define what "too close" means. You might make a 🔧variable named `too_close`.
- After that, you'll need to 🔧compare the distance measured by the `up` ranger with your `too_close` limit.

You've used `if` statements before, with `True`/`False` conditions like `buttons.was_pressed()`.

But how about:

```
if up < too_close:
    # Sound the alarm!
```

*Could your **security** 🔧comparison be that easy?*

Flight Controller 🔧Reboot

Did you notice when you printed the 🔧ranger values to the console, the first several readings were not valid?

```
(None, None, None)
...
(0, 0, 0)
```

```
...
(fwd, up, down)
```

Those `(None, None, None)` and `(0, 0, 0)` values from `get_data()` indicate that the flight controller has not yet finished its *startup initialization* after a reboot.

- When CodeAIR starts your Python code, it also reboots the flight controller!
- You'll need to deal with that "startup condition" so you don't trigger a false alarm when your program first starts running.

Check the 'Trek!

Adding to your *Rangers* file, now expand on that test loop.

- Oooh... Unpacking the ranger values is a cool trick.
- Be sure to make the *Alarm* unique - *get creative y'all!*

Physical Interaction: *Invisible Barrier*

Test the perimeter defenses, like a velociraptor in a dinosaur movie!

- *Hey, too bad they didn't have CodeAIR...*

CodeTrek:

```
"""Laser presence detection system"""
from codeair import *
from flight import *
from time import sleep
```

Add a docstring comment to remind you what this program does.

- And you will need the `codeair` module as well as `sleep()`

```

# How close is too close?  (millimeters)
too_close = 300
```

Create a new variable for the detection *distance*.

- Quick: *How many centimeters is that?*

```

def alarm():
    """Play one "cycle" of the alarm.
       Called repeatedly while presence is detected.
    """
    pixels.fill(MAGENTA)
    speaker.beep(1200,50)
    pixels.off()
```

define a function to sound/show the *alarm* condition.

- Define functions like this early in the file, so you can call them later on.
- Notice there's a `"""documentation string"""` here, like the one at the top of the file.

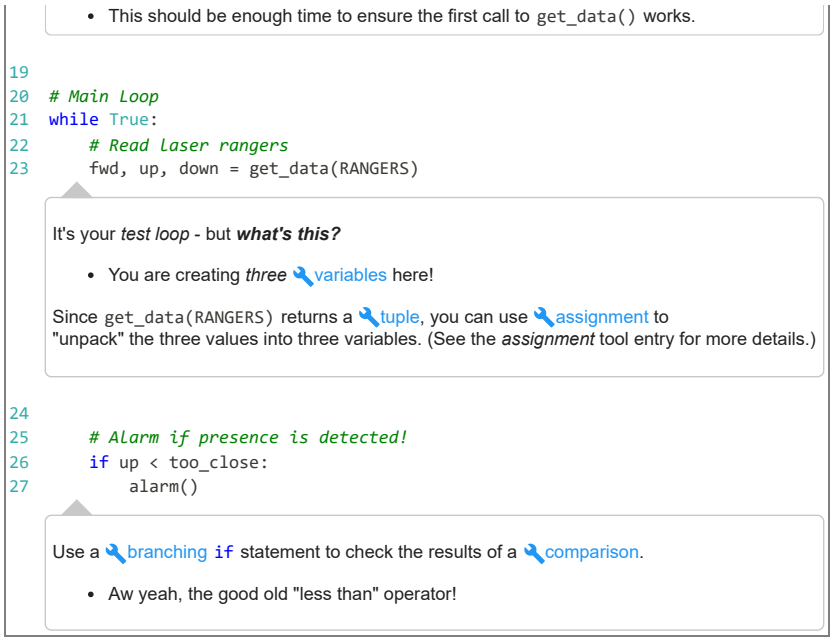
You can use functions to organize your code into meaningful bite-sized pieces.

- Each piece of code has a job to do!
- Each should be aptly named, commented, and easy to understand on its own.

```

# Wait for Flight Controller boot
sleep(3)
```

A simple solution to the Flight Controller reboot problem.



- This should be enough time to ensure the first call to `get_data()` works.

```
19
20  # Main Loop
21  while True:
22      # Read laser rangers
23      fwd, up, down = get_data(RANGERS)
```

It's your *test loop* - but ***what's this?***

- You are creating *three* 🔧variables here!

Since `get_data(RANGERS)` returns a 🔧tuple, you can use 🔧assignment to "unpack" the three values into three variables. (See the *assignment* tool entry for more details.)

```
24
25      # Alarm if presence is detected!
26      if up < too_close:
27          alarm()
```

Use a 🔧branching `if` statement to check the results of a 🔧comparison.

- Aw yeah, the good old "less than" operator!

Goals:

- Update your `while True:` 🔧loop to *unpack* the three 🔧variables `fwd`, `up`, and `down` from `get_data(RANGERS)`.
- Use 🔧branching and 🔧comparison with the `<` operator to decide when to sound the *alarm.*
- Define a function named `alarm()` which will be called when presence is detected.

Tools Found: Laser Range Sensors, Variables, tuple, Comparison Operators, Reboot, Loops, Assignment, Branching, Comments, Functions

Solution:

```
1  """Laser presence detection system"""
2  from codeair import *
3  from flight import *
4  from time import sleep
5
6  # How close is too close?  (millimeters)
7  too_close = 300
8
9  def alarm():
10     """Play one "cycle" of the alarm.
11        Called repeatedly while presence is detected.
12     """
13     pixels.fill(MAGENTA)
14     speaker.beep(1200,50)
15     pixels.off()
16
17 # Wait for Flight Controller boot
18 sleep(3)
19
20 # Main Loop
21 while True:
22     # Read laser rangers
23     fwd, up, down = get_data(RANGERS)
24
25     # Alarm if presence is detected!
26     if up < too_close:
27         alarm()
```

***Objective 6* - Flight Ceiling**

Flight Ceiling

Now how about using that **UP** 🔧ranger while in flight!

- Detecting if there's an obstacle (or ceiling) above the drone would be a nice feature.

Basic Detection

You already know how to detect if there's something above the drone.

- To start with, just add that capability to your Hover code.

Note:

Since your Hover code is using **blocking** functions for movement, you can't check the sensors while you're moving.

- For this Objective, keep it "basic". Check the sensor *after* you move.
- Don't worry, you will learn to use the **non-blocking** functions in a later Objective!

Concept: *Polling*

Remember that you can't use `sleep(seconds)` while flying, since the 🔧MotionCommander needs to continuously update the flight controller. Instead you use the `fly.steady(seconds)` to keep the drone steady while it's still actively flying. This function *blocks* other code from running, but it's also continuously ***polling*** the flight controller.

Polling simply means repeatedly checking something to see if anything has changed.

- Your code can use *polling* also!
- What if you 🔧loop : checking the 🔧laser ranger, and calling `fly.steady(0.1)` over and over?
- Then you can keep flying AND check the sensor 10 times per second!

Hand → *Land*

The goal of this Objective is for CodeAIR - after it reaches hover - to detect when you place your hand above it.

- CodeAIR should hover until it detects an object above at `up < too_close`.
- When it detects something above, it should descend to the ground with `fly.land()`.
- If no object is detected, it should descend after a 30 second timeout.

Create a New File!

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

Check the 'Trek!

Try Your Skills

Can you make the 🔧pixel LEDs turn a different color when landing due to sensors versus a timeout?

- Currently you aren't using the `True`/`False` 🔧return value from `poll_sensors()`. *That could be useful!*

CodeTrek:

```
1  """Hover until an obstacle above is detected"""
2  # TODO: Imports
```

Add the necessary `import` statements!

- Check your **Hover** program if you're not sure what you need here.

```
3
```

```
4  # How close is too close?  (millimeters)
5  too_close = 300
```

Just like *BackOff*, this is where you set the limit for detecting an obstacle.

```
6
7  def poll_sensors(timeout):
8      """Check sensors while flying steady. Return True if sensor event detected,
9         or False if timeout (seconds) elapsed with no event.
10     """
```

A new 🔧function.

- Like `fly.steady()`, this will keep flying until the *timeout* expires.
- But it *also* continuously checks the UP 🔧ranger!

```
11     ticks = timeout * 10
12     for i in range(ticks):
13         fly.steady(0.1)    # Wait 0.1 second "tick"
```

Basic Poll Loop

Every repeat of this 🔧loop is a *tenth-second* slice of time.

- That's why you multiply `timeout * 10` to know how many times to loop.

```
14
15         # Read laser rangers and check UP distance
16         fwd, up, down = get_data(RANGERS)
17         if up < too_close:
18             # Obstacle above detected!
19             return True
```

Ten times per second:

- Check the UP ranger.
- 🔧return immediately if an obstacle is detected!

```
20
21     # Timeout
22     return False
```

If you never detected an object...

- The *whole* `timeout` has elapsed.
- 🔧return `False` to signify that.

```
23
24
25 # Repeat the flight test
26 while True:
27
28     # Safety-check button press
29     if button_arm():
30         # Begin flight!
31         pixels.fill(GREEN)
32         fly.take_off(1.0)
33         pixels.fill(BLUE)
34
35         # Instead of blocking, "poll" with timeout
36         poll_sensors(30)
```

Almost Exactly the same as your **Hover** code!

