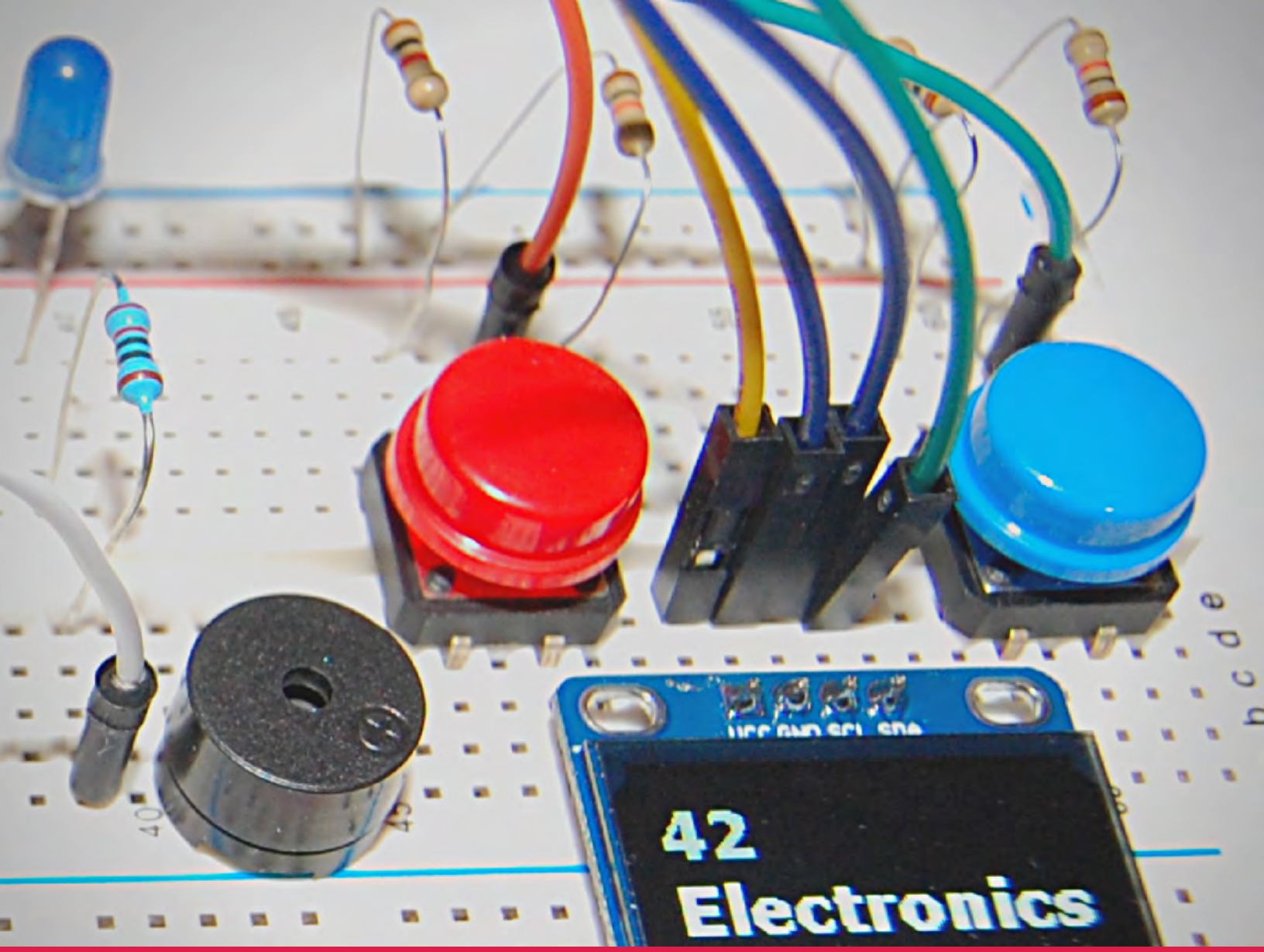


INTRO TO ROBOTICS

LEVEL B

Working with Sensors & Intermediate Programming

BY ERIC FEICKERT

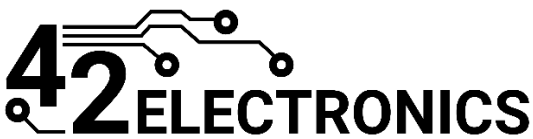
42 ELECTRONICS

Intro to Robotics Level B

Working with Sensors & Intermediate Programming

By Eric Feickert

V21057



www.42electronics.com

Level B TABLE OF CONTENTS

LESSON B-5

SWITCHES & CORRECTING FOR SWITCH BOUNCE

Lesson Overview

- ► Pull-Up / Pull-Down Options
- ► Slide & Toggle Switches
- ► Switch Bounce
- ► Using Delays

Concepts to Review

- ► Switches (Lesson A-5)
- ► Boolean Logic, If/Else Statements (Lesson A-14)
- ► GPIO Pin States, GPIO Pin Cleanup (Lesson A-16)

Materials Needed

- ❑ Raspberry Pi connected to Monitor, Keyboard, and Mouse
- ❑ Circuit from Lesson B-4
- ❑ Slide Switch x1
- ❑ Pushbutton Switch x1
- ❑ Long Male-to-Male Jumper Wires x3
- ❑ Short Male-to-Male Jumper Wires x2

LESSON B-5

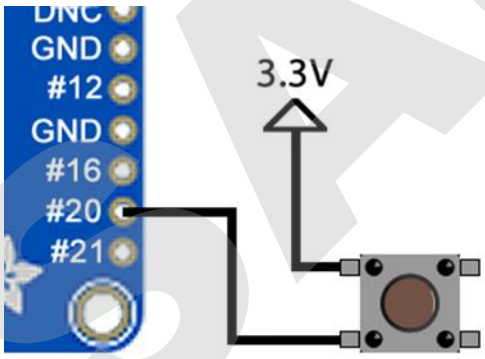
SWITCHES & CORRECTING FOR SWITCH BOUNCE

In this lesson, you will learn to work with different types of switches including both slide and pushbutton switches to accomplish various tasks. You will also learn to write code to overcome an electrical shortcoming of switches known as switch bounce.

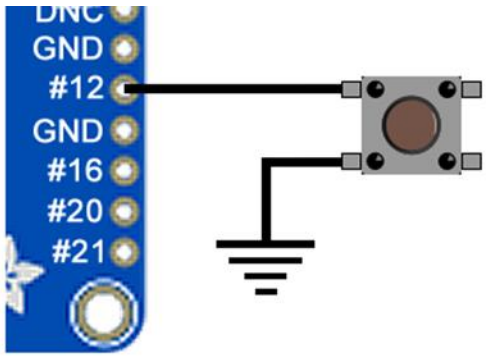
Pull-Up / Pull-Down Options

In Level A, you learned how to use resistors in a specific configuration to keep inputs reading high or low, when the pushbutton switch was not pressed. This was opposed to inputs being left floating, where an input is not connected to anything until a switch is pressed, which can lead to inputs being read unreliably by your program.

While it's good to understand the basics behind using resistors for pull-up and pull-down, the Raspberry Pi can help you simplify circuit design by doing the pull-up and pull-down work for you. The Raspberry Pi contains internal resistors that can be accessed when setting up a GPIO pin as an input. This will simplify your switch wiring to a single jumper wire from the GPIO pin to the switch, and another wire from the switch to 3.3V or ground, depending on whether you want a high or low to trigger the input.



