Blinka Jump PyBadge Game

Created by Liz Clark

https://learn.adafruit.com/blinka-jump-pybadge-game

Last updated on 2021-11-15 08:05:25 PM EST
# Table of Contents

## Overview
- Inspiration
- Prerequisite Guides
- Supplies

## Installing CircuitPython
- Set up CircuitPython Quick Start!
- Further Information

## CircuitPython Libraries
- The Adafruit CircuitPython Library Bundle
- Downloading the Adafruit CircuitPython Library Bundle
- The CircuitPython Community Library Bundle
- Downloading the CircuitPython Community Library Bundle
- Understanding the Bundle
- Example Files
- Copying Libraries to Your Board
- Understanding Which Libraries to Install
- Example: ImportError Due to Missing Library
- Library Install on Non-Express Boards
- Updating CircuitPython Libraries and Examples

## Coding the PyBadge

## Creating Sprites

## Creating Text

## Animate Blinks

## Randomly Generate Blue Smoke Monsters

## Make Blinks Jump

## Game Lives

## Keep Score

## Playing Blinks Jump
Overview

Blinka Jump is a video game created with CircuitPython. Using the displayio library you can use sprites, text and other features to build your own game.

The game was coded with the PyBadge in mind. The PyBadge board has everything you need onboard to be a compact handheld gaming device.

Inspiration

This game is based on the Chrome browser's jumping dinosaur game Easter egg. In this version, Blinka needs to jump over the Sparky the Blue Smoke Monsters to save the circuits running on CircuitPython.
Prerequisite Guides

Before diving into this guide, it's recommend to look at the Creating Your First Tilemap Game with CircuitPython (https://adafruit.it/L-C) Learn Guide by Tim C (https://adafruit.it/L-D). It goes into detail about how the displayio mechanics work when coding a game.

For additional references, you'll also want to check out the CircuitPython Animated Sprite Pendents (https://adafruit.it/L-E) guide and the CircuitPython Display Support Using displayio (https://adafruit.it/EGh) guide.

Supplies

- **Adafruit PyBadge for MakeCode Arcade, CircuitPython, or Arduino**
  What's the size of a credit card and can run CircuitPython, MakeCode Arcade or Arduino? That's right, its the Adafruit PyBadge! We wanted to see how much we...
  https://www.adafruit.com/product/4200

- **Lithium Ion Polymer Battery Ideal For Feathers - 3.7V 400mAh**
  Lithium-ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light, and powerful. The output ranges from 4.2V when completely charged to 3.7V. This...
  https://www.adafruit.com/product/3898
Pink and Purple Braided USB A to Micro B Cable - 2 meter long
This cable is super-fashionable with a woven pink and purple Blinka-like pattern!
First let's talk about the cover and over-molding. We got these in custom colors,...
https://www.adafruit.com/product/4148

Installing CircuitPython

*CircuitPython* (https://adafru.it/tB7) is a derivative of *MicroPython* (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the CIRCUITPY flash drive to iterate.

The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

Set up CircuitPython Quick Start!

Follow this quick step-by-step for super-fast Python power :)

Download the latest version of CircuitPython for PyBadge via circuitpython.org
https://adafru.it/EF4

Further Information

For more detailed info on installing CircuitPython, check out *Installing CircuitPython* (https://adafru.it/Amd).
Click the link above and download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).

Plug your PyBadge into your computer using a known-good USB cable.

A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the Reset button on the back of your board (indicated by the green arrow in the first image). You will see an image on the display instructing you to drag a UF2 file to your board, and the row of NeoPixel RGB LEDs on the front will turn green (indicated by the arrow and square in the second image). If they turn red, check the USB cable, try another USB port, etc.

Your reset button may be white or black!

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!
You will see a new disk drive appear called BADGEBOOT.

Drag the adafruit_circuitpython_etc.uf2 file to BADGEBOOT.

The LED will flash. Then, the BADGEBOOT drive will disappear and a new disk drive called CIRCUITPY will appear.

That's it, you're done! :)

CircuitPython Libraries

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.
Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. These files are called libraries. Some of them are built into CircuitPython. Others are stored on your CIRCUITPY drive in a folder called lib. Part of what makes CircuitPython so great is its ability to store code separately from the firmware itself. Storing code separately from the firmware makes it easier to update both the code you write and the libraries you depend.

Your board may ship with a lib folder already, it’s in the base directory of the drive. If not, simply create the folder yourself. When you first install CircuitPython, an empty lib directory will be created for you.

CircuitPython libraries work in the same way as regular Python modules so the Python docs (https://adafru.it/rar) are an excellent reference for how it all should work. In Python terms, you can place our library files in the lib directory because it’s part of the Python path by default.

One downside of this approach of separate libraries is that they are not built in. To use them, one needs to copy them to the CIRCUITPY drive before they can be used. Fortunately, there is a library bundle.