- The only difference is to replace `fly.steady()` with a call to your new `poll_sensors()` 🔧function.

```
37
38          pixels.fill(YELLOW)
39          fly.land()
40          pixels.off()
41
```

Goals:

- Define a 🔧function called `def poll_sensors(timeout)` that checks the 🔧rangers 10 times per second.

  It should `return True` if an object is detected, or `return False` if the `timeout` expires with no object detected.

- Call your new `poll_sensors()` function to wait while *hovering*.

Tools Found: Laser Range Sensors, MotionCommander Flight Interface, Loops, RGB "pixel" LEDs, Parameters, Arguments, and Returns, Functions

Solution:

```
1  """Hover until an obstacle above is detected"""
2  from codeair import *
3  from flight import *
4  from safety import *
5
6  # How close is too close?  (millimeters)
7  too_close = 300
8
9  def poll_sensors(timeout):
10     """Check sensors while flying steady. Return True if sensor event detected,
11        or False if timeout (seconds) elapsed with no event.
12     """
13     ticks = timeout * 10
14     for i in range(ticks):
15         fly.steady(0.1)    # Wait 0.1 second "tick"
16
17         # Read laser rangers and check UP distance
18         fwd, up, down = get_data(RANGERS)
19         if up < too_close:
20             # Obstacle above detected!
21             return True
22
23     # Timeout
24     return False
25
26
27 # Repeat the flight test
28 while True:
29
30     # Safety-check button press
31     if button_arm():
32         # Begin flight!
33         pixels.fill(GREEN)
34         fly.take_off(1.0)
35         pixels.fill(BLUE)
36
37         # Instead of blocking, "poll" with timeout
38         poll_sensors(30)
39
40         pixels.fill(YELLOW)
41         fly.land()
42         pixels.off()
43
```

Objective 7 **- Theremin**

Good Vibrations

Ever heard of a Theremin?

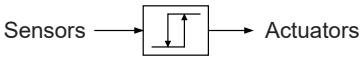
- Classically used by the Beach Boys and other bands experimenting with a haunting "spacy" sound.
- The original theremin, invented by scientist Léon Theremin in 1920, was played by waving your hands near radio antennas. *This was one of the first electronic musical instruments!*

Wave Your Hands Much?

I think I saw you recently waving your hands over CodeAIR! Since you can detect the distance pretty accurately, why not turn that into *music* by making your own **Theremin**?

An Engineering *Control Loop*

- You'll need to code a basic **control loop** - not that different from how CodeAIR's *flight controller* controls motors based on sensor inputs.



- CodeAIR has many *sensors,* as you've seen. For flying, the 🔧motors are the *actuators*.
- For your **Theremin** project, the *sensor* is the UP 🔧laser ranger, and the *actuator* is the 🔧speaker.

Concept: *Continuous Sound*

You've been playing "beeps" with the 🔧speaker, but what about continuous tones?

- Just use a duration of `0` and the tone will keep playing until you stop it!
- The `speaker.off()` function will stop the currently playing tone.

```
speaker.beep(440, 0)  # Start playing 440Hz tone
# do some other stuff...
speaker.off()  # Stop playing
```

Create a New File!

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

Check the 'Trek!

If you're up for it, give it a go yourself before you peek at the CodeTrek.

- After all, a certain amount of 🔧debugging is good for you!

Of course, The CodeTrek has your back if you need a little guidance.

Physical Interaction: *Make Some Music*

Okay, so maybe it's not *that* musical. But you get the idea!

- How does the sound change if you move your hand *very slowly?*
- Can you explain the "grainy" sound versus a "smooth ramp"?

CodeTrek:

```
"""Theremin... CodeAIRemin?"""
from codeair import *
from flight import *
from time import sleep
```

```
5
6  # Wait for flight controller boot
7  sleep(3)
8
9  # Stop sound if greater than this distance
10 ceiling = 1500
11
12 while True:
13     fwd, up, down = get_data(RANGERS)
14
15     if up < ceiling:
16         speaker.beep(400 + up, 0)
17     else:
18         speaker.off()
```

This whole program is similar to your "Back Off" code, but even simpler!

- Choose a `ceiling` value that's not too large, or else your Theremin will be constantly squealing.

```
19
```

Goal:

- Run the code, and *MAKE SOME NOISE* with your UP 🔧ranger.

Tools Found: Motors and Props, Laser Range Sensors, Speaker, Debugging

Solution:

```
"""Theremin... CodeAIRemin?"""
from codeair import *
from flight import *
from time import sleep

# Wait for flight controller boot
sleep(3)

# Stop sound if greater than this distance
ceiling = 1500

while True:
    fwd, up, down = get_data(RANGERS)

    if up < ceiling:
        speaker.beep(400 + up, 0)
    else:
        speaker.off()


```

Quiz 2 - Laser Rangers

Question 1: What line of code unpacks the data returned by the laser rangers?

✓ `fwd, up, down = get_data(RANGERS)`

✗ `get_data.unpack()`

✗ `rangers.data()`

✗ `rangers.read()`

Question 2: What is the result of the code?

```
too_close = 300
up = 250
```

```
if up < too_close:
    return True
```

✓ `True` is returned

✗ `False` is returned

✗ Nothing happens

✗ The program stops

Question 3: What line of code will play a beep continuously?

✓ `speaker.beep(440, 0)`

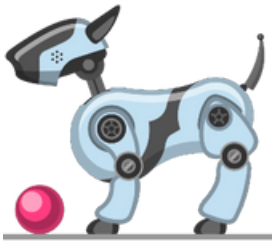
✗ `speaker.beep(440)`

✗ `speaker.beep(440,100)`

✗ `speaker.on(440)`

***Objective 8* - Hall Monitor**

Hall Monitor



This Objective is all about taking your *sensor* processing capabilities to the next level.

So far, you've been checking your 🔧ranger sensor value against a threshold:

```
if value < limit:
    # Do Something!
```

But what if the *instantaneous* sensor reading is not what you're looking for?

- Can your code *remember* stuff about this *sensor data?*
- A simple example would be to **count** how many times an event happened.

Concept: *Updating a Variable*

You can use 🔧variables to give your code *memory.*

```
count = 0  # Remember the count
...
# Detected an event!
count = count + 1  # Now count is 1
...
# Detected another event
count = count + 1  # Now count is 2
```

Notice how the above code adds 1 to `count` every time an event is detected?

- The `count` variable is first set to an 🔧integer value of `0`.
- Then it is used to calculate a new value, which becomes the new `count`.

See the 🔧assignment tool entry for more on this.

Create a New File!

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

People Counter

Your challenge is to use CodeAIR to create a "Hall Monitor" which counts how many people have passed by a particular point.

- Maybe you've seen "counters" like this used to monitor traffic, or secure building entrances.
- Use the FORWARD laser ranger this time.
  - `if fwd < detect_distance:`

Uh-oh. *Problem!?*

Imagine a loop where you keep increasing the `count` whenever `fwd < detect_distance` is `True`.

- What if someone just *stands* in front of the detector?
- The count would constantly increase!!
- One person could look like 10 people, *or more.*

Solution: *Define Your Event*

What exactly is the *event* that you are counting here?

- When one person walks by, there are *two changes* as they pass:
  1. `person_detected` : `False` ⟶ `True`
  2. `person_detected` : `True` ⟶ `False`

So how about just counting when `person_detected` goes from `False` to `True`.

- You'll need to track `person_detected` in another variable.
- Hey, that's *more memory* you're using! With `person_detected` CodeAIR is remembering that there wasn't a person there before, but now there is!

Check the 'Trek!

Your *detection loop* is gonna be sweet!

There's a new **Concept** hidden in the CodeTrek this time also.

- You are `not` going to want to miss it!

Physical Interaction: *Setting Up*

CodeTrek:

```
1  from codeair import *
2  from flight import *
3  # TODO: one more import needed here
```

There's a handy function you need to `import`, from a standard Python module.

- **Hint**: ..."wait" for it.

```
4
5  detect_distance = 1000  # millimeters
6  person_detected = False
7  count = 0
```

Defining and initializing some variables.

- In this program `detect_distance` doesn't change, so I should really call it a constant.
- But `person_detected` and `count` are your essential *memory* for this program!

```
8
9  # TODO: wait for flight controller to boot
```

How many seconds do you need to `sleep()`?

```
10
```

```
while True:
    fwd, up, down = get_data(RANGERS)
    if fwd < detect_distance:
        if not person_detected:
            person_detected = True
```

Concept: *Logical Operators*

```
            count = count + 1
            print("Count =", count)
            speaker.beep(700, 100)
```

A new person has been detected. The above four lines:

1. Remember it.
2. Increase the count.
3. ==print()== it to the **Console**
4. Beep!

```
    else:
        person_detected = False
```

And if the 🔧ranger doesn't see anything nearby?

- Make sure `person_detected` is set to `False`.

```

```

Goal:

- Run your code, and test it to at least a count of 1 0.
    - I'll be watching your 🔧print() statements on the **Console**.

Tools Found: Laser Range Sensors, Variables, int, Assignment, Loops, Print Function, Functions, import, Constants, Logical Operators, bool, undefined

Solution:

```
from codeair import *
from flight import *
from time import sleep

detect_distance = 1000
person_detected = False
count = 0

sleep(3)

while True:
    fwd, up, down = get_data(RANGERS)
    if fwd < detect_distance:
        if not person_detected:
            speaker.beep(700, 100)
            person_detected = True
            count = count + 1
            print("Count =", count)
    else:
        person_detected = False


```

***Objective 9* - Obstacle Detection**

Obstacle Detection and Avoidance

It's time to get your drone back in the air, and put some of your new *coding skills* to the test.