Active high - needs pull down in software



Active low - needs pull up in software

In Python, pulling an input up or down can be done when the pin is assigned as an input, by supplying a third parameter called **pull_up_down**:

```
GPIO.setup(12, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

This command will initialize **GPIO12** as an input, and configure that input to be internally pulled up. This input will be an active low, which means this input will remain high until it is made active by connecting it to ground or low, hence the name "active low". **GPIO12** will now only need to be connected to ground to trigger the input, which can be done very simply using a switch and two wires.

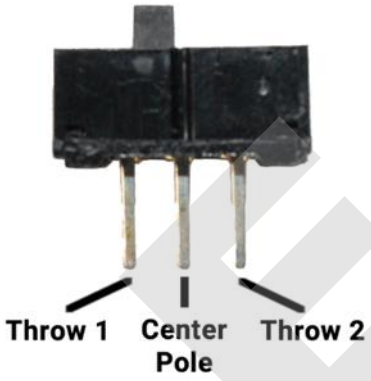
The opposite of active low is active high, which means the input will remain grounded or low until it is made active by connecting to 3.3V. This can be accomplished by using **GPIO.PUD_DOWN** after the **pull_up_down** parameter:

```
GPIO.setup(20, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
```

This command will initialize **GPIO20** as an input and configure that input to be internally pulled down. **GPIO20** will now only need to be connected to 3.3V to trigger the input, which can be done with a switch and two wires.

Slide or Toggle Switch

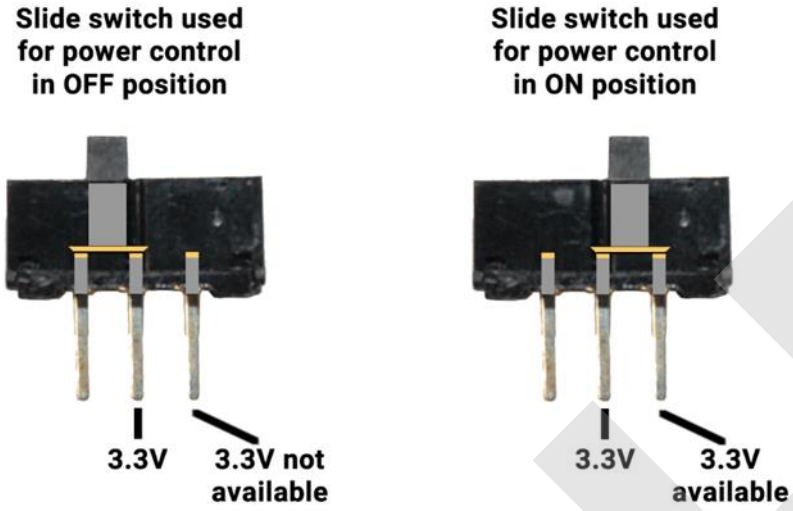
Until now, you have only worked with pushbutton switches that make connection when they are pressed, but lose that connection when they are released. There may come a time when you want a switch to stay connected without being pressed. This type of switch is called a toggle switch.



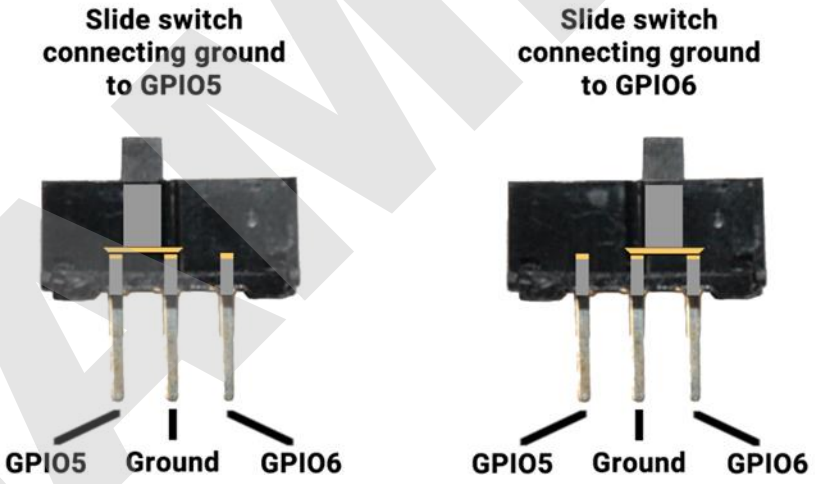
A toggle switch "toggles" one or more inputs between one or more outputs. One very common type of toggle switch is a two-position slide switch.

In a two-position slide switch the center pole can connect to one of the throws at a time, based on the position of the slider.

These switches can be used as power on/off switches for electronic devices by connecting power through the center pole as well as one of the throws:



A slide switch can also be used as a selector between two different inputs. In the diagram below, a slide switch is connected to two GPIO pins, with the center terminal attached to ground.



In one position, GPIO5 is grounded and a GPIO6 will be internally pulled up by the Raspberry Pi. In the other position, GPIO6 will be grounded and GPIO5 will be internally pulled up by the Raspberry Pi.

Programming for a slide switch is identical to the programming you've already done for pushbutton switches. Your program could use these switch inputs to light up different LEDs, modify the frequency of a piezo buzzer, or anything else you might want to control.

Switch Bounce

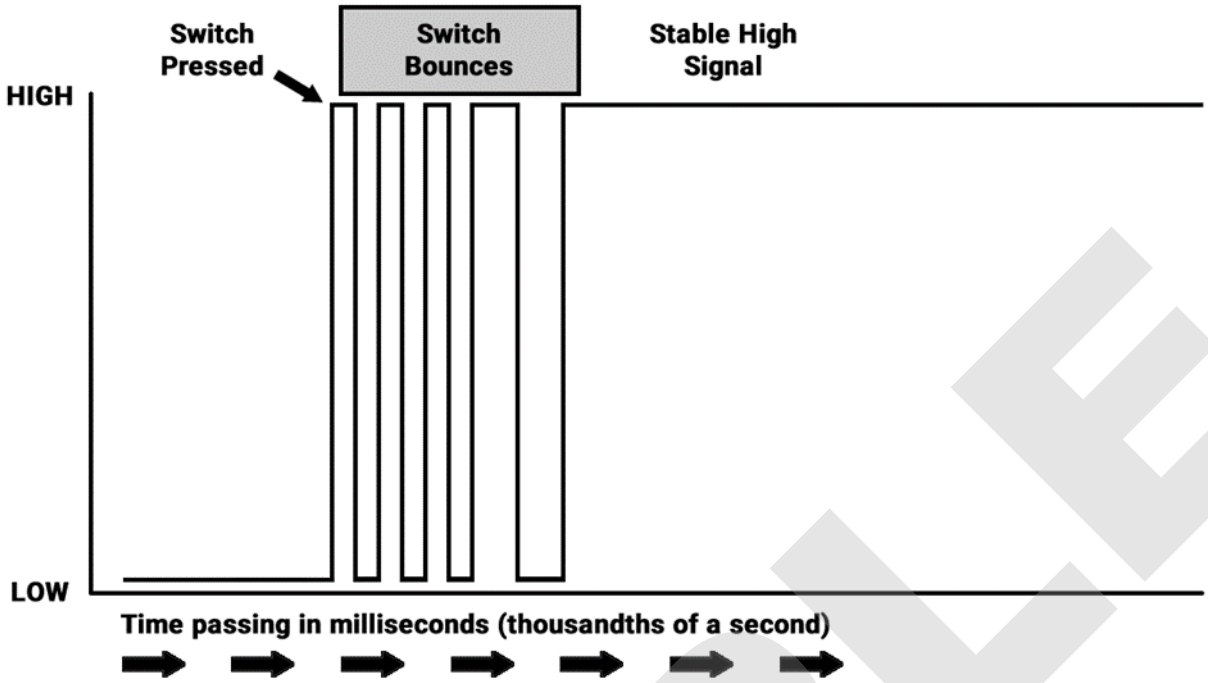
When switches are opening or closing, metal plates called contacts must come together or move apart to control the flow of electricity through the switch. These plates are made of metal, so they will conduct electricity, but this metal-to-metal contact can result in the two plates bouncing off each other a very tiny amount, until they settle either fully together, or fully apart. This microscopic movement can result in the switch being open and closed very quickly, typically less than one millisecond, until the switch settles into the desired state. This phenomenon is referred to as switch bounce.

All mechanical switches will exhibit some level of switch bounce, but it may not be a problem in their specific application. For example, the light switch in your room has switch contact bounce but you don't see the light flash on and off multiple times when you flip the switch on. This is due to the latency in overhead lighting and most other electronics.

Switch bounce may become a problem depending on how you plan to use the switch input. If the program just turns on an LED based on switch input, then bounce won't really be a problem. Extremely rapid on/off LED activity will be masked by the amount of time it takes the LED to turn on and off, so the LED will appear to turn on and off smoothly, even though switch bounce is still occurring.

The problem occurs when you have the ability to check a switch very quickly, like the Raspberry Pi is able to do, thousands of times per second. Imagine you have program that is supposed to keep track of how often you drink water each day. Each time you drink a glass of water you press a pushbutton, and the program keeps track of how many times the switch went from low to high, adding to that count throughout the day.

Pressing the button quickly, only one time, should result in one being added to the current count. Due to switch bounce, there may be a few extra low to high transitions for every time the button is pushed, and these will all be logged to the counter.



Pressing the button 8 times throughout the day might result in a final count of 32 glasses of water, which is obviously very inaccurate. Removing these extra counts is called debouncing, which can be done via hardware using additional components, or via software. In the interest of keeping electronic connections to the Pi as simple as possible, we will focus on debouncing using software in Python. You may find many effective methods for software debouncing in programs online. Below, we will detail a few of the simplest options.

One method of debouncing involves checking the input a certain amount of time after the first time an input is triggered. This will allow time for the bouncing to settle and the input state can be confirmed once it is steady. This delay is usually not very long, as a delay of around 0.02 seconds should be enough to allow the input to settle into either a steady high or low state. If GPIO6 was configured as an active low input, that code would look something like this:

```
while True:

    if GPIO.input(6) == False:

        time.sleep(0.02)

        if GPIO.input(6) == False:

            water_count = water_count + 1
```

The first if statement will trigger the very first time GPIO6 goes low. The program will sleep for 0.05 seconds, and then check GPIO6 again. If GPIO6 is still False, then one will be added to **`water_count`**. If this second check of GPIO6 is True then nothing else will happen, and the value of **`water_count`** will not be changed.

There are a couple of problems with this method. If you somehow managed to press the button for less than .02 seconds, no water would be recorded, as the second level if statement would not evaluate as False. Also, since this entire loop will be evaluated every .02 seconds, you can easily end up with multiple **`water_count`** increments for a single button press, which defeats the purpose of the delay.

You could increase the amount of delay to avoid multiple **`water_count`** events being counted when the button is held down. Increasing this delay will however increase the amount of time between the first and second checks, which could allow you a greater chance of pressing the button and releasing without being counted. You could tune this value to find a balance between quick presses not registering and longer presses registering more than once, but it may still not give you performance you would like.

Another method of debouncing includes adding a delay to the loop that runs after an input is triggered:

```
while True:

    if GPIO.input(6) == False:

        water_count = water_count + 1

        time.sleep(.1)
```

The very first time GPIO6 goes low the loop will be triggered. One will be added to **water_count** and the program will sleep for .1 seconds before resuming further checks. This will eliminate the possibility of quick button presses not being counted, and the delay in this loop can be modified to eliminate single button presses registering more than one count per press, without affecting the responsiveness of the first button press.

The only problem left with this code is that button presses longer than .1 seconds will continue to register additional **water_count** events. This can be fixed by adding a while loop to hold the program as long as GPIO6 remains low:

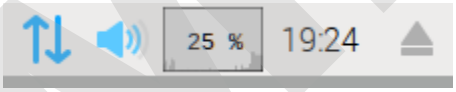
```
while True:

    if GPIO.input(6) == False:

        water_count = water_count + 1

        time.sleep(.1)

        while GPIO.input(6) == False:

            pass

        time.sleep(0.02)
```

The **pass** command in the new while loop tells that loop to do nothing when triggered, which is exactly what we want. As long as GPIO6 remains low the program will remain stuck in this while loop, doing nothing, and not adding to the **water_count** value. As soon as GPIO6 is no longer False, the rest of the initial if statement will run. The final **time.sleep(0.02)** was added to debounce the opening of the switch, as without the delay, the bouncing during release of the switch might cause **water_count** to be increased unintentionally.

Using a Delay to Save System Resources

Due to its tiny size, your Raspberry Pi has a limited amount of processing power that can be devoted to tasks. This processing power is how much work can be done by the CPU, which stands for Central Processing Unit. CPU utilization is a value that indicates how hard the CPU is working. A utilization value of 100% indicates that the CPU cannot handle any more work, and system performance will be highly degraded. A low CPU utilization value like 2% means that 98% of the CPU is still available and ready to do processing work.

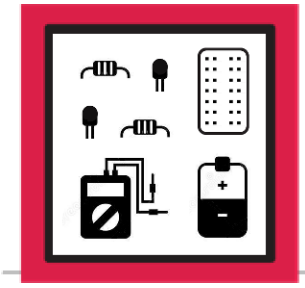
The current utilization of the CPU is displayed in the top-right corner of the menu bar:



In this image, the CPU utilization is at 25%, meaning 25% of the available processing power is being used, with 75% still free.

If CPU utilization is too high, no processing power is left over for routine operating system tasks. By running a loop in your program with no delay, you can easily consume more system resources than you really need, causing instability in the Raspbian Operating System.

If you have a loop checking an input in a program, adding a small delay like 0.1 seconds will free up valuable resources, allowing the OS to complete all the tasks it needs in the background. Your program will likely run no different, however you will see a big difference in the CPU utilization number, as well as how much heat is being generated by your Raspberry Pi.



LESSON B-5

ACTIVITY #1:
ADDING SWITCHES TO THE CIRCUIT

In this activity you will add switches to the circuit you built in Lesson B-4

Step #1

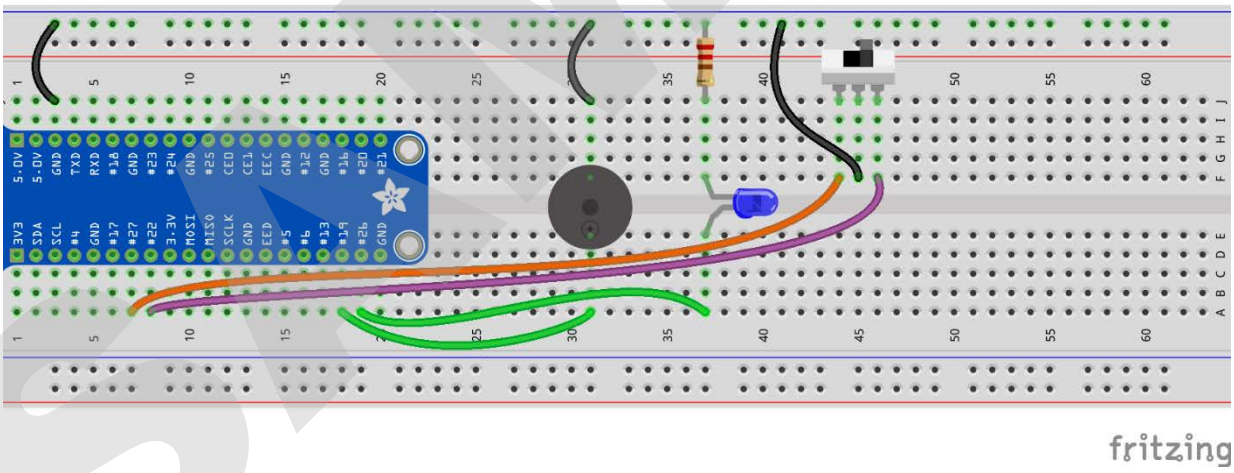
Make sure your Raspberry Pi is powered off. Using the circuit from Lesson B-4 as a starting point, add a slide switch connected between these points:

Slide switch in J44 through J46

Long jumper wire from F44 to A7

Short jumper wire from F45 to N2-41

Long jumper wire from F46 to A8



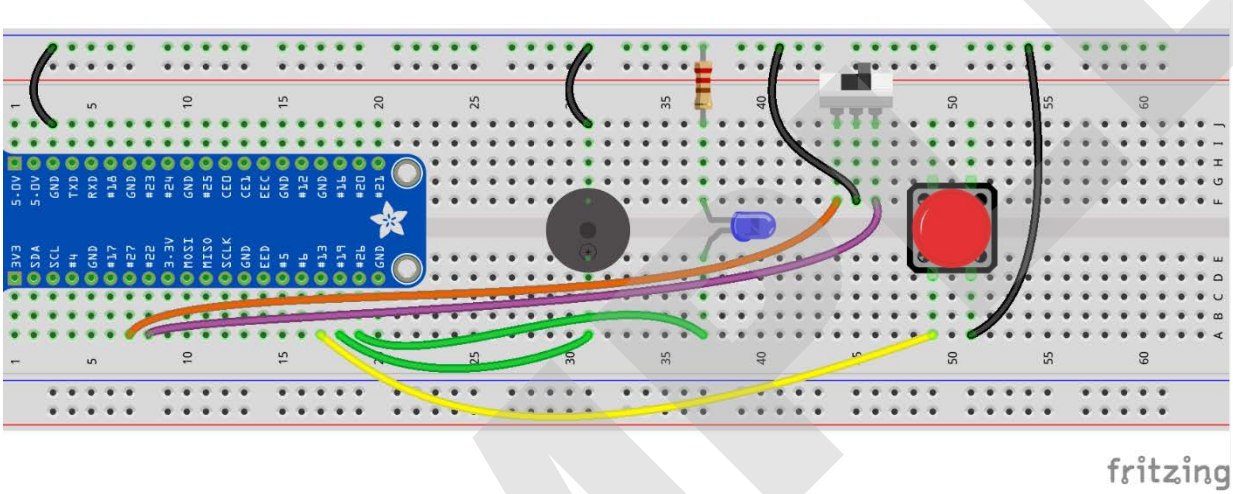
Step #2

Next, you will add a pushbutton switch that will be used as a stop button for your program. Add a pushbutton switch connected between GPIO13 and ground.

Pushbutton switch in D49, D51, G49, and G51

Long jumper wire from A49 to A17

Short jumper wire from A51 to N2-54



The breadboard circuit is now ready for you to create a program that will use the switches to control some program functions.

LESSON B-5

ACTIVITY #2:
COUNTING BUTTON PRESSES

In this activity, you will create a program that will count each time the pushbutton connected to GPIO13 is pressed and print the current count to the console.

Step #1

The first step will be to create an empty program that you can use to make a program that counts button presses. Create a new program in Thonny and save it to your Desktop as **`button_counter`**.

Step #2

Since this program will interact with GPIO pins and have a delay, you must import the **`RPi.GPIO`** and **`time`** modules. Import these modules at the top of your program:

```
import RPi.GPIO as GPIO, time
```

Step #3

Next, the GPIO pin mode will need to be set to BCM. **GPIO13** will also need to be configured as an active-low input since it's connected to ground. Since this switch does not have any pull-up resistors attached, this will need to be configured within the program. Add the following two highlighted lines just below your import code:

```
import RPi.GPIO as GPIO, time

GPIO.setmode(GPIO.BCM)

GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

Step #4

A variable will need to be used to hold the amount of button presses that have occurred, and this value should be zero when the program starts. Create a variable called **button_count** and set it equal to zero:

```
import RPi.GPIO as GPIO, time

GPIO.setmode(GPIO.BCM)

GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)


button_count = 0
```

Step #5

Since this program is interacting with GPIO pins, use a **`try/except`** loop so the **`except:`** condition can catch keyboard interrupts and run a **`GPIO.cleanup()`** before the program exits. Add the following code to the bottom of your program:

```
button_count = 0


try:


except KeyboardInterrupt:
    GPIO.cleanup()
```

Your program can now exit smoothly, and without any errors, when the stop button is pressed in Thonny.

Step #6

Now it's time to build the main loop that will watch **GPIO13** for a low or **False** state. If **GPIO13** is low, the button has been pressed, and the value of **button_count** will be incremented by 1. You will also use this loop to print the current value of **button_count**. Add the highlighted block of code to the end or your program:

```
button_count = 0


try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
except KeyboardInterrupt:
    GPIO.cleanup()
```

Make sure that the print statement is indented inside the **if** statement. If not, the value of **button_count** will be printed every time the main loop runs, which is very, very often.

Step #7

Since this program will be checking **GPIO13** very often, it will consume more resources than it really needs. Adding a delay of **0.05** seconds to the main loop will slow down the button checks a small amount, while drastically reducing the CPU load:

```
try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
        time.sleep(0.05)
except KeyboardInterrupt:
```

Make sure the indentation on this delay is aligned with the **if:** block above so that it runs even if the **if:** block does not get triggered by **GPIO13**.

Step #8

Run the program and press the button a few times. Do you notice anything about the count? It's not very accurate, at all! You didn't add any code to slow down the checking of **GPIO13** which is happening about 1000 times per second. Every time that **GPIO13** is checked, and found to be low (button is pressed), **button_count** is incremented by 1 and the value is printed to the console. You will need to tune up this loop with a delay to make it run more accurately, which you will do in the next activity.