The bundle and the library releases on GitHub also feature optimized versions of the libraries with the .mpy file extension. These files take less space on the drive and have a smaller memory footprint as they are loaded.

Due to the regular updates and space constraints, Adafruit does not ship boards with the entire bundle. Therefore, you will need to load the libraries you need when you begin working with your board. You can find example code in the guides for your board that depends on external libraries.

Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.
The Adafruit CircuitPython Library Bundle

Adafruit provides CircuitPython libraries for much of the hardware they provide, including sensors, breakouts and more. To eliminate the need for searching for each library individually, the libraries are available together in the Adafruit CircuitPython Library Bundle. The bundle contains all the files needed to use each library.

Downloading the Adafruit CircuitPython Library Bundle

You can download the latest Adafruit CircuitPython Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

Match up the bundle version with the version of CircuitPython you are running. For example, you would download the 6.x library bundle if you're running any version of CircuitPython 6, or the 7.x library bundle if you're running any version of CircuitPython 7, etc. If you mix libraries with major CircuitPython versions, you will get incompatible mpy errors due to changes in library interfaces possible during major version changes.

Click to visit circuitpython.org for the latest Adafruit CircuitPython Library Bundle

https://adafru.it/ENC

Download the bundle version that matches your CircuitPython firmware version. If you don't know the version, check the version info in boot_out.txt file on the CIRCUITPY drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

There's also a py bundle which contains the uncompressed python files, you probably don't want that unless you are doing advanced work on libraries.
The CircuitPython Community Library Bundle

The CircuitPython Community Library Bundle is made up of libraries written and provided by members of the CircuitPython community. These libraries are often written when community members encountered hardware not supported in the Adafruit Bundle, or to support a personal project. The authors all chose to submit these libraries to the Community Bundle make them available to the community.

These libraries are maintained by their authors and are not supported by Adafruit. As you would with any library, if you run into problems, feel free to file an issue on the GitHub repo for the library. Bear in mind, though, that most of these libraries are supported by a single person and you should be patient about receiving a response. Remember, these folks are not paid by Adafruit, and are volunteering their personal time when possible to provide support.

Downloading the CircuitPython Community Library Bundle

You can download the latest CircuitPython Community Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

Click for the latest CircuitPython Community Library Bundle release

https://adafruit.io/VCn

The link takes you to the latest release of the CircuitPython Community Library Bundle on GitHub. There are multiple versions of the bundle available. Download the bundle version that matches your CircuitPython firmware version. If you don't know the version, check the version info in boot_out.txt file on the CIRCUITPY drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

Understanding the Bundle

After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.
Open the bundle folder. Inside you'll find two information files, and two folders. One folder is the lib bundle, and the other folder is the examples bundle.

Now open the lib folder. When you open the folder, you'll see a large number of .mpy files, and folders.

Example Files

All example files from each library are now included in the bundles in an examples directory (as seen above), as well as an examples-only bundle. These are included for two main reasons:

- Allow for quick testing of devices.
- Provide an example base of code, that is easily built upon for individualized purposes.
Copying Libraries to Your Board

First open the lib folder on your CIRCUITPY drive. Then, open the lib folder you extracted from the downloaded zip. Inside you'll find a number of folders and .mpy files. Find the library you'd like to use, and copy it to the lib folder on CIRCUITPY.

If the library is a directory with multiple .mpy files in it, be sure to copy the entire folder to CIRCUITPY/lib.

This also applies to example files. Open the examples folder you extracted from the downloaded zip, and copy the applicable file to your CIRCUITPY drive. Then, rename it to code.py to run it.

Understanding Which Libraries to Install

You now know how to load libraries on to your CircuitPython-compatible microcontroller board. You may now be wondering, how do you know which libraries you need to install? Unfortunately, it's not always straightforward. Fortunately, there is an obvious place to start, and a relatively simple way to figure out the rest. First up: the best place to start.

When you look at most CircuitPython examples, you'll see they begin with one or more `import` statements. These typically look like the following:

- `import library_or_module`

However, `import` statements can also sometimes look like the following:

- `from library_or_module import name`
They can also have more complicated formats, such as including a `try / except` block, etc.

The important thing to know is that an `import` statement will always include the name of the module or library that you're importing.

Therefore, the best place to start is by reading through the `import` statements.

Here is an example import list for you to work with in this section. There is no setup or other code shown here, as the purpose of this section involves only the import list.

```python
import time
import board
import neopixel
import adafruit_lis3dh
import usb_hid
from adafruit_hid.consumer_control import ConsumerControl
from adafruit_hid.consumer_control_code import ConsumerControlCode
```

Keep in mind, not all imported items are libraries. Some of them are almost always built-in CircuitPython modules. How do you know the difference? Time to visit the REPL.