Types of Navigation

Autonomous vehicles can navigate without human control using various methods, each suited to different environments and tasks. These methods can broadly be classified into *Dead Reckoning* and *Sensor-based approaches.*

Dead Reckoning

This approach calculates the vehicle's position based on its known starting point, along with records of speed, direction, and elapsed time. Essentially, it "reckons" its location from its last known position, moving according to pre-defined distances and turns.

- While dead reckoning can be useful for short, controlled distances, it tends to accumulate errors over time. *(Very dependent on the accuracy of speed, direction, and time measurements!)*

Sensor-Based Navigation

This method relies on real-time data gathered by sensors (like 🔧rangers) to detect obstacles and adjust course accordingly. Sensor-based navigation is *adaptive,* allowing vehicles to respond dynamically to changes in the environment.

Your Challenge - *Escape Room*

Use *Sensor-Based Navigation* with CodeAIR!

- Explore an area *autonomously* by using the FORWARD 🔧laser ranger to detect walls or other objects.
- Each time an obstacle is detected, make a simple 90-degree left turn, and continue exploring the space!
- If no wall is encountered for 5 seconds, CodeAIR has "escaped" the room and can **land** *victoriously!*
- Keep a `count` of turns, and light up one of the blue 🔧leds corresponding to the `count`.

This Objective's code will be quite similar to your ***Ceiling*** code. That's a great starting point.

Open your *Ceiling* code file, then:

Save to a New File!

Use the **File** → **Save As** menu to create a new file called ***Avoidance***.

Check the 'Trek!

Be sure to start out using the recommended `altitude` and `too_close` distances.

- Notice the 🔧pixel LEDs are set to different colors based on the action!

⚠️ Be ready to "box-in" CodeAIR by placing obstacles in front of it! ⚠️

Physical Interaction: *Escape the Walls*

Give CodeAIR a chance to escape after ***FOUR*** turns, and verify it lands gracefully.

⚠️ Try more than SEVEN turns - *You'll discover a BUG!* ⚠️

Got Bugs?

You should still be able to escape the room, as long as there are fewer than 8 turns.

- Continue to the next Objective to fix that limitation!

CodeTrek:

```python
"""Obstacle avoidance"""
from codeair import *
from flight import *
from safety import *

too_close = 500  # Wall distance (millimeters)
altitude = 0.5  # meters (safe height in case of crash)
```

Set your distances.

- There *might* be a bug in this code that causes the drone to fall from the sky. Keep the `altitude` low, *just in case*...

```python

def poll_sensors(timeout):
    """Check sensors while flying steady. Return True if sensor event detected,
       or False if timeout (seconds) elapsed with no event.
    """
    ticks = timeout * 10
    for i in range(ticks):
        fly.steady(0.1)    # Wait 0.1 second "tick"
```

This is exactly how `poll_sensors()` worked in your **Ceiling** code.

- Checking sensors in $\frac{1}{10}$ second time slices.

```python

        # Read laser rangers and check FORWARD distance
        fwd, up, down = get_data(RANGERS)
        if fwd < too_close:
            # Obstacle in front detected!
            return True
```

Check the ***forward*** sensor this time.

```python

    # Timeout
    return False

# Repeat the flight test
while True:
    count = 0
    leds.set_mask(0, 0)
```

A shortcut to turn off ALL the blue 🔧LEDs.

- You'll learn more about this function in a later Mission.
- *Can't wait? Check the toolbox entry :-)*

```python

    # Safety-check button press
    if button_arm():
        # Begin flight!
        pixels.fill(GREEN)
        fly.take_off(altitude)
        pixels.fill(BLUE)

        # Loop: Fly forward and make left turns!
        while True:
            fly.start_forward()
```

A *non-blocking* 🔧MotionCommander function.

- This function starts moving forward at the default velocity (20 cm/s) and 🔧returns immediately!

```python
            # Instead of blocking, "poll" with timeout
            if poll_sensors(5):
```

```
43                # Detected a wall, get ready to turn
44                fly.stop()
45                pixels.fill(PINK)
```

Wait up to 5 seconds to detect a wall.

- IF the *forward* 🔧*ranger* sees something, stop and prepare to *turn...*

```
46
47                # Show count of turns on LEDs
48                count = count + 1
49                leds.set(count, 50)
```

Increment the `count` 🔧variable, and set the corresponding blue 🔧LED to show the user how many turns the drone has made.

```
50
51                # Blocking turn
52                fly.turn_left(90)
53                pixels.fill(BLUE)
```

Use the *blocking* 🔧MotionCommander function `fly.turn_left(90)` to make a 90° left turn.

```
54            else:
55                # Escaped!
56                pixels.fill(MAGENTA)
57                break
```

`else`: the `poll_sensors(5)` has timed out!

- Flying this far with no wall = *ESCAPED!*

```
58
59        fly.land()
60        pixels.off()
61
```

Goals:

- Use the *non-blocking* `fly.start_forward()` function to move while checking the FORWARD 🔧ranger.
- Set the blue 🔧LED to show the `count` of left turns.
- Use the *blocking* `fly.turn_left(90)` function to turn left 90° when a wall is detected.

Tools Found: Laser Range Sensors, BYTE LEDs, RGB "pixel" LEDs, Debugging, MotionCommander Flight Interface, Parameters, Arguments, and Returns, Variables

Solution:

```
1  """Obstacle avoidance"""
2  from codeair import *
3  from flight import *
4  from safety import *
5
6  too_close = 500  # Wall distance (millimeters)
7  altitude = 0.5  # meters (safe height in case of crash)
8
9  def poll_sensors(timeout):
10     """Check sensors while flying steady. Return True if sensor event detected,
11        or False if timeout (seconds) elapsed with no event.
12     """
13     ticks = timeout * 10
14     for i in range(ticks):
```

```
        fly.steady(0.1)    # Wait 0.1 second "tick"

        # Read Laser rangers and check FORWARD distance
        fwd, up, down = get_data(RANGERS)
        if fwd < too_close:
            # Obstacle in front detected!
            return True

    # Timeout
    return False

# Repeat the flight test
while True:
    count = 0
    leds.set_mask(0, 0)

    # Safety-check button press
    if button_arm():
        # Begin flight!
        pixels.fill(GREEN)
        fly.take_off(altitude)
        pixels.fill(BLUE)

        # Loop: Fly forward and make left turns!
        while True:
            fly.start_forward()
            # Instead of blocking, "poll" with timeout
            if poll_sensors(5):
                # Detected a wall, get ready to turn
                fly.stop()
                pixels.fill(PINK)

                # Show count of turns on LEDs
                count = count + 1
                leds.set(count, 50)

                # Blocking turn
                fly.turn_left(90)
                pixels.fill(BLUE)
            else:
                # Escaped!
                pixels.fill(MAGENTA)
                break

        fly.land()
        pixels.off()

```

Quiz 3 **- Detection**

Question 1: What function is non-blocking?

✓ `fly.start_forward()`

✗ `fly.steady(seconds)`

✗ `fly.take_off(altitude)`

✗ `fly.turn_left(degrees)`

Question 2: What will print after the code runs?

```
my_var = True
my_var = not my_var
print(my_var)
```

✓ `False`

✗ `True`

❌ my_var

❌ An error occurs

Question 3: What is the result of the code?

```
count = 7
count = count + 1
if count == 8:
    pixels.fill(WHITE)
```

✔ All pixels turn WHITE

❌ All pixels turn off

❌ Nothing happens

❌ The program stops

Question 4: What function turns off all blue LEDs?

✔ `leds.set_mask(0, 0)`

❌ `leds.set_off()`

❌ `leds.set(0)`

❌ `leds.set(BLACK)`

***Objective 10* - Escape Bug**

Wipeout!

If you're not crashing, you're not coding!

What's Up?

Seriously? One little bug and CodeAIR drops like a rock!?

- Yeah, pretty much. There are some major real-world disasters linked to software bugs. *So it's not just you!*
- Oh yeah, and in this case, I set you up ;-)

LED Mischief

The blue 🔧LEDs are the culprit here. What happens when you try to light an LED that doesn't exist?

```
leds.set(8, 50)  # There's no 8th LED...
```

Try it on the ≡ REPL

So far you have used the ≡ **Console** to *output* messages using the 🔧print function.

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

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

Your CodeAIR must be connected and ■ stopped so you can interact with it on the 🔧REPL.

- Open the ≡ **Console**, *click* in the console window and type:

```
from codeair import *
leds.set(0, 50)
```

You should see blue LED 0 light up!

Now try:

```
leds.set(8, 50)
```

Whoa! This throws an 🔧Exception!

So, if your PC had been connected when CodeAIR made that 8th turn, you'd have seen this on the Console:

```
'ValueError: LED num must have a value between 0 and 7'
```

What's the Fix?

There are a few directions you could take here. How do YOU want it to work? Some ideas:

1. You could reset the count back to zero every time it reaches 8.
2. Or you might just say CodeAIR only gets 7 attempts to escape, and change the code to make a graceful landing on the 8th turn.
3. OR you could leave it as-is and say you only get 7 chances before your bot is caught trying to escape and *knocked out* by the "guards!"