Below is a copy of the whole program for your reference:

```
import RPi.GPIO as GPIO, time

GPIO.setmode(GPIO.BCM)

GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)


button_count = 0

try:

    while True:

        if GPIO.input(13) == False:

            button_count = button_count + 1

            print(button_count)

        time.sleep(0.05)

except KeyboardInterrupt:

    GPIO.cleanup()
```

LESSON B-5

ACTIVITY #3:
TUNING A LOOP USING A DELAY

In this activity, you will modify the program from the last activity to make it count button presses more accurately.

Step #1

You will be adding a delay to the main program to slow down how often the state of GPIO13 is checked. Use a delay value of 2 seconds and see how the program responds. Add a **`time.sleep(2)`** just below the print statement:

```
try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
            time.sleep(2)
         time.sleep(0.05)
except KeyboardInterrupt:
    GPIO.cleanup()
```

Step #2

Run the program and press the button a few times. Pressing the button results in **button_count** being incremented by 1, the new value of **button_count** being printed to the console, and then a two second delay. This delay works well to slow down the loop and make counts more accurate, but the program feels a little unresponsive during those large two second delays when new presses are ignored.

Reduce the value of the delay from **2** down to **0.2**:

```
try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
            time.sleep(0.2)
        time.sleep(0.05)
except KeyboardInterrupt:
    GPIO.cleanup()
```

Step #3

Run the program again with the new delay value and press the button a few times. You will find that a value of 0.2 for the delay is just right for keeping the program responsive after a button press, as well as eliminating a single press being counted multiple times.

The word "tuning" was used in title of this section because a loop might need slight timing changes based on how it's causing your program to behave. No delay caused multiple counts, while 2 full seconds made the program feel unresponsive. For this specific program, a delay of 0.2 was a good balance between those two to ensure the most accurate counting performance from the program.

LESSON B-5

ACTIVITY #4:
ADDING LED CONFIRMATION USING SLIDE SWITCH

In this activity, you will modify the program from the last activity to flash the LED each time the value of **button_count** is incremented. Input from the slide switch will be used to enable or disable the LED indicator.

Step #1

The GPIO pins for the slide switch and LED will need to be configured so they can be used in this program. **GPIO22** will need to be configured as an active-low input (software pull-up required), and **GPIO26** will need to be an output. Using your program from the last activity as a starting point, make these additions just below the existing **GPIO.setup** line:

```
GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(22, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(26, GPIO.OUT)

button_count = 0
```

Step #2

The main loop will already be running and checking the pushbutton switch on **GPIO13**. We only want the LED to come on when an increment event happens, so we can nest the LED code inside of the existing if block. After the value is **button_count** is printed, add an **if** condition that will check to see if **GPIO22** is low or **False**, and if so, turn on the LED at **GPIO26**:

```
try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
            if GPIO.input(22) == False:
                GPIO.output(26, GPIO.HIGH)
            time.sleep(0.2)
```

The time delay from the last lesson can now perform the additional task of keeping the LED on for 0.2 seconds, which is a good quick flash for an indicator light.

Step #3

The only thing missing is a way to turn the LED off. We can fix this by adding a **GPIO.output** line to the end of the main loop that will turn off the LED:

```
try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
            if GPIO.input(22) == False:
                GPIO.output(26, GPIO.HIGH)
            time.sleep(0.2)
            GPIO.output(26, GPIO.LOW)
```

The LED will be turned off every time the main loop runs. If the slide switch is applying ground to **GPIO22** when the main loop runs, then the LED will be turned on, delayed for 0.2 seconds, and then turned off at the end of the main loop.

If the slide switch is not applying a ground to **GPIO22** when the main loop runs, the LED will not turn on, the program will still delay 0.2 seconds, and then the LED will be told to turn off, even though it's already off. Sending a **GPIO.LOW** command to a GPIO pin that's already low will not harm anything. This just ensures that the LED gets turned off at the end of the main loop, regardless of whether it happens to be on or off.

Step #4

Run the program. Press the pushbutton to accumulate some counts, and then move the slide switch to the opposite position, and verify the LED reacts as expected. With the slide switch selector closest to the pushbutton switch, counts will be indicated by the LED flash, With the slide switch in the opposite position, the LED will stay off, but counts will continue to accumulate and be printed when the pushbutton is pressed.

A copy of the entire program is available on the next page for reference.

Leave the circuit assembled for use in Lesson B-6.

```
import RPi.GPIO as GPIO, time
GPIO.setmode(GPIO.BCM)
GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(22, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(26, GPIO.OUT)

button_count = 0

try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
            if GPIO.input(22) == False:
                GPIO.output(26, GPIO.HIGH)
            time.sleep(0.2)
            GPIO.output(26, GPIO.LOW)
        time.sleep(0.05)

except KeyboardInterrupt:
    GPIO.cleanup()
```

LESSON B-5

QUESTIONS FOR UNDERSTANDING

1. Can pull-ups and pull-downs be added within your program, or does every input switch require connections to physically pull-up or pull-down resistors?

2. Why is switch contact bounce a problem for programs that are counting switch presses?

3. Does an active-low input need to be connected to 3.3V or ground to trigger that input?

Answers Can be Found on the Next Page

Answers

1. **Can pulling inputs up and down be done within your program, or does every switch require connections to physically pull-up or pull-down resistors?**

   ANSWER: Pulling inputs up and down can be done within your program, when you configure the GPIO pin as an input.

2. **Why is switch contact bounce a problem for programs that are counting switch presses?**

   ANSWER: If your application for counting switch contact is very accurate like a computer and can register multiple switch contacts per second, so a single button press could result in several switch counts, rather than only one being counted.

3. **Does an active-low input need to be connected to 3.3V or ground to trigger that input?**

   ANSWER: An active-low input means that the input must be low to activate the input. That input will need a ground applied to overcome the pull-up and activate the input in software.

LESSON B-11

RFID SYSTEMS

Lesson Overview

- How RFID Works
- RFID for Access Control
- Security Concerns with RFID
- MFRC522 Tag Reader
- Removing Trailing Spaces in Python
- Shebang or #!

Concepts to Review

- Administrative File Management (Lesson B-1)
- Using Github to Clone Libraries (Lesson B-8)

Materials Needed

- ❑ Raspberry Pi connected to Monitor, Keyboard, and Mouse
- ❑ Assembled Circuit from Lesson B-10
- ❑ RFID Reader x1
- ❑ RFID Tag x1
- ❑ RFID Card x1

LESSON B-11

RFID SYSTEMS

In this lesson, you will learn about RFID systems, their components, and some of their applications. You will connect an RFID reader that will write and read RFID cards, as well as create a program that will read an RFID card and perform an action based on the value that's read from the card.

RFID Technology

RFID stands for radio-frequency identification. This system was originally designed for powering a device using radio waves. The RFID receiver was powered by a transmitter sending radio waves at a very specific frequency. The receiver did not contain its own power source, instead it contained special circuitry to convert the incoming radio waves into power that could be used to power up additional circuits in the receiver.