In the [Interacting with the REPL section](https://adafruit.it/Awz) on The REPL page in this guide, the `help("modules")` command is discussed. This command provides a list of all of the built-in modules available in CircuitPython for your board. So, if you connect to the serial console on your board, and enter the REPL, you can run `help("modules")` to see what modules are available for your board. Then, as you read through the `import` statements, you can, for the purposes of figuring out which libraries to load, ignore the statement that import modules.

The following is the list of modules built into CircuitPython for the Feather RP2040. Your list may look similar or be anything down to a significant subset of this list for smaller boards.
Now that you know what you're looking for, it's time to read through the import statements. The first two, `time` and `board`, are on the modules list above, so they're built-in.

The next one, `neopixel`, is not on the module list. That means it's your first library! So, you would head over to the bundle zip you downloaded, and search for neopixel. There is a neopixel.mpy file in the bundle zip. Copy it over to the lib folder on your CIRCUITPY drive. The following one, `adafruit_lis3dh`, is also not on the module list. Follow the same process for adafruit_lis3dh, where you'll find adafruit_lis3dh.mpy, and copy that over.

The fifth one is `usb_hid`, and it is in the modules list, so it is built in. Often all of the built-in modules come first in the import list, but sometimes they don't! Don't assume that everything after the first library is also a library, and verify each import with the modules list to be sure. Otherwise, you'll search the bundle and come up empty!

The final two imports are not as clear. Remember, when `import` statements are formatted like this, the first thing after the `from` is the library name. In this case, the library name is `adafruit_hid`. A search of the bundle will find an adafruit_hid folder. When a library is a folder, you must copy the entire folder and its contents as it is in the bundle to the lib folder on your CIRCUITPY drive. In this case, you would copy the entire adafruit_hid folder to your CIRCUITPY/lib folder.

Notice that there are two imports that begin with `adafruit_hid`. Sometimes you will need to import more than one thing from the same library. Regardless of how many times you import the same library, you only need to load the library by copying over the adafruit_hid folder once.

That is how you can use your example code to figure out what libraries to load on your CircuitPython-compatible board!
There are cases, however, where libraries require other libraries internally. The internally required library is called a dependency. In the event of library dependencies, the easiest way to figure out what other libraries are required is to connect to the serial console and follow along with the `ImportError` printed there. The following is a very simple example of an `ImportError`, but the concept is the same for any missing library.

**Example: `ImportError` Due to Missing Library**

If you choose to load libraries as you need them, or you're starting fresh with an existing example, you may end up with code that tries to use a library you haven't yet loaded. This section will demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

This demonstration will only return an error if you do not have the required library loaded into the lib folder on your CIRCUITPY drive.

Let's use a modified version of the Blink example.

```python
import board
import time
import simpleio
led = simpleio.DigitalOut(board.LED)

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.

You have an `ImportError`. It says there is no module named `simpleio`. That's the one you just included in your code!
Click the link above to download the correct bundle. Extract the lib folder from the downloaded bundle file. Scroll down to find simpleio.mpy. This is the library file you're looking for! Follow the steps above to load an individual library file.

The LED starts blinking again! Let's check the serial console.

No errors! Excellent. You've successfully resolved an `ImportError`!

If you run into this error in the future, follow along with the steps above and choose the library that matches the one you're missing.

**Library Install on Non-Express Boards**

If you have an M0 non-Express board such as Trinket M0, Gemma M0, QT Py M0, or one of the M0 Trinkeys, you'll want to follow the same steps in the example above to install libraries as you need them. Remember, you don't need to wait for an `ImportError` if you know what library you added to your code. Open the library bundle you downloaded, find the library you need, and drag it to the lib folder on your CIRCUITPY drive.

You can still end up running out of space on your M0 non-Express board even if you only load libraries as you need them. There are a number of steps you can use to try to resolve this issue. You'll find suggestions on the [Troubleshooting page](https://adafruit.it/Den).

**Updating CircuitPython Libraries and Examples**

Libraries and examples are updated from time to time, and it's important to update the files you have on your CIRCUITPY drive.

To update a single library or example, follow the same steps above. When you drag the library file to your lib folder, it will ask if you want to replace it. Say yes. That's it!
A new library bundle is released every time there's an update to a library. Updates include things like bug fixes and new features. It's important to check in every so often to see if the libraries you're using have been updated.

## Coding the PyBadge

Once you've finished setting up your PyBadge with CircuitPython, you can add these libraries to the `lib` folder:

- `adafruit_bitmap_font`
- `adafruit_bus_device`
- `adafruit_display_text`
- `adafruit_imageload`
- `simpleio.mpy`

Then, you can click on the Download: Project Zip link below at the top of the code to download the code file and bitmap file for the game sprites.