Oooh, I like that last one! *"Call it a feature"* is a time-honored strategy among software engineers when a bug is discovered ;-)

Check the 'Trek!: *Bugfix*

I'm not going to let you use the *"It's a feature!"* strategy... *this* time.

The CodeTrek will lead you to the "Graceful Surrender" solution, where CodeAIR lands peacefully after 7 attempts.

Be sure to test your change - you'll need to make a few more turns to be sure!

CodeTrek:

```
"""Obstacle avoidance"""
from codeair import *
from flight import *
from safety import *

too_close = 500  # Wall distance (millimeters)
altitude = 0.5  # meters (safe height in case of crash)

def poll_sensors(timeout):
    """Check sensors while flying steady. Return True if sensor event detected,
       or False if timeout (seconds) elapsed with no event.
    """
    ticks = timeout * 10
    for i in range(ticks):
        fly.steady(0.1)    # Wait 0.1 second "tick"

        # Read laser rangers and check FORWARD distance
        fwd, up, down = get_data(RANGERS)
        if fwd < too_close:
            # Obstacle in front detected!
            return True

    # Timeout
    return False

# Repeat the flight test
while True:
    count = 0
    leds.set_mask(0, 0)

    # Safety-check button press
    if button_arm():
        # Begin flight!
        pixels.fill(GREEN)
```

```
        fly.take_off(altitude)
        pixels.fill(BLUE)

        # Loop: Fly forward and make left turns!
        while True:
            fly.start_forward()
            # Instead of blocking, "poll" with timeout
            if poll_sensors(5):
                # Detected a wall, get ready to turn
                fly.stop()
                pixels.fill(PINK)

                # Show count of turns on LEDs
                count = count + 1
                if count == 8:
                    # Yikes! Limit reached - land now.
                    pixels.fill(WHITE)
                    break
```

Fix the bug by making a new rule:

- **You only get 7 attempts (turns) to make your escape!**

On the 8th attempt, show the white flag of surrender and land peacefully.

```
                leds.set(count, 50)

                # Blocking turn
                fly.turn_left(90)
                pixels.fill(BLUE)
            else:
                # Escaped!
                pixels.fill(MAGENTA)
                break

        fly.land()
        pixels.off()

```

Goal:

- Fix the bug, by checking `if count == 8:` and taking an alternative action.
    - I'm looking for that exact `if` statement in your code.

Tools Found: BYTE LEDs, Print Function, REPL, import, API, Exception

Solution:

```
"""Obstacle avoidance"""
from codeair import *
from flight import *
from safety import *

too_close = 500  # Wall distance (millimeters)
altitude = 0.5  # meters (safe height in case of crash)

def poll_sensors(timeout):
    """Check sensors while flying steady. Return True if sensor event detected,
       or False if timeout (seconds) elapsed with no event.
    """
    ticks = timeout * 10
    for i in range(ticks):
        fly.steady(0.1)    # Wait 0.1 second "tick"

        # Read laser rangers and check FORWARD distance
        fwd, up, down = get_data(RANGERS)
        if fwd < too_close:
            # Obstacle in front detected!
            return True
```

```
22
23      # Timeout
24      return False
25
26  # Repeat the flight test
27  while True:
28      count = 0
29      leds.set_mask(0, 0)
30
31      # Safety-check button press
32      if button_arm():
33          # Begin flight!
34          pixels.fill(GREEN)
35          fly.take_off(altitude)
36          pixels.fill(BLUE)
37
38          # Loop: Fly forward and make left turns!
39          while True:
40              fly.start_forward()
41              # Instead of blocking, "poll" with timeout
42              if poll_sensors(5):
43                  # Detected a wall, get ready to turn
44                  fly.stop()
45                  pixels.fill(PINK)
46
47                  # Show count of turns on LEDs
48                  count = count + 1
49                  if count == 8:
50                      # Yikes! Limit reached - land now.
51                      pixels.fill(WHITE)
52                      break
53                  leds.set(count, 50)
54
55                  # Blocking turn
56                  fly.turn_left(90)
57                  pixels.fill(BLUE)
58              else:
59                  # Escaped!
60                  pixels.fill(MAGENTA)
61                  break
62
63          fly.land()
64          pixels.off()
65
```

Mission 5 Complete

Spectacular Soaring!

You achieved a LOT of Python learning in this Mission. *That's what it takes, if you truly want to fly autonomously!*

Algorithms, Sensors, Navigation?

Yes, if you're talking about something that flies on its own, it's gonna need all that and more.

- CodeAIR has plenty more capabilities to discover.
- ...and you're going to learn how to master all of them, right?

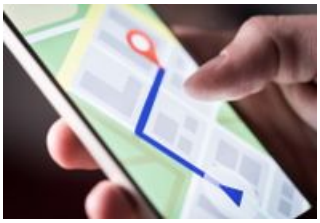
Try Your Skills: *Remix!*

Take some time to experiment with what you've learned so far.

- You have some powerful stuff in your **toolbox** - try it!

Mission 6 **- Navigate**

Navigate!



In this Mission, you'll guide CodeAIR as it explores an indoor environment using its 🔧Flow sensor for navigation. This mission introduces concepts of position tracking, sensor limitations, and selectable operations to control various flight parameters, giving you hands-on experience in navigating autonomously and adapting to real-world challenges!

Mission Targets

Get ready to:

- Explore *Positioning Systems* with the 🔧Flow sensor for x, y tracking.
- Observe and analyze *flow sensor accuracy* by flying CodeAIR in a square.
- Conduct a *Battery Check* to ensure safe and sustained flight.
- Customize *Selectable Operations* to control your code's behavior at runtime.
- Experiment with *Flight Parameters* including height, distance, and velocity.

Objective 1 **- Positioning with Flow**

Go with the *Flow!*

Positioning Systems Review

You've learned about the importance of *Positioning Systems* to autonomous vehicles like CodeAIR. After all, how can it explore an area if it doesn't know its own location?

- With the 🔧rangers you experienced the accuracy of *Sensor-Based Navigation.*
- And you learned that the alternative *Dead Reckoning* technique relies on measured *speed, direction, and elapsed time.*

CodeAIR's Speedometer?

You're going to be doing quite a lot of *Dead Reckoning* with CodeAIR. So how can it measure speed and direction? A car measures how fast it's going by sensing the rotation of the wheels. And it's pretty simple to calculate the distance traveled (odometer) based on wheel rotation *(as easy as Pi!)*

- But it would be pretty awkward for a drone to have to lower a *measurement wheel* to the ground to figure out how far and fast it's going!
- However, "**looking** at the ground" *is* a pretty good strategy for measuring movement!

Concept: *Flow Sensor*

Measuring X, Y Movement

The 🔧Flow sensor is an optical device with a lens pointed at the ground. It's like a very low-resolution camera that can detect just a few 🔧pixels.

- Think of it like a grid with (X,Y) coordinates projected on the ground!
- With *good lighting* the 🔧Flow sensor can discern *patterns* on the ground and report movement in the X and Y directions.

What about Z?

CodeAIR moves in 3-dimensions, and the height or *altitude* dimension is along the **Z-axis**.

- The X direction is *FORWARD* for CodeAIR.
- As you've learned, the Down-facing 🔧laser ranger gives a very accurate height (Z) measurement.

Delta Force

You'll use the flight module to read changes in position reported by the Flow sensor. The sensor reports *changes* in position or "deltas". (From the Greek letter Δ uppercase delta, used in math and science to represent change.)

```
# Get flow "deltas"
dx,dy = get_data(FLOW)
```

If you print() these deltas by themselves they're not all that helpful. But if you ***sum up the changes*** over time you can really see how the sensor tracks CodeAIR's motion.

Pretty Printing

When you `print()` values to the console, the simplest thing would be something like `print(x, y)`. The `print()` statement can handle multiple variables, so you'd get output like:

```
0 3
-3 0
-2 2
```

But printing numbers with just a space between them is not very user-friendly! *Wouldn't it be better if the output was:*

```
Flow Sensor Output:
x=0, y=3
x=-3, y=0
x=-2, y=2
```

You can print a text message to the console by enclosing it with quotation marks to make a string, like `print("Flow Sensor Output")`

Concept: *Format Strings*

To print an integer (or other data type) Python first has to convert it to a string.

- A string is just a sequence of characters all *strung* together. Numbers, letters, spaces, whatever!
- The `print()` statement does the conversion automatically, but if you want more control you can use a format string.

Ex: *The following prints `12345` to the Console.*

```
x_value = 12345
print(f"x = {x_value}")
```

Notice the string has an `f` just before the first quotation mark. That's an f-string, which allows you to embed expressions using braces!

Create a New File!

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

Check the 'Trek!

The Flow sensor might sometimes report larger-than-expected values due to rapid movements or drift. To address this:

- Ignore values over 50, as these will introduce noise in the data.
- Since the values are positive AND negative, you also want to throw away large *negative* numbers.

In math you may have learned about the *absolute value* of a number.

$$|\text{value}| > 50$$

Naturally Python has a built-in function for this. Check it out in the CodeTrek!

Run It!

Try holding CodeAIR just above a surface like a desktop, keyboard, notebook paper, etc.

Watch the ☰ Console as you:

- Move it in the **X** direction across the surface.
- Move it in the **Y** direction across the surface.

Are X and Y increasing and decreasing as shown in the diagram here?