Around 1970, new ideas and patents started to emerge from this existing RFID technology. Proposed use cases involved automated toll collection systems, banking, security, and medical applications, as well as many others.

RFID technology can now be seen everywhere. Automated toll collection is installed on many of highways. Most pets have RFID implants that allow for identification of the animal without any external markings like a collar or tags. Many businesses rely on RFID cards to manage employee access control, instead of handing out physical keys to the building.

How RFID Works

Modern RFID systems operate using readers and tags. The reader contains a radio transmitter and receiver, commonly referred to as a transceiver, as well as other circuitry to decode signals received from scanned tags. The reader is always broadcasting radio signals and waiting to hear back from a tag.

Tags, or cards, contain an antenna, circuitry to convert radio waves to DC power, a radio transceiver, as well as a tiny amount of storage that can be used to hold data specific to that card. When the card receives a radio signal in a specific frequency, the antenna will harvest energy from that signal, and power up the storage chip and the transceiver.

The card's transceiver will then send the contents of its tag back to the reader, which can happen over different distances based of the type of tag. Passive tags do not contain a power source and their distance is generally limited to anywhere between a few millimeters to a couple of inches, based on the design of the tag.

Active tags contain a battery that can be used to boost the tag's transmission distance. Some of these tags can be read from hundreds of meters away, but the tags are much larger than passive tags due to containing a battery, larger antenna, and more radio amplification circuitry.

Since passive tags don't contain a power source or much circuitry, they can be much smaller. Some versions are flat stickers that can be stuck to products in a store, and some are almost as small as a grain of rice and can be implanted under an animal’s skin.

RFID can be used in many applications, but the underlying technology is the same. The reader sends out radio waves, and the tag responds with values stored within the tag.

RFID for Access Control

When used for building access control, a central computer or controller is used to monitor one or more card readers. This controller will also have the ability to unlock doors electronically. This means that the controller is the heart of the access control system.

The controller is where access control permissions are managed. It stores all tag or keycard information, along with what areas or doors that keycard is permitted to access.

The software inside the controller operates much like some of the if/else programs you have created throughout this course. When a keycard is scanned, it's unique value is used to check a list of permissions stored inside the controller. For example:

User A is issued card number 123. Card 123 is allowed to access door 1 and door 2.

User B is issued card number 456. Card 456 is only allowed to access door 2.

If card 123 is scanned at door 1 or 2 then access will be granted. If card 456 is scanned at door 2 then access will be granted, but when scanned at door 1, access will be denied. This is nice because access to areas can be granted or revoked just be changing the permissions in the main controller, instead of handing out individual metal keys, or changing locks.

Security Concerns with RFID

While more convenient, RFID does have some inherent security risks associated with the technology. With a metal key in your pocket there is little to no risk of anyone duplicating a key while it's in your pocket. The same cannot be said for RFID cards.

Although encryption is used to secure data storage and communication on some higher end cards, most cards are not very secure. These less secure cards will respond to any reader that requests its data, given that reader is operating in the right frequency range. The tags in your kit operate in the 13.56MHz range, so they will start transmitting their stored data anytime they receive a 13.56MHz signal.

This is where the problem occurs. Your card doesn't know the difference between the RFID reader in your kit and any other 13.56MHz reader it sees. Imagine that you wired your front door to be conveniently controlled by an RFID lock that only opened for a tag

that sent out the string **`Open Sesame`**. Everything is working great and you no longer have to carry keys, using only the tag to unlock your front door.

Since your tag will respond to any 13.56MHz reader, it's possible that your tag might get scanned by someone hoping to duplicate your tag in order to access your house. Someone close enough to your tag in line at a coffee shop could send a 13.56MHz signal your way, and your tag would respond to them with Open Sesame. They can now duplicate or clone your card and unlock your front door with their copy of your card.

Your tag won't know that it was scanned and the reader on your front door won't know the difference, since it's only looking for the string **`Open Sesame`** to unlock the door. This attack is fairly uncommon, but it is made possible by the convenience of RFID. There are RFID shielding devices available that make this type of attack impossible, but this security does come at the cost of convenience.

These RFID shielding devices completely enclose your card in a material that radio waves cannot penetrate. This shielding can be built directly into wallets, purses, or smaller pouches and can hold a single card or tag. While inside this shielding, your card cannot be read by an attacker, but it also cannot be read by your front door reader either. You would need to remove your card from this shielded pouch in order for the radio waves emitted by the reader to interact with the card, and for the card to send its data back to the reader.

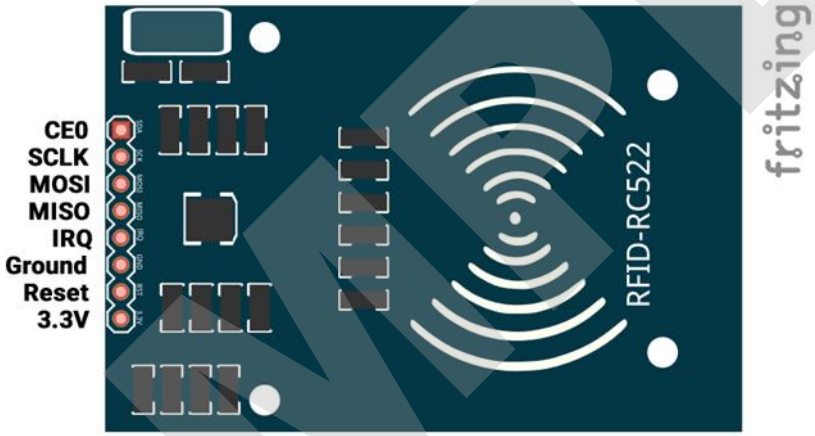
As with everything, there are trade-offs between security, convenience, and cost. The benefits of deploying an RFID system would need to be weighed against the possible cloning and misuse of a card. More secure RFID systems can lessen the risk, if not eliminate, the possibility of these attacks, but this will require a higher-level system with encryption that will come at a higher financial cost.

MFRC522 Tag Reader

The RFID reader included in your kit runs on the MFRC522 chipset which can read and write 13.56MHz tags. The Raspberry Pi requires installation of a special library to communicate with the MFRC522, but once installed, communication with the device is very simple over the SPI bus.

These libraries do not support software SPI, so hardware SPI will be used to communicate with the reader. Using hardware SPI means that specific GPIO pins will be used for this communication, and that a setting in the Raspberry Pi will need to be changed in order to enable hardware SPI. In the activities for this lesson, you will install this library and change the SPI setting once the reader is wired up.

Here is a pinout of the MFRC522 board, as well as signal descriptions:



CE0 – Chip Enable

SCLK – Serial Clock

MOSI – Master Out, Slave In

MISO – Master In, Slave Out

IRQ – Interrupt, not used in Raspberry Pi with hardware SPI

Ground – Connect to Ground

Reset – Will trigger a reset of the MFRC522

3.3V – Connect to 3.3V supply

There are many RFID cards on the market, but not all are compatible with this reader. If you plan to purchase additional cards for other projects, ensure they belong to one of the families listed below:

MIFARE1 S50

MIFARE1 S70

MIFARE Ultralight

MIFARE Pro

MIFARE DESFire

Cards may also list compatibility with ISO14443A, which means they will also work with this reader. If you plan to use a card not listed above, please do your research to ensure it will be compatible with the MFRC522 reader before purchasing.