```python
import time
from random import randint
from micropython import const
import board
import terminalio
import displayio
import adafruit_imageload
import digitalio
import simpleio
from gamepadshift import GamePadShift
from adafruit_display_text import label

# setup for PyBadge buttons
BUTTON_LEFT = const(128)
BUTTON_UP = const(64)
BUTTON_DOWN = const(32)
BUTTON_RIGHT = const(16)
BUTTON_SEL = const(8)
BUTTON_START = const(4)
BUTTON_A = const(2)
BUTTON_B = const(1)

pad = GamePadShift(digitalio.DigitalInOut(board.BUTTON_CLOCK),
                   digitalio.DigitalInOut(board.BUTTON_OUT),
                   digitalio.DigitalInOut(board.BUTTON_LATCH))

current_buttons = pad.get_pressed()
last_read = 0

# enables speaker
speakerEnable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
speakerEnable.switch_to_output(value=True)

# Sprite cell values
EMPTY = 0
BLINKA_1 = 1
BLINKA_2 = 2
```
SPARKY = 3
HEART = 4
JUMP_1 = 5
JUMP_2 = 6

# creates display
display = board.DISPLAY
# scale=2 allows the sprites to be bigger
group = displayio.Group(scale=2)

# Blinka sprite setup
blinka, blinka_pal = adafruit_imageload.load("/spritesNew.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette)

# creates a transparent background for Blinka
blinka_grid = displayio.TileGrid(blinka, pixel_shader=blinka_pal,
    width=2, height=1,
    tile_height=16, tile_width=16,
    default_tile=EMPTY)

blinka_grid.x = 0
blinka_grid.y = 32

blinka_group = displayio.Group()
blinka_group.append(blinka_grid)

# first Sparky sprite
sparky0, sparky0_pal = adafruit_imageload.load("/spritesNew.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette)

sparky0_pal.make_transparent(7)
sparky0_grid = displayio.TileGrid(sparky0, pixel_shader=sparky0_pal,
    width=1, height=1,
    tile_height=16, tile_width=16,
    default_tile=SPARKY)

sparky0_grid.x = 100
sparky0_grid.y = 32

sparky0_group = displayio.Group()
sparky0_group.append(sparky0_grid)

# 2nd Sparky sprite
sparky1, sparky1_pal = adafruit_imageload.load("/spritesNew.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette)

sparky1_pal.make_transparent(7)
sparky1_grid = displayio.TileGrid(sparky1, pixel_shader=sparky1_pal,
    width=1, height=1,
    tile_height=16, tile_width=16,
    default_tile=SPARKY)

sparky1_grid.x = 100
sparky1_grid.y = 32

sparky1_group = displayio.Group()
sparky1_group.append(sparky1_grid)

# 3rd Sparky sprite
sparky2, sparky2_pal = adafruit_imageload.load("/spritesNew.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette)

sparky2_pal.make_transparent(7)
sparky2_grid = displayio.TileGrid(sparky2, pixel_shader=sparky2_pal,
    width=1, height=1,
    tile_height=16, tile_width=16,
    default_tile=SPARKY)

sparky2_grid.x = 100
sparky2_grid.y = 32
sparky2_group = displayio.Group()
sparky2_group.append(sparky2_grid)

# heart sprite group
life_bit, life_pal = adafruit_imageload.load("/spritesNew.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette)
life_grid = displayio.TileGrid(life_bit, pixel_shader=life_pal,
    width=3, height=1,
    tile_height=16, tile_width=16,
    default_tile=HEART)

life_group = displayio.Group()
life_group.append(life_grid)

# adding all graphics groups to the main display group
group.append(blinka_group)
group.append(sparky0_group)
group.append(sparky1_group)
group.append(sparky2_group)
group.append(life_group)

# text area for the running score
score_text = "      
font = terminalio.FONT
score_color = 0x0000FF

# text for "game over" graphic
game_over_text = label.Label(font, text="         
# score text
score_area = label.Label(font, text=score_text, color=score_color)
# text for "new game" graphic
new_game_text = label.Label(font, text="           

# coordinants for text areas
score_area.x = 57
score_area.y = 6
game_over_text.x = 13
game_over_text.y = 30
new_game_text.x = 8
new_game_text.y = 30

# creating a text display group
text_group = displayio.Group()
text_group.append(score_area)
text_group.append(game_over_text)
text_group.append(new_game_text)

group.append(text_group)

group.append(text_group)

display.show(group)

# state for hit detection
flash = False
# states to see if a Sparky is on screen
sparky0 = False
sparky1 = False
sparky2 = False

# array of Sparky states
sparky_states = [sparky0, sparky1, sparky2]
# array of x location for Sparky's
sparky_x = [sparky0_grid.x, sparky1_grid.x, sparky2_grid.x]

def life():
    for _ in range(0, 3):
        life_grid[_, 0] = EMPTY
for hearts in range(life_count):
    life_grid[hearts, 0] = HEART

# lives at beginning of the game
life_count = 3

# variables for scoring
jump_score = 0
total_score = 0
bonus = 0

# state for Blinka being in default 'slither' mode
snake = True

# state to check if Blinka has jumped over Sparky
cleared = False

# state for the end of a game
end = False

# state for a new game beginning
new_game = True

# state for detecting game over
game_over = False

# variable to change between Blinka's two slither sprites
b = 1

# variable to hold time.monotonic() count for Blinka slither animation
slither = 0

# variables to hold time.monotonic() count to delay Sparky spawning
blue = 0
smoke = 0
monster = 0

while True:
    # checks if button has been pressed
    if (last_read + 0.01) < time.monotonic():
        buttons = pad.get_pressed()
        last_read = time.monotonic()

    # new game
    if new_game and not game_over:
        # graphics for new game splash screen
        blinka_grid.y = 16
        blinka_grid[0] = JUMP_1
        blinka_grid[1] = JUMP_2
        sparky0_grid.x = 5
        sparky1_grid.x = 40
        sparky2_grid.x = 65
        score_area.text = 300
        new_game_text.text = "BLINKA JUMP"
        life()
        # if start is pressed...
        if current_buttons != buttons:
            if buttons & BUTTON_START:
                # prepares display for gameplay
                print("start game")
                new_game_text.text = "start"
                life_count = 3
                start = time.monotonic()
                new_game = False
                end = False
                sparky0_grid.x = 100
                sparky1_grid.x = 100
                sparky2_grid.x = 100

    # if game has started...
    if not game_over and not new_game:
        # gets time.monotonic() to have a running score
        mono = time.monotonic()
        score = mono - start
        # adds 10 points every time a Sparky is cleared
        total_score = score + jump_score
        # displays score as text
        score_area.text = int(total_score)
puts Sparky states and x location into callable arrays