- *Increasing* in the direction of the **arrow**, *decreasing* when you move the opposite way!
- Use the **BTN_0** reset function to refresh data easily for repeated tests.

CodeTrek:

```
1  from flight import *
2  from codeair import *
3  from time import sleep
4
5  # Cumulative X and Y distances traveled
6  x = 0
7  y = 0
```

Start by initializing two 🔧variables which will hold your **sum** of delta measurements.
This is the *accumulated distance* traveled.

```
8
9  # Wait for flight controller boot
10 sleep(3)
11
12 while True:
13     # Unpack the flow deltas
14     dx,dy = get_data(FLOW)
```

At the beginning of your *infinite loop* **read** the 🔧Flow sensor.

- This is your *delta X* and *delta Y* measurement.

```
15
16     # Discard values exceeding +/-50
17     if abs(dx) > 50:
18         dx = 0
19     if abs(dy) > 50:
20         dy = 0
```

Throw away the outliers!

- Notice the `abs()` *absolute value* function.
- It's a Python 🔧built-in that always gives back a positive number!

```
21
22     # Sum the deltas
23     x = x + dx
24     y = y + dy
```

The familiar "update a variable" pattern.

- Remember, *right-hand side* is evaluated first...
- Then the result is 🔧assigned to the 🔧variable on the left-hand side.

```
25
26     # Pretty print!
27     print(f"x={x}, y={y}")
```

🔧f-string!

- Okay, settle down now ;-)
- A format string lets you put 🔧variables like `{x}` and `{y}` right inside the message!

```
28
29      # Reset distances if BTN_0 was pressed
30      if buttons.was_pressed(BTN_0):
31          x = y = 0
32
```

A handy *reset* feature.

- Clear the accumulated `x` and `y` values back to zero.
- Oooh, *cascaded* 🔧assignment. Remember, evaluate from *right to left*.

Goals:

- 🔧Loop while reading the 🔧Flow sensor and `print()`ing the X, Y values.
  - I'll be looking for `x=...` and `y=...` on the 🔧console!
- Move CodeAIR in the X-axis to measure from -50 to +50 units.
- Move CodeAIR in the Y-axis to measure from -50 to +50 units.

Tools Found: Laser Range Sensors, Optical Flow Sensor, Pixel, Flight Module, Print Function, str, int, Data Types, Character Encoding, String Formatting, Built-In Functions, Loops, Variables, Assignment

Solution:

```
1  from flight import *
2  from codeair import *
3  from time import sleep
4
5  # Cumulative X and Y distances traveled
6  x = 0
7  y = 0
8
9  # Wait for flight controller boot
10 sleep(3)
11
12 while True:
13     # Unpack the flow deltas
14     dx,dy = get_data(FLOW)
15
16     # Discard values exceeding +/-50
17     if abs(dx) > 50:
18         dx = 0
19     if abs(dy) > 50:
20         dy = 0
21
22     # Sum the deltas
23     x = x + dx
24     y = y + dy
25
26     # Pretty print!
27     print(f"x={x}, y={y}")
28
29     # Reset distances if BTN_0 was pressed
30     if buttons.was_pressed(BTN_0):
31         x = y = 0
32
```

***Objective 2* - Square Up!**

Navigating a Pattern

Now that you're familiar with the 🔧flow sensor it's time to use it for flight navigation.

- Your first challenge will be to fly in a ***square***.

Sounds easy, right?

Positioning System

Actually, the MotionCommander API *does* provide some easy-to-use functions for navigating forward, back, left, right, etc.

- These functions use the flow sensor, and the flight controller takes care of the low-level work converting dX and dY values to an approximate X, Y position for CodeAIR.

Concept: *Sensor Fusion*

Sensor Fusion

One challenge the **flight controller** deals with is how *altitude* affects the flow values.

- To understand this, make a small circle with your hand as shown here.
- Imagine this is the *pixel size* of the flow sensor - it's your "pixel window".
- Look down at the floor, through your "pixel window". *What happens when you move closer?*
- Of course, closer to the floor means *less area* covered by the pixel.
- Moving at a constant horizontal speed, more stuff will pass beneath the pixel when you're high up. So unless the *altitude* is accounted for, the flow sensor will indicate a faster speed when CodeAIR is up high, and slower speeds when it's down low.
- The flight controller continuously checks the down-facing laser ranger so it can factor altitude into the position calculations! *When data from multiple sensors is combined like this it's called **sensor fusion** - ooooh, fancy!*

Sensor Drift and Accuracy

When you test your code, you'll notice the positioning is not always precise!

- A well-lit, visual pattern on the floor is very helpful for the flow sensor.
- But with *dead reckoning* like this, even small errors in position **accumulate** over time.
- That makes it quite challenging for CodeAIR to fly a square pattern and land on exactly the same spot it started from!

Create a New File!

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

Check the 'Trek!

This code should be pretty familiar to you by now!

- Check out the comments - and maybe add a few of your own.
- This is *your* code to hack as you wish, after all!

Run It!

Try a few runs, and see how *square* your square can be.

- Experiment with the distances of the sides (make sure you have ample space!)
- Try higher and lower altitudes, and see how that affects the navigation.
- How about the floor surface? Test with different conditions.

CodeTrek:

```
1  from flight import *
2  from safety import *
3  from codeair import *
4
5  altitude = 0.5  # meters
6  side_distance = 1.0  # meters
```

Define some variables you might want to experiment with later.

- Better to have names for these things, rather than *magic numbers* down in the code.
- You know, readability and all that!

```
7
8  def wait(seconds):
9      """Fly steady, with LED indication"""
10     pixels.fill(BLUE)
11     fly.steady(seconds)
12     pixels.off()
```

A helpful function that will give a visual "pause" in flight at the corners of your square.

- Or actually, anywhere you decide to call this function from.
- Now with just one line of code you can do a "flash-hover!"

```
13
14 # Repeat the flight test
15 while True:
16     if button_arm():
17
18         fly.take_off(altitude)
19         wait(1)
```

Take off!

And once you're up there... *"Flash-Hover!"*

```
20
21         # Fly in a square!
22         fly.forward(side_distance)
23         wait(1)
24         fly.left(side_distance)
25         wait(1)
26         fly.back(side_distance)
27         wait(1)
28         fly.right(side_distance)
29         wait(1)
```

A complete square, facing forward the whole time.

- Check out MotionCommander for more details on the `fly` commands.

```
30
31         # Back to terra firma.
32         fly.land()
```

Don't forget to land.

- And then your program loops back to await another button press!

Goals:

- Define a 🔧function `wait(seconds)` that shows BLUE on the `CodeAIR::pixel LEDs` while flying steady.
- Fly in a SQUARE by using the 🔧MotionCommander functions:
  - `fly.forward()`
  - `fly.left()`
  - `fly.back()`
  - `fly.right()`

Tools Found: Optical Flow Sensor, MotionCommander Flight Interface, Laser Range Sensors, Comments, Functions, Variables, Readability, Loops

Solution:

```python
from flight import *
from safety import *
from codeair import *

altitude = 0.5  # meters
side_distance = 1.0  # meters

def wait(seconds):
    """Fly steady, with LED indication"""
    pixels.fill(BLUE)
    fly.steady(seconds)
    pixels.off()

# Repeat the flight test
while True:
    if button_arm():

        fly.take_off(altitude)
        wait(1)

        # Fly in a square!
        fly.forward(side_distance)
        wait(1)
        fly.left(side_distance)
        wait(1)
        fly.back(side_distance)
        wait(1)
        fly.right(side_distance)
        wait(1)

        # Back to terra firma.
        fly.land()
```

Quiz 1 - Nav Basics

Question 1: If the 🔧flow sensor reports that X is *increasing*, which direction is CodeAIR moving?

✓ Forward

✗ Backward

✗ Right

✗ Left

Question 2: What is printed by the following?

```
x = 3
dx = 7
x = x + dx

print(f"X = {x}")
```

✓ X = 10

✗ 10

✗ X = 7

✗ 3

Question 3: What's the value of `dy` after the following code runs?

```
dy = -27
if abs(dy) > 20:
    dy = 0
```

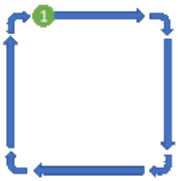
✓ 0

✗ 27

✗ -27

***Objective 3* - Rotate**

Rotation Challenge



Ready for a curve ball? Let's dive deeper into the intricacies of CodeAIR's navigation system by exploring how it behaves when we introduce rotation into the movements.

- How do you think the **flow sensor** will handle it if CodeAIR rotates while navigating?
- Imagine what the sensor will "see" while spinning — patterns of the ground below will swirl across its field of view.

The Swirly Lollipop Effect

To help you picture this, think of the flow sensor as "looking" at the ground through a swirling lens. When CodeAIR rotates, the surface patterns below will appear to spin in the opposite direction of the rotation. This creates a kind of "swirly lollipop" effect in the flow sensor's perspective.

What Happens During Rotation?

Unlike simple forward or side-to-side motion, rotation presents a unique challenge for processing data from the flow sensor. The flight controller is trying to detect and calculate movement based on changes in ground patterns. However, when CodeAIR rotates:

1. The patterns move in curved trajectories rather than linear ones.
2. The flight controller algorithms may interpret these curved patterns as unpredictable motion, leading to drift or inaccuracies in positioning.

Save to a New File!

Use the **File** → **Save As** menu to create a new file called ***SquareTurns***.