RFID Tags

Tags can come in many shapes and sizes and can store different amounts of information. The tags in your kit will hold 1KB (one kilobyte) of information. This is not a ton of room, but it is enough to store enough information to identify the card with a reader. Your program can then respond however you would like to the presence of the card.

The tags and library you will be using in this lesson support the reading of two values:

The card's UID value, or Unique ID, is a unique 12 to 13-digit value that is assigned at the time the card is manufactured. This value can be read but cannot be modified.

The card's text value is a 48-character field that can store any data you like. This could be anything from a single letter, number, or character, up to a complex string of all these combined. This value can be read and written by the MFRC522 reader.

Reading Text From the Tag

One thing to note about the text value is that it is 48 characters long, and shorter strings written to the card will be formatted to use all 48 of these characters. If you intend to write the value **`card`** to your tag, only four of the character slots will be used, leaving 44 unused.

These unused values cannot be left empty, so spaces will be used to fill up the rest of the characters available. This means that when trying to write a short value like **`'card'`**, the actual value written to the card will be:

`'card                                                    '`

This wont normally be an issue unless you want your program to act on this value when read back from the card.

This wouldn't normally be a problem unless you want to make an action happen in your program based on this text value:

`'card'` does not equal **`'card                                                    '`**

If **`'card                                                    '`** is read from the text field of the tag, and your program is looking for the string **`'card'`**, then the strings will not match, and the program will not operate as expected.

Leading spaces refer to spaces that occur before the information in your string:

`'     card'`

Trailing spaces refer to spaces that occur after the information in your string:

`'card     '`

Python has a built-in function to get rid of these extra spaces, and you will learn more about it in the next section.

Removing Trailing Spaces in Python

As you found in the last section, sometimes it is necessary to remove leading and trailing spaces, also known as whitespace, from a string. Fortunately, Python makes this easy with the **strip()** command:

```
short_var = long_var.strip()
```

In this example, the string long_var will be stripped of all leading and trailing spaces, and saved as short_var.

```
long_var = '      42 Electronics      '

short_var = long_var.strip()

print(short_var)
```

The **strip()** command will only strip leading and trailing spaces from the string, and not the one between the words. After stripping and being saved as short_var, the print command will print **42 Electronics** to the console, instead of the longer version with extra spaces.

There are also some other variations of the **strip()** command:

lstrip() Removes only leading whitespace from a string

rstrip() Removes only trailing whitespace from a string

You may have noticed that the **rstrip()** command could also be used to eliminate trailing whitespace from the text value read from a card. This is true, but **strip()** will ensure that any extra spaces, whether before or after the value in the string, are removed.

Determining Program Type: Shebang or #!

You haven't seen them in programs yet, but a shebang line can be used at the beginning of a program to help some systems identify what type of program is below. The line will be the very first line of a program, and it will start with the characters #! which are referred to as a shebang. The line of code might look something like this:

```
#!/usr/bin/env python
```

or

```
#!/usr/bin/env python3
```

In Unix-like operating systems, this line can be used to determine the path to locate the program that should be used to execute the program, and which program should be used to run the program. Both examples above specify the path for finding Python as /usr/bin/env, but they specify different versions of Python. The first example specifies Python, but not which version. Without modifications to your system, Raspbian will use Python 2 for a file containing this type of shebang. The second example specifies that the program must run in Python 3.

These lines of code are only used when the Python version is not specified when running the program on the command-line. Starting the program using **`sudo python program.py`** or **`sudo python3 program.py`** will override this line, and it will only be treated as any other comment in the program. This is the same behavior we see in Thonny. The environment setting in Thonny will determine which version of Python is used to run a file, not the shebang line.

This won't normally be an issue for you since programs so far have been run in either Thonny using Python 3, or in the command-line using Python 2. This information can be helpful if you're trying to build a project you found online, and odd errors seem to be popping up.

A shebang including of something like **`#!/usr/bin/env python3`** or **`#!/usr/bin/env python2.6`** means that program was intended to specifically run in that version of Python. Attempting to run a program intended for Python 2.6 using Thonny, where the default is Python 3, will likely result in program errors.

Reading and Writing Tags

In order to read and write tags, you will install a couple of libraries in Activity #1 that will enable your Pi to communicate with the MFRC522. One of the libraries is SPI-Py which is takes care of the communication framework needed for the SPI bus. The second library is MFRC522-python which is used to greatly simplify sending data to and from the MFRC522.

Once these libraries are installed, and the SimpleMFRC522 module is imported into your program, working with the reader becomes very easy:

```
reader = SimpleMFRC522.SimpleMFRC522()
```

This line of code will allow you to refer to the reader as **`reader`** in your program, instead of the much longer name above.

```
id, text = reader.read()
```

This command will read the id number and text value stored on the tag and set them equal to **`id`** and **`text`**. These variables can then be printed or used in other ways throughout your program.

```
reader.write('card')
```

The command above can be used to modify the text value stored on the tag. In this example, **`card`** will be written to the tags text value. The id number field is fixed so there is no command available for changing that value.

That is the extent of the commands that we need to interact with the reader. Using these two commands you can change the text value of a tag, read the id and text value of a tag, and then have your program take any actions you would like, based on the tag data that is presented.

LESSON B-11

ACTIVITY #1:
ADDING THE RFID READER AND SOFTWARE

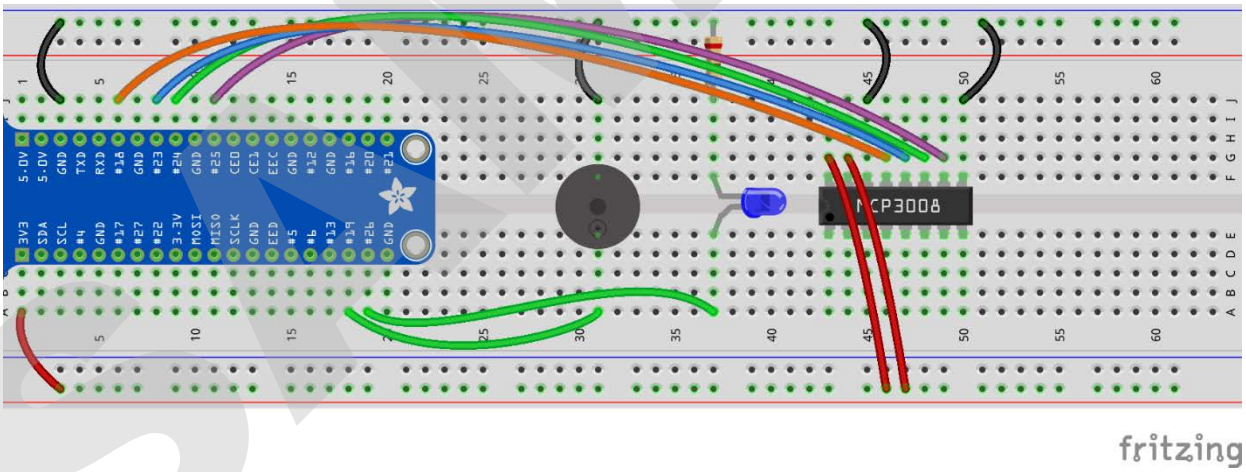
In this activity you will connect the RFID reader and install software that will allow you to read and write tags. The circuit from Lesson B-10, Activity #2 will be used as the starting point for this lesson.