```
for s in range(3):
    sparky_state = sparky_states[s]
    sparky_location = sparky_x[s]
```

# Sparkys are generated using a staggered delay
# and matching an int to a random int
# 1st Sparky
if (blue + 0.03) < time.monotonic():
    if randint(1, 15) == 3:
        sparky_states[0] = True
        blue = time.monotonic()
# 2nd Sparky
if (smoke + 0.07) < time.monotonic():
    if randint(1, 15) == 7:
        sparky_states[1] = True
        smoke = time.monotonic()
# 3rd Sparky
if (monster + 0.12) < time.monotonic():
    if randint(1, 15) == 12:
        monster = time.monotonic()

# if a Sparky is generated, it scrolls across the screen 1 pixel at a time
# 1st Sparky
if sparky_states[0] is True:
    sparky0_grid.x -= 1
    sparky_x[0] = sparky0_grid.x
display.refresh(target_frames_per_second=120)
    # when a Sparky is 16 pixels off the display,
    # it goes back to its starting position
    if sparky0_grid.x is -16:
        sparky_states[0] = False
        sparky0_grid.x = 100
        sparky_x[0] = sparky0_grid.x
# 2nd Sparky
if sparky_states[1] is True:
    sparky1_grid.x -= 1
    sparky_x[1] = sparky1_grid.x
display.refresh(target_frames_per_second=120)
    if sparky1_grid.x is -16:
        sparky_states[1] = False
        sparky1_grid.x = 100
        sparky_x[1] = sparky1_grid.x
# 3rd Sparky
if sparky_states[2] is True:
    sparky2_grid.x -= 1
    sparky_x[2] = sparky2_grid.x
display.refresh(target_frames_per_second=120)
    if sparky2_grid.x is -16:
        sparky2_grid.x = 100
        sparky_x[2] = sparky2_grid.x
```

# if no lives are left then the game ends
```
if life_count is 0:
    game_over = True
```

# if the A button is pressed then Blinky is no longer in the default
# slither animation aka she jumps
if current_buttons != buttons:
    if buttons & BUTTON_A:
        snake = False