Check the 'Trek!

Change your code to implement the "always move forward" approach! *(last time it was always FACE forward)*

- The `wait()` function helpfully shows when each movement is happening.
- This version uses a 🔧loop to traverse the `range(4)` sides of the square!

⚠️ Make sure you allow plenty of space for CodeAIR to drift through the corners! ⚠️

Physical Interaction: *Test Cornering*

Run a few tests with this approach to flying the square pattern. *Pay close attention to what's happening!*

- Is the rotation itself accurate? That is, are the turns 90° as expected?
- Does the 🔧flow sensor *lose traction* on the turns?
- Try setting a different `altitude` - and maybe different lighting and floor patterns too.

CodeTrek:

```
from flight import *
from safety import *
from codeair import *

altitude = 0.5  # meters
side_distance = 1.0  # meters

def wait(seconds):
    """Fly steady, with LED indication"""
    pixels.fill(BLUE)
    fly.steady(seconds)
    pixels.off()

# Repeat the flight test
while True:
    if button_arm():

        fly.take_off(altitude)
        wait(1)

        # Fly in a square, with rotations!
        for i in range(4):
            fly.forward(side_distance)
            wait(1)
            fly.turn_left(90)
            wait(1)
```

Replace Lines with a 🔧Loop

A square has 4 sides, so your loop will be `range(4)`.

- See the 🔧range toolbox entry for more on that.
- Be sure to `wait()`
  - After every turn!
  - After every forward move!

```

        # Back to terra firma.
        fly.land()
```

Goal:

- Replace the *square sequence* commands with strictly forward movement and 90° left turns.

Tools Found: Loops, Optical Flow Sensor, Ranges

Solution:

```
from flight import *
from safety import *
from codeair import *

```

```
altitude = 0.5  # meters
side_distance = 1.0  # meters

def wait(seconds):
    """Fly steady, with LED indication"""
    pixels.fill(BLUE)
    fly.steady(seconds)
    pixels.off()

# Repeat the flight test
while True:
    if button_arm():

        fly.take_off(altitude)
        wait(1)

        # Fly in a square, with rotations!
        for i in range(4):
            fly.forward(side_distance)
            wait(1)
            fly.turn_left(90)
            wait(1)

        # Back to terra firma.
        fly.land()
```

***Objective 4* - Battery Check**

Battery Check!

How's your battery charge level?

- Hmmm... How would you know?

When CodeAIR is plugged into USB the battery is constantly being charged.

- The USB port can power everything on CodeAIR *except* the motors.
- So, while you're plugged in modifying code, the battery is getting filled up!
- If you keep your CodeAIR plugged in while you're working on the code, as you make brief test flights you'll always have plenty of charge.
- But if you make really long flights, or do lots of flight testing without much time being plugged into USB, your battery level will get low.
- Starting from empty it can take around an hour to fully charge up.

Checking the Charge Level

Naturally you don't want the battery to die in the middle of a flight! So checking the charge level is a pretty important feature.

- Fortunately, CodeAIR can measure its own battery voltage.
- You just need a little Python code to check the voltage and indicate status to the user!

Concept: *CodeAIR Power Monitoring*

With `from codex import *` you get access to the `power` object.

- That object provides some nice functions your Python code can use to determine what's going on with CodeAIR's power supply.

```
volts = power.battery_voltage(10)  # read batt voltage, average 10 samples
amps = power.charger_current()     # read charging current
usb_connected = power.is_usb()     # True if currently powered by USB
```

Notes:

1. When the USB is plugged-in you will see the *charging voltage*. This will be a pulsed voltage that's *higher* than the battery voltage when unplugged. ***Battery level can only be assessed when you're NOT plugged into USB.***

2. The battery voltage will drop considerably when it's *under load.* ***Testing with the motors powered is the best way to know the true battery level.***

Power UP!

Create a New File!

Run It!

If you run the code as-is, your CodeAIR will take off, hover for 20 seconds, and then land.

- While it's running you'll see the pixel LEDs flash periodically:
  - GREEN = HIGH
  - YELLOW = MEDIUM
  - ORANGE = LOW
  - RED = VERY LOW

You will see GREEN flashes if your battery is full. If you like, you can change the expected voltage levels to test color changes, OR just extend the test run so you can watch the colors change as the battery slowly discharges.

Wait!

Run It!

CodeTrek:

```
1  from codeair import *
2  from flight import *
3  from safety import *
```

The usual imports

```
4
5  #--- The following two functions should be copied into the safety.py program ---
6  def check_batt():
7      """Call this while hovering, will light LEDs and return True if batt okay to fly"""
8      vbatt = power.battery_voltage(10)
```

Feel the `power`!!

Measure the battery voltage, using 10 samples to get a quick but accurate measurement.

```
9      if vbatt > 3.9:
10         pixels.fill(GREEN)
11     elif vbatt > 3.6:
12         pixels.fill(YELLOW)
13     elif vbatt > 3.3:
14         pixels.fill(ORANGE)
15     else:
16         pixels.fill(RED)
17         return False
18
19     return True
```

This function does TWO things

1. Light all the pixel LEDs based on the battery level, AND leave them ON!
2. Return `True` if CodeAIR is good to fly, or `False` if there's not enough battery left.

```
20
21 def batt_check_steady(seconds=1.0):
```

```
    """While flying, check batt and show LEDs for 1s. Hold steady longer if needed."""
    if check_batt():
        fly.steady(0.5)  # Flash briefly
        pixels.off()
        fly.steady(max(0.5, seconds - 0.5))
        return True
    else:
        # Leave lights on and land.
        fly.land()
        return False
```

The "Flying Battery Check"

Okay, remember you can't `sleep()` while flying. So how will you *time* the LED flashing?

- This function uses `fly.steady()` for timing, to flash the LEDs ON and OFF in 1 second.
- You can also (🔧optional argument) pass it a total steady-time in `seconds` to `fly.steady()` but it's gonna take a minimum of 1 second to flash ON and OFF.

If the battery is *very low* this function will land immediately!

- In that case it will return `False`.

```


#-------------------------------------------------------------------------
# Test Program - do NOT put this code in safety.py
if button_arm():
    fly.take_off(0.5)

    # Loop for a while, testing battery while hovering
    for i in range(10):

        if not batt_check_steady(2.0):
            break

    # Land when loop ends
    fly.land()
```

Test Program

The test program uses the `button_arm()` function from `safety.py` to wait for a user to arm the drone.

- Then it takes off and hovers.
- While hovering, it repeatedly calls `batt_check_steady()`.
- Each call takes `2.0` seconds. That includes blinking the LEDs!

So you should see CodeAIR blinking once every two seconds while hovering.

- The total hover time is determined by the `for` 🔧loop.
- For example, 10 loop iterations would take 20 seconds...

Goal:

- Define 🔧functions `check_batt()` and `batt_check_steady()`, and run your test program that hovers while calling `batt_check_steady()`.

Tools Found: RGB "pixel" LEDs, Functions, import, Default function parameters, Loops

Solution:

```
from codeair import *
from flight import *
from safety import *

#--- The following two functions should be copied into the safety.py program ---
def check_batt():
    """Call this while hovering, will light LEDs and return True if batt okay to fly"""
```

```
    vbatt = power.battery_voltage(10)
    if vbatt > 3.9:
        pixels.fill(GREEN)
    elif vbatt > 3.6:
        pixels.fill(YELLOW)
    elif vbatt > 3.3:
        pixels.fill(ORANGE)
    else:
        pixels.fill(RED)
        return False

    return True

def batt_check_steady(seconds=1.0):
    """While flying, check batt and show LEDs for 1s. Hold steady longer if needed."""
    if check_batt():
        fly.steady(0.5)  # Flash briefly
        pixels.off()
        fly.steady(max(0.5, seconds - 0.5))
        return True
    else:
        # Leave lights on and land.
        fly.land()
        return False

#--------------------------------------------------------------------------------
# Test Program - do NOT put this code in safety.py
if button_arm():
    fly.take_off(0.5)

    # Loop for a while, testing battery while hovering
    for i in range(10):

        if not batt_check_steady(2.0):
            break

    # Land when loop ends
    fly.land()


```

Quiz 2 - Knowledge is Power

Question 1: Why does rotation cause the 🔧flow sensor readings to drift?

✓ It produces curved 🔧pixel trajectories which are not properly interpreted by the flight controller.

✗ Rotation causes 🔧motor oscillations, which perturb the flight dynamics.

✗ It doesn't. Rotation has no effect on the flight controller's processing of 🔧flow sensor readings.

Question 2: What is the purpose of the `battery_check_steady(seconds)` function?

✓ To test the battery and provide a visual indication while hovering.

✗ To make sure the battery level does not change for a specified period `seconds`.

✗ To confirm the battery is firmly attached to CodeAIR.

Question 3: How long does the following 🔧function take to run?

- Assume a fully charged battery.

```
isOkay = battery_check_steady(0.7)
```

✓ 1.0 sec

✗ 0.7 sec

❌ 1.2 sec

❌ 1.7 sec

***Objective 5* - Selectable Ops**

The Test Pilot *Grind*

Your next flight objective is to run a series of *navigation* tests.

- You will be flying routes with different *distances, altitudes, and speeds.*
- It would be very **tedious** to plug back in and modify your code between every test.
- The test pilot grind is grueling enough already - *write some code to enable easier changes!*

Selectable Operations

What you need is a nice User Interface on the CodeAIR. One that lets you select different routes before each flight!