# heart sprites are displayed to show life count
```
life()
```

# if Blinky is slithering...
```
if snake:
# Blinka default location
blinka_grid.y = 32
# empty 2nd tile so that the jump sprite can be shown using
# the same tilegrid
blinka_grid[1] = EMPTY
# every .15 seconds Blinka's slither sprite changes
# so that her slithering is animated
# b holds tilegrid position to display correct sprite
if (slither + 0.15) < time.monotonic():
    blinka_grid[0] = b
    b += 1
    slither = time.monotonic()
if b > 2:
    b = 1
# if a Sparky collides with Blinka while she is slithering...
for s in range(3):
    if sparky_x[s] == 8 and blinka_grid.y == 32:
        # tone is played
        simpleio.tone(board.SPEAKER, 493.88, 0.05)
        simpleio.tone(board.SPEAKER, 349.23, 0.05)
        # lose a life
        life_count = life_count - 1
# if the A button is pressed then...
else:
    # Blinka JUMPS
    # y location changes one row up and both jump sprites are shown
    blinka_grid.y = 16
    blinka_grid[0] = JUMP_1
    blinka_grid[1] = JUMP_2
    # if Blinka jumps over a Sparky...
    for j in range(3):
        if sparky_x[j] == 8 and not cleared:
            # 10 points to the player
            bonus += 1
            jump_score = bonus * 10
            cleared = True
            # special victory tone is played
            simpleio.tone(board.SPEAKER, 523.25, 0.005)
            simpleio.tone(board.SPEAKER, 783.99, 0.005)
            # resets back to Blinka animation
            snake = True
            # resets that Blinka has not jumped over a Sparky
            cleared = False
# if there are no more lives, the game is over
if game_over and not new_game:
    # game over text is displayed
    game_over_text.text = "GAME OVER"
    score_area.text = "   ">
    # end game tone is played
    # and then the screen holds with the last
    # sprites on screen and game over text
    if not end:
        simpleio.tone(board.SPEAKER, 220, 0.05)
        simpleio.tone(board.SPEAKER, 207.65, 0.05)
        simpleio.tone(board.SPEAKER, 196, 0.5)
        end = True
# if the start button is pressed...
if (current_buttons != buttons) and game_over:
    if buttons & BUTTON_START:
        # display, states and score are reset for gameplay
        game_over_text.text = "   "
        life_count = 3
        start = time.monotonic()
        game_over = False
        end = False
        total_score = 0
        jump_score = 0
bonus = 0
score = 0
blue = 0
smoke = 0
monster = 0
sparky0_grid.x = 100
sparky1_grid.x = 100
sparky2_grid.x = 100
# game begins again with all Sparky's off screen

Your PyBadge CIRCUITPY drive should look like this after you load the libraries, bitmap and code.py file:

Creating Sprites

All of the sprites in the game come from the single bitmap file. Different tilegrids are setup so that the sprites can move independently while still referencing the same file.

The bitmap file is created to be 16 pixels high by 112 pixels wide. This way it can be divided evenly into 16x16 squares to access the individual sprites. These sprites can then be called by index position, similar to an array.

The sprite names are assigned as variables to match their index positions on the bitmap.

    gaming_spritesNew.bmp
In total, there will be five tilegrids for the sprites: three for Blinka, Sparky the Blue Smoke Monster and one for a heart. Each of them are setup to use the adafruit_imageload library to load the bitmap file. All of their backgrounds are made to be transparent by eliminating the seventh color in their indexed color profile, which in this case is black.

All of them have a tile_height and tile_width of 16, but their width and height will vary depending on how many sprites will be shown. For example, Blinka has a width of 2 and a height of 1 so that when Blinka jumps, there is enough space for the two sprites that create the Blinka jumping sprite. The default_tile defines which sprite is shown as a default for each tilegrid and will also vary between tilegrids.

```python
blinka, blinka_pal = adafruit_imageload.load("/spritesNew.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette)
#  creates a transparent background for Blinka
blinka_pal.make_transparent(7)
blinka_grid = displayio.TileGrid(blinka, pixel_shader=blinka_pal,
    width=2, height=1,
    tile_height=16, tile_width=16,
    default_tile=EMPTY)
```

In finishing up each tilegrid's setup, the default position is setup by defining the x and y coordinates for each sprite on the display's grid. To finish up, a display group is created for each tilegrid and the tilegrid is added to that group. Later, the individual sprite's groups are added to the main display group. This allows you to have greater control over the sprites individually later in the code.

```python
blinka_grid.x = 0
blinka_grid.y = 32
blinka_group = displayio.Group()
blinka_group.append(blinka_grid)
```
Creating Text

There are three text objects for the game: the score, the game title and the game over text. All three are setup using the label class from the adafruit_display_text library.

The `score_text` will hold the score of the game and update constantly. `new_game_text` is shown when the PyBadge is booted up with the code to say "BLINKA JUMP" and `game_over_text` shows "GAME OVER" when you lose a game.

You can either hard code the text that will be displayed or leave it open so that it defaults to show nothing and can be updated later in the code. In the case of all three of these text objects, the default text is left empty by using `text = "   "`. The number of spaces between the quotation marks matters, since a string that is longer than the spacing will cause an error.

```python
score_text = "   "
font = terminalio.FONT
score_color = 0x0000FF

game_over_text = label.Label(font, text = "   ", color = 0xFF00FF)

score_area = label.Label(font, text = score_text, color = score_color)

new_game_text = label.Label(font, text = "   ", color = 0xFF00FF)
```

After the setup, all of the text objects' default positions are defined with `x` and `y` coordinates. Following that, a display group for the text objects is created and all of the text objects are added to that group. Finally, the `text_group` is added to the main display group, which also has the sprites that were created earlier in the code.
# coordinates for text areas
score_area.x = 57
score_area.y = 6
game_over_text.x = 13
game_over_text.y = 30
new_game_text.x = 8
new_game_text.y = 30

# creating a text display group
text_group = displayio.Group()
text_group.append(score_area)
text_group.append(game_over_text)
text_group.append(new_game_text)

# adding text group to main display group
group.append(text_group)

## Animate Blinka

When Blinka isn't jumping, she's happily slithering along. In the code, this is accomplished by quickly switching between two Blinka sprites: one where she is sitting in her usual coiled position and the second where her neck is outstretched.

Blinka slithers whenever the `snake` state is `True`. This state tracks whether or not Blinka is jumping. If you remember back to Blinka's tilegrid setup, her tilegrid is two tiles wide. As a result, the second square, or index 1, of `blinka_grid` is set to be `EMPTY` when Blinka is slithering.

```python
if snake:
    # Blinka default location
    blinka_grid.y = 32
    # empty 2nd tile so that the jump sprite can be shown using the same tilegrid
    blinka_grid[1] = EMPTY
```
To switch back and forth between the two sprites, you could use `time.sleep(x)` to alternate between displaying the sprites with a delay. However, doing this actually delays the entire loop and slows everything down.

To get around this, you can use `time.monotonic()` to count time and change sprites after a certain amount of time has passed; in this case **0.15** seconds.

Every **0.15** seconds, Blinka's sprite is updated. The variable `b` holds the sprite index, alternating between 1 and 2 or `BLINKA_1` and `BLINKA_2`.

`slither` is reset each time to hold `time.monotonic()` to be able to compare to the current `time.monotonic()` count.

```python
# every .15 seconds Blinka's slither sprite changes
# so that her slithering is animated
# b holds tilegrid position to display correct sprite
if (slither + 0.15) < time.monotonic():
    blinka_grid[0] = b
    b += 1
    slither = time.monotonic()
if b > 2:
    b = 1
```
Randomly Generate Blue Smoke Monsters

In total, there are three Sparky's that can be on the screen at any given time. To keep game play interesting, each of their appearances on screen are randomized.

The first way that they're randomized is by having varying delays using \texttt{time.monotonic()}, similar to how Blinka's animation is delayed without slowing down the loop. Each Sparky's generation is delayed by 0.03 seconds, 0.07 seconds and 0.12 seconds respectively. This allows their appearances to be staggered.

Once the defined time has elapsed, then there is a check to see if a random integer matches with a predefined integer: 3, 7 and 12 respectively.

If the random integer matches, then the \texttt{sparky\_state} for the corresponding Sparky sprite is updated to \texttt{True}, which allows them to appear on screen. If the random integer is not a match, then the \texttt{time.monotonic()} count begins again.

```python
# Sparkys are generated using a staggered delay
# and matching an int to a random int
# 1st Sparky
if (blue + 0.03) < time.monotonic():
    if randint(1, 15) == 3:
        sparky_states[0] = True
    blue = time.monotonic()
# 2nd Sparky
if (smoke + 0.07) < time.monotonic():
    if randint(1, 15) == 7:
        sparky_states[1] = True
    smoke = time.monotonic()
# 3rd Sparky
if (monster + 0.12) < time.monotonic():
    if randint(1, 15) == 12:
    monster = time.monotonic()
```
When a Sparky's state is True, their x coordinate location updates by 1 pixel continuously, allowing them to smoothly move across the screen.

The corresponding index in the sparky_x array is also updated to hold the x coordinate. This is used later in the loop for discerning whether Blinka and a Sparky have a collision.

Finally, when a Sparky's x location is -16 pixels, which is off-screen, their x coordinate is reset to 100 to prepare them for the next time they have a random integer match.

```python
if sparky_states[0] is True:
    sparky0_grid.x -= 1
    sparky_x[0] = sparky0_grid.x
    display.refresh(target_frames_per_second=120)
    # when a Sparky is 16 pixels off the display,
    # it goes back to its starting position
    if sparky0_grid.x is -16:
        sparky_states[0] = False
        sparky0_grid.x = 100
        sparky_x[0] = sparky0_grid.x
    # 2nd Sparky
    if sparky_states[1] is True:
        sparky1_grid.x -= 1
        sparky_x[1] = sparky1_grid.x
        display.refresh(target_frames_per_second=120)
        if sparky1_grid.x is -16:
            sparky_states[1] = False
            sparky1_grid.x = 100
            sparky_x[1] = sparky1_grid.x
    # 3rd Sparky
    if sparky_states[2] is True:
        sparky2_grid.x -= 1
        sparky_x[2] = sparky2_grid.x
        display.refresh(target_frames_per_second=120)
        if sparky2_grid.x is -16:
            sparky2_grid.x = 100
            sparky_x[2] = sparky2_grid.x
```
Make Blinka Jump

Blinka jumps every time the A button is pressed on the PyBadge board. To do this, when a button press is detected from the A button, the `snake` state is set to `False`. You'll remember that whenever the `snake` state is `True`, then the Blinka sprite is in the default slithering animation.