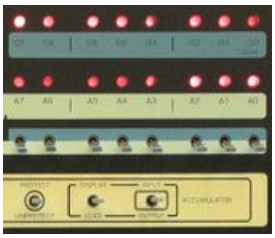
- Aw, but there's no screen on this thing.
- And just a couple of buttons.

  No way you can make a cool, selectable user interface, right??

WRONG!

Hacker UI

Actually, you have everything you need! Just like the hackers of old, you can use those buttons and LEDs to program your flights at *runtime!*



- Back in the day they entered whole computer programs using toggle switches and LEDs!

Concept: *Binary Numbers*

The BYTE LEDs are so-named because they're arranged as a 🔧binary byte.

- That's 8 bits, or "binary digits."

Take a few minutes to explore the 🔧binary toolbox entry to get familiar with how you can make numbers with ON and OFF lights!

LED Binary Patterns

Those 8 blue 🔧LEDs can display an 🔧integer value between 0 and 255.

- That's 256 different numbers, since 2^8 is 256!
- CodeAIR's LED 🔧API also provides a way to set multiple LEDs at once in 🔧binary, using the `leds.set_mask()` function.

```
# Set BYTE LEDs to 255 (all ON) with brightness=50
leds.set_mask(255, 50)
```

So if your flight routes (or anything else) are numbered 1, 2, 3, ... up to 255, then this fancy BYTE display can handle it!

User Interface Plan

Code a "Selectable Operations" UI using the two 🔧buttons and the 🔧LEDs.

- Press BTN-1 to scroll UP through the "menu". Wrap back around to 1 if you max-out.
- Show the current selection number on the BYTE LEDs in 🔧binary.
  - Hey, if the user doesn't know *binary* they got no business trying to fly a quadcopter, am I right?
- Press BTN-0 to *confirm* the current selection, and **start the action!**

Create a New File!

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

Be sure to name it exactly this way, with the `.py` extension so it will remain on the CodeAIR and you can later do `from utility import *`.

- You will need this *Selectable Ops* user interface for future Missions!
- Making a `utility` module is a good idea for miscellaneous *helper functions* you'll want to reuse.
- As your set of utility functions grows over time, you may want to organize this into separate modules like `user_ifc.py`, etc. But for now, keep it simple.

Check the 'Trek!

Run It!

Use BTN-1 to scroll, and BTN-0 to activate and test some colors.

CodeTrek:

```
1  from codeair import *
2  from time import sleep
3
4  def select_index(num_items):
5      """Wait for user to select an index from 1 to num_items by scrolling in binary
6         on the BYTE LEDs. Since zero would be all off, counting starts at 1. Max count
7         is all on, which is 255. Use BTN_1 to select and BTN_0 to confirm. Return chosen
8         index. Since it's 1-based you'll need to subtract 1 to use this for list indexing!
9      """
```

`select_index(num_items)` is a blocking 🔧function that waits for the user to select a number.

- Check out this amazing 🔧docstring. *Get in the habit of documenting all your functions!*

```
10     num_items = min(num_items, 255)
11     choice = 1
12     leds.set_status(50)
13     leds.set_mask(choice, 50)
```

That `leds.set_mask(bitmask, brightness)` function is the 🔧binary way to control the BYTE 🔧LEDs.

- Try something like `leds.set_mask(0b10101010, 50)` on the REPL and *feel the power of binary!*

```
14     while True:
15         if buttons.was_pressed(BTN_1):
16             choice = choice + 1
17             if choice > num_items:
18                 choice = 1
```

If the user goes past the max index, wrap back around to the first one.

```
19             speaker.beep(880, 50)
20             leds.set_mask(choice, 50)
21             sleep(0.15)
22             buttons.was_pressed()
```

A little *debouncing* here.

```
23         elif buttons.was_pressed(BTN_0):
24             speaker.beep(1000, 50)
25             speaker.beep(1200, 20)
```

```
            break
```

When BTN_0 is pressed, play a "confirmation beep" and break out of the loop!

```

    # Confirmation blink
    for i in range(3):
        leds.set_mask(choice, 70)
        sleep(0.3)
        leds.set_mask(0, 0)
        sleep(0.2)
```

Blink the selected choice so the user is certain what was selected.

- Notice `leds.set_mask(0, 0)` turns OFF all LEDs.

```

    leds.set_status(0)
    buttons.was_pressed()
    return choice
```

Finally, a bit more button-debouncing then 🔧return the choice.

```


# Test program for selectable ops
if __name__ == '__main__':
    print("Loaded module for testing.")
```

Detecting `import` versus a test program run:

This `utility.py` file is meant to be `import`ed by your code in the future.

- And in that future code, you don't really want to run a "color selection test" every time you `import utility`.

The `if __name__ == '__main__':` detects when this program is being run as the "main program" - not an import!

```

    color_list = [
        BLUE,
        WHITE,
        GREEN,
        RED,
        MAGENTA
    ]

    # Test the selector!
    while True:
        i = select_index(len(color_list))
        print("selected index=", i)

        color = color_list[i - 1]
        pixels.fill(color)

        freq = 400 + i * 100
        speaker.beep(freq, 100)
```

Your test program loop.

- Exercise and verify operation of your fancy new `select_index()` function by putting it through its paces with color-selection and whatnot!

Goals:

- Define a 🔧function `select_index(num_items)` that uses the 🔧buttons and 🔧LEDs to allow user selection of an item number.
- Call the `select_index()` function inside a `while True` loop, to cycle through a 🔧list of colors.

- Test the following sequence ***exactly***: 1, 3, 5, 4, 2
  - Based on the color list in the CodeTrek, that would show: BLUE, GREEN, MAGENTA, RED, WHITE in order!

Tools Found: Binary Numbers, BYTE LEDs, int, API, Buttons, RGB "pixel" LEDs, Print Function, Functions, list, Comments, Parameters, Arguments, and Returns

Solution:

```
from codeair import *
from time import sleep

def select_index(num_items):
    """Wait for user to select an index from 1 to num_items by scrolling in binary
       on the BYTE LEDs. Since zero would be all off, counting starts at 1. Max count
       is all on, which is 255. Use BTN_1 to select and BTN_0 to confirm. Return chosen
       index. Since it's 1-based you'll need to subtract 1 to use this for list indexing!
    """
    num_items = min(num_items, 255)
    choice = 1
    leds.set_status(50)
    leds.set_mask(choice, 50)
    while True:
        if buttons.was_pressed(BTN_1):
            choice = choice + 1
            if choice > num_items:
                choice = 1
            speaker.beep(880, 50)
            leds.set_mask(choice, 50)
            sleep(0.15)
            buttons.was_pressed()
        elif buttons.was_pressed(BTN_0):
            speaker.beep(1000, 50)
            speaker.beep(1200, 20)
            break

    # Confirmation blink
    for i in range(3):
        leds.set_mask(choice, 70)
        sleep(0.3)
        leds.set_mask(0, 0)
        sleep(0.2)

    leds.set_status(0)
    buttons.was_pressed()
    return choice


# Test program for selectable ops
if __name__ == '__main__':
    print("Loaded module for testing.")

    color_list = [
        BLUE,
        WHITE,
        GREEN,
        RED,
        MAGENTA
    ]

    # Test the selector!
    while True:
        i = select_index(len(color_list))
        print("selected index=", i)

        color = color_list[i - 1]
        pixels.fill(color)

        freq = 400 + i * 100
        speaker.beep(freq, 100)
```

***Objective 6* - Crash Testing**

Crash Testing

Ready to put your selectable ops code to work on some **flight tests?**

- Um, yeah, about that.
- Someone filed a report about your *alleged "drone dropped out of the sky like a rock" incident.*
- Remember last Mission, the ***Escape** challenge?* That's when you learned about 🔧exceptions in Python!

  You're gonna have to put some precautions in place!

Expect the Unexpected!

Last time, you modified the code to avoid the exception altogether. But you're bound to run into more surprises in the future. *How can you handle errors you aren't expecting?*

- When you're coding, sometimes there are errors that keep your code from running at all.
- But *sometimes* your code may not hit the errors until later, **after CodeAIR is flying!**
- Robust real-world software should handle 🔧exceptions, so your program can fail *gracefully.*

Concept: *Exception Handling*

In Python, you can handle 🔧exceptions (errors that might happen during your program) using a `try` block:

```
try:
    do_something()
except:
    print("Something went wrong!")
```

If an error occurs in `do_something()` and isn't handled there, the `except` block lets your program respond without crashing. For more advanced ways to handle specific errors, check the 🔧Exception Toolbox entry.

Crash Test Code

This Objective brings in your `utility.py` module. You will use it to select from a list of flight routes.

- But for starters, the flying part is `#TODO` - don't worry about that yet.
- Instead, see if you can break your code **without** *falling from the sky!*

Create a New File!

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

Check the 'Trek!

Run It!

Open the ☰Console

- Use BTN-1 and BTN-0 to select/confirm index 1,2,3, and watch those `print()` statements showing the selected route:
  `Selected route: 1.5m forward at 0.3m high.`
- Be sure to go *beyond* the valid selection range (more than the route list items) to see the 🔧exception.

CodeTrek:

```
1  from utility import *
```

Try out your new `utility` 🔧module.