```python
if current_buttons != buttons:
    if buttons & BUTTON_A:
        snake = False
```

Once `snake` is `False`, then the Blinka sprite's `y` coordinate is updated to be `16`, or one row up, and the tilegrid's sprites are updated to be `JUMP_1` and `JUMP_2`.

```python
else:
    # Blinka JUMPS
    # y location changes one row up and both jump sprites are shown
    blinka_grid.y = 16
    blinka_grid[0] = JUMP_1
    blinka_grid[1] = JUMP_2
```
Blinka Jump uses a "three strikes and you're out" game play rule, where you have three lives to use during your game. These lives are represented by heart sprites at the top of the screen.

There is a function, called `life()`, that displays the heart sprites to represent game play lives. It references the variable `life_count` to display the correct number of heart sprites.

```python
# function to display the heart sprites for lives
def life():
    for _ in range(0, 3):
        life_grid[_, 0] = EMPTY
    for hearts in range(life_count):
        life_grid[hearts, 0] = HEART

# lives at beginning of the game
life_count = 3
```

In the loop, the `life()` function runs to display the sprites and there is a check for when the `life_count` reaches 0, since that ends the game.

```python
# heart sprites are displayed to show life count
life()

# if no lives are left then the game ends
if life_count is 0:
    game_over = True
```
The `life_count` decreases by one every time that Blinka collides with a Sparky. This is detected when one of the Sparkys' `x` location is 8 and Blinka is slithering. When this occurs, the built-in speaker plays two tones (specifically a tritone, which sounds menacing) and `life_count` is updated to show the decrease.

```python
# if a Sparky collides with Blinka while she is slithering...
for s in range(3):
    if sparky_x[s] == 8 and blinka_grid.y == 32:
        # tone is played
        simpleio.tone(board.SPEAKER, 493.88, 0.05)
        simpleio.tone(board.SPEAKER, 349.23, 0.05)
        # lose a life
        life_count = life_count - 1
```

### Keep Score

The score in Blinka Jump is actually a count of the time that the game has been running using `time.monotonic()`. The score is displayed in the `score_area` text object and is updated constantly throughout the game.
In addition to the `time.monotonic()` component of the score, every time Blinka jumps over a Sparky you get a 10 point bonus. The game counts a successful jump by checking if a Sparky is at \( x \) coordinate 8 and Blinka is jumping. This bonus score is stored in `jump_score` and is added to the running `total_score`.

Two tones are also played (a triumphant perfect fifth) every time Blinka clears a Sparky.

When you power up your PyBadge for a relaxing game of Blinka Jump, you are greeted with a new game splash screen.
To start the game, you press the start button at the top of the PyBadge. When the game detects the start button is pressed, the game components are reset to their starting states. The three Sparky's are sent off screen, life_count is reset and the new_game_text is emptied.

```python
# if start is pressed...
if current_buttons != buttons:
    if buttons & BUTTON_START:
        # prepares display for gameplay
        print("start game")
        new_game_text.text = "        "
        life_count = 3
        start = time.monotonic()
        new_game = False
        end = False
        spakry0_grid.x = 100
        spakry1_grid.x = 100
        spakry2_grid.x = 100
```
Gameplay begins, utilizing all of the gaming mechanics discussed previously in the guide. You'll use the A button on the PyBadge to successfully help Blinka jump over all the Sparky the Blue Smoke Monsters.

When you run out of game lives, the game goes into the `game_over` state. The Blinka and Sparky sprites freeze in their positions at the time of the final collision being detected, "GAME OVER" text is displayed in the center of the screen and three tones (a sad chromatic slide) play.

```python
if game_over and not new_game:
    # game over text is displayed
    game_over_text.text = "GAME OVER"
    score_area.text = "    "
    # end game tone is played
    # and then the screen holds with the last
    # sprites on screen and game over text
    if not end:
        simpleio.tone(board.SPEAKER, 220, 0.05)
        simpleio.tone(board.SPEAKER, 207.65, 0.05)
        simpleio.tone(board.SPEAKER, 196, 0.5)
    end = True
```
The game remains in this state until you press the start button again on the PyBadge to begin a new game. When that button press is detected, the game states are reset to their starting positions, similar to the initial new game mode when you first boot up the PyBadge.

```python
# if the start button is pressed...
if (current_buttons != buttons) and game_over:
    if buttons & BUTTON_START:
        # display, states and score are reset for gameplay
        game_over_text.text = "  
        life_count = 3
        start = time.monotonic()
        game_over = False
        end = False
        total_score = 0
        jump_score = 0
        bonus = 0
        score = 0
        blue = 0
        smoke = 0
        monster = 0
        sparky0_grid.x = 100
        sparky1_grid.x = 100
        sparky2_grid.x = 100
        # game begins again with all Sparky's off screen
```