- You DID run `utility.py` in your last Objective, *right?*

```
2
3  # List of (Height, Distance) tuples
4  routes = [
5      (0.3, 1.0),
6      (0.3, 1.5),
7      (0.3, 2.0),
8  ]
```

Make a list of flight routes.

- This is just a start.
- `(0.3, 1.0)` → means fly at 0.3m high for a distance of 1.0m

```
9
10 while True:
11     i = select_index(9)  # Up to 9 routes?
```

Set up the CRASH...

- Give `select_index()` a larger `num_items` than needed.
- There are only 3 items in the `routes` 🔧list after all!

```
12
13     height, dist = routes[i - 1]
```

Index the selected route, and 🔧unpack the 🔧tuple into `height, dist`.

- That's all fine if `i < len(routes)`

But if `i` is 3 or higher...

BOOM!

```
14     print(f"Selected route {i}: {dist}m forward at {height}m high.")
15
16     # TODO: Fly the selected route!
```

A nice 🔧print statement, and a *#TODO* you'll take care of in the next Objective.

```
17
```

Goals:

- 🔧Import your `utility` module.
- Make at least 3 in-range selections.
  - Watch the console `print()` statements.
- Select an *out of range* item. Make it go **BOOM!**
  - Show me an 🔧exception so you can move on to the next Objective and *handle it.*

Tools Found: Exception, import, list, Assignment, tuple, Print Function

Solution:

```
1  from utility import *
2
3  # List of (Height, Distance) tuples
4  routes = [
5      (0.3, 1.0),
6      (0.3, 1.5),
7      (0.3, 2.0),
```

```
]

while True:
    i = select_index(9)  # Up to 9 routes?

    height, dist = routes[i - 1]
    print(f"Selected route {i}: {dist}m forward at {height}m high.")

    # TODO: Fly the selected route!

```

Quiz 3 - Exceptional

Question 1: The LEDs show the following pattern. What number is being displayed in binary?

```
0 0 0 0 0 0 1 1
```

✓ 3

✗ 2

✗ 1

✗ 4

Question 2: What is the purpose of the following line of code?

```
if __name__ == '__main__':
```

✓ To execute the following code block only if this file is run as the main program, not an import.

✗ To ensure that this code is named `'__main__'`, regardless of the filename chosen by the programmer.

✗ Without this, the file cannot be imported by other programs.

Question 3: What is printed by the following code?

```
try:
    print("Starting")
    x = 1 / 0  # causes exception (ZeroDivisionError)
    print("Finished")
except:
    print("Ouch!")
```

✓

Starting Ouch!

✗

Starting Finished Ouch!

✗

Starting Ouch! Finished

Objective 7 - Test Pilot

Test Pilot

With all that preparation, no doubt you are READY TO FLY!

- Good. Because there's a lot of *testing* to do.

Goal: Push the Limits of Flow-Based Navigation

You've already experienced *sensor drift* and seen some of the drawbacks of *dead reckoning* navigation.

- Without an *External Positioning System* (see *Concept* box below) CodeAIR will need to work hard to maintain its position, using all the sensor data it can muster!

As the drone programmer, *you* need to know the limits:

- How accurately can you move a particular distance, using flow sensor X/Y?
- Is flow-sensor accuracy dependent on altitude?
- Would a slower velocity help or hurt?

> Concept: *External Positioning Systems*
>
> Many drones depend on **external positioning systems** to determine their location:
>
> - Outdoor drones often use GPS.
> - Indoor drones may rely on fixed-location *beacons*, which track the drone's position. These systems can be expensive and require careful planning and setup.
>
> While it's possible to integrate external systems with CodeAIR using its expansion connections, CodeAIR's default setup is designed to be **self-reliant** - it figures out its own position without external help!

Test Plan

> Check the 'Trek!

CodeTrek:

```
1  from codeair import *
2  from flight import *
3  from safety import *
4  from utility import *
```

Import your `safety` and `utility` modules.

```
5
6  # List of (Height, Distance) tuples
7  routes = [
8      (0.3, 1.0), # route 1
9      (0.3, 1.5),
10     (0.3, 2.0),
11     (0.6, 1.0),
12     (0.6, 1.5),
13     (0.6, 2.0),
14     (1.0, 1.0),
15     (1.0, 1.5),
16     (1.0, 2.0),
17 ]
```

A good starting 🔧list of *routes* for testing.

- You'll use `select_index()` to choose a route from this list.
- Each route contains a `(height, distance)` 🔧tuple.

```
18
19 try:
20     while True:
21         i = select_index(len(routes))
```

`try` to keep up here.

But seriously, check out how the whole `while` 🔧loop is 🔧indented beneath this `try:` statement.

- Scroll down and you'll see the matching `except` clause. *We'll get to that later!*

Also, check it out:

This is how the *pros* call `select_index()`.

- For the `num_items` 🔧argument just pass the length of the 🔧list!

```
22
23          height, dist = routes[i - 1]
24          print(f"Selected route {i}: {dist}m forward at {height}m high.")
```

Subtract 1 from the selection number to index the `routes` list. *Because ==lists= start at zero, right?*

Check out how this 🔧unpacking assignment splits the 🔧tuple into `height` and `dist` 🔧variables.

```
25
26          velocity = 0.2  # meters/sec
```

The velocity doesn't change... *yet.*

- It would be *sweet* to add this to the `routes` 🔧tuple though, eh?

```
27
28          if button_arm():
29              fly.take_off(height)
30              batt_check_steady()
```

Take off, then pause for a `batt_check_steady()`.

- Will do the *default* 1-second hover while flashing the battery status lights.

```
31
32              # Forward
33              pixels.fill(YELLOW)
34              fly.forward(dist, velocity)
35
36              # Hover pause
37              pixels.fill(BLUE)
38              fly.steady(0.5)
39
40              # Land
41              pixels.fill(WHITE)
42              fly.land()
43              pixels.off()
```

The Test Sequence

1. Fly forward at the specified `height` and `dist`.
2. Hover / pause briefly.
3. Land!

The flight stages are color coded YELLOW → BLUE → WHITE for easier tracking.

```
44
45  except:
46      # Exception! Emergency landing.
47      print("Exception!")
48      pixels.fill(PINK)
49      fly.land()
```

This `except:` block matches your `try:` block above.

- Bright pink 🔧pixel leds will alert you that an exception happened!

- And of course, land gracefully when the program ends rather than falling from the sky.

50

Hint:

- **🔧Pixel LEDs turning PINK?**

  Why do they turn pink when you plug CodeAIR back into the computer.

  - Or, when you press the STOP button...

  This is because when you press STOP, or when CodeSpace detects a new USB connection to your device, CodeSpace sends a CTRL-C aka `KeyboardInterrupt` to CodeAIR.

  - The `KeyboardInterrupt` is a type of **Exception**.
  - And since your code is inside a `try:` block, this 🔧exception is caught by your `except:` block!

Goals:

- Define a list of `routes` with at least 6 entries (🔧tuples).
- Place your main loop inside a `try:` block.
- Land gracefully in the `except:` block.
- When armed: Take off, check battery, fly forward, hover, then land.

Tools Found: Exception, Optical Flow Sensor, Laser Range Sensors, tuple, list, Loops, Indentation, Keyword and Positional Arguments, Assignment, Variables, RGB "pixel" LEDs

Solution:

```python
from codeair import *
from flight import *
from safety import *
from utility import *

# List of (Height, Distance) tuples
routes = [
    (0.3, 1.0), # route 1
    (0.3, 1.5),
    (0.3, 2.0),
    (0.6, 1.0),
    (0.6, 1.5),
    (0.6, 2.0),
    (1.0, 1.0),
    (1.0, 1.5),
    (1.0, 2.0),
]

try:
    while True:
        i = select_index(len(routes))

        height, dist = routes[i - 1]
        print(f"Selected route {i}: {dist}m forward at {height}m high.")

        velocity = 0.2  # meters/sec

        if button_arm():
            fly.take_off(height)
            batt_check_steady()

            # Forward
            pixels.fill(YELLOW)
            fly.forward(dist, velocity)

            # Hover pause
```

```
            pixels.fill(BLUE)
            fly.steady(0.5)

            # Land
            pixels.fill(WHITE)
            fly.land()
            pixels.off()

except:
    # Exception! Emergency landing.
    print("Exception!")
    pixels.fill(PINK)
    fly.land()

```

Mission 6 Complete

Nice Navigation!

You've *navigated* another challenging series of Objectives as you charted a course to deeper Python knowledge, and increased mastery of your CodeAIR's onboard systems.

- The flow sensor is amazing, but it has limitations which you now have a hands-on feel for.
- Checking the battery *while flying* is essential for safe flying, and you've now updated your `safety` module with that capability.
- And don't forget a nice taste of exception handling you experienced in this Mission! *It won't be your last tangle with that topic.*

Try Your Skills: *Remix!*

Navigation challenges are a staple of aerial robotics competitions. Do a little brainstorming and write down some of your own ideas for challenges with CodeAIR.

- Could you enhance `routes` to run a different course based on the selection? *(Not just go forward and land...)*
- How about adding a `safety` function to prevent *take off* if there's an object too close *above* CodeAIR?
- ...I'm sure you have LOTS more ideas!

More Missions Coming Soon!

The Firia Labs team is busy working on new Missions, and you can expect to see additional updates as new Missions are added regularly.

- There is SO much more to discover with CodeAIR!