Step #1

Shut down the Pi and disconnect power before proceeding.

To make room for the new components, the phototransistor, potentiometer, and any associated components will need to be removed from the breadboard.

Using the circuit from Lesson B-10, Activity #2 as a starting point, remove the phototransistor, potentiometer, resistor, and any associated jumper wires. The circuit should now look like this:

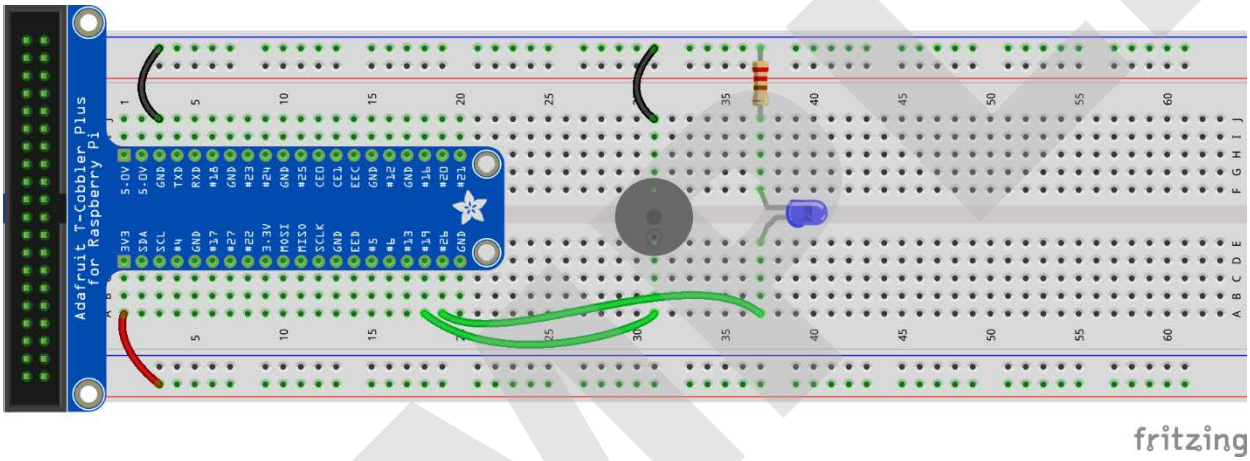


Step #2

The MCP3008 will also need to be removed from the breadboard. Exercise a lot of caution when removing the IC from the breadboard. If the IC comes out of the board unevenly it can cause the pins to bend, and they may break when straightened.

Gently inserting a small, flat screwdriver under alternating sides of the IC is the safest way to remove the IC.

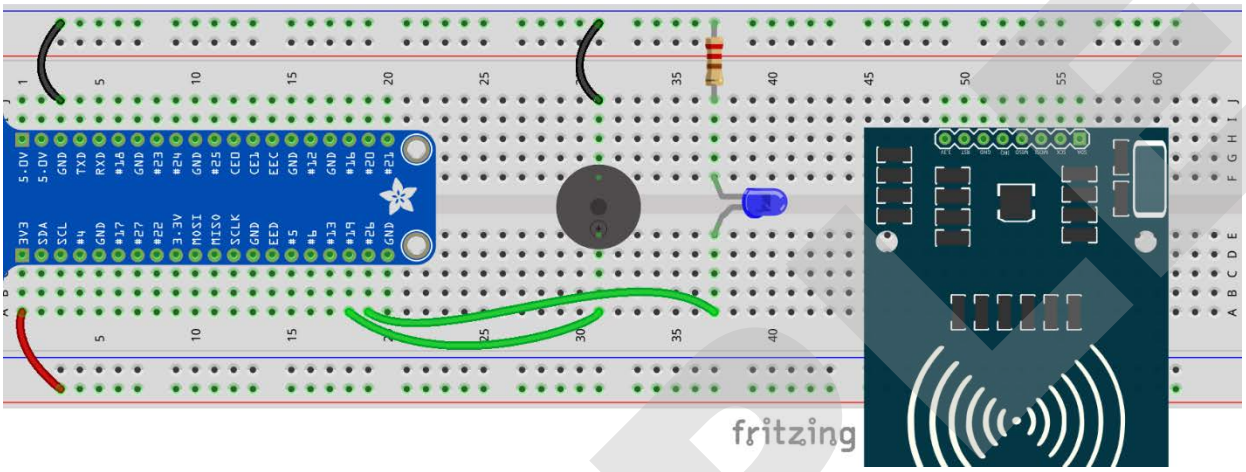
Remove the IC and associated jumper wires from the breadboard. If you're at all unsure on the best way to do this, watch the short video on the Level B Resource Page before attempting to remove the IC. The breadboard should now look like this:



Step #3

Now that you have room on the breadboard, it's time to install and connect the reader.

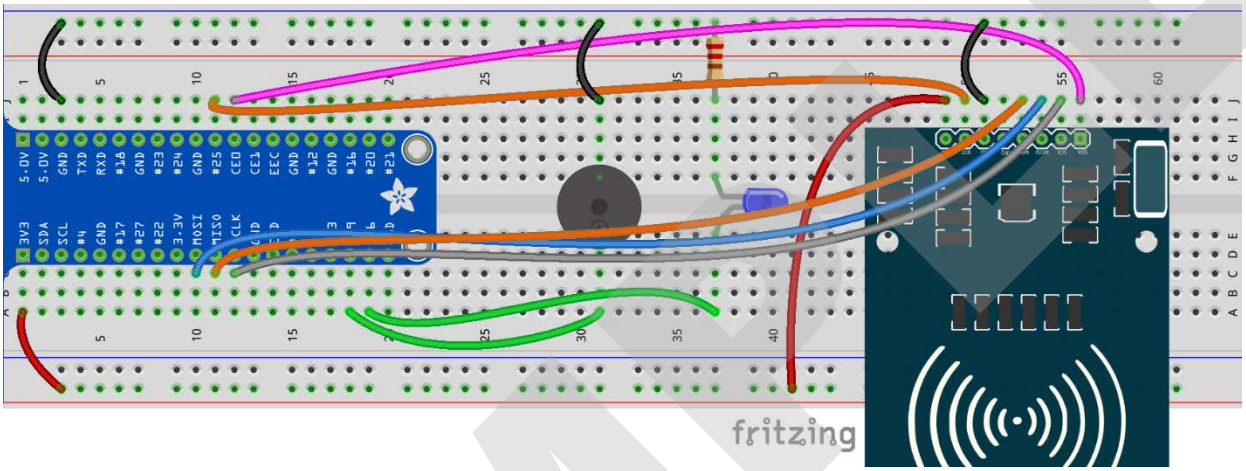
Install the 8-pin connector of the reader into H49 through H56. The reader should be oriented such that the main body of the reader is above covering columns A through H.



Step #4

The reader is now ready to be connected to the wedge. Make the following connections between the reader and the wedge:

3.3V – J49 to P1-41	MISO – J53 to C11
Reset – J50 to J11	MOSI – J54 to C10
Ground – J51 to N2-51	SCLK – J55 to C12
No connection – J52	SDA – J56 to J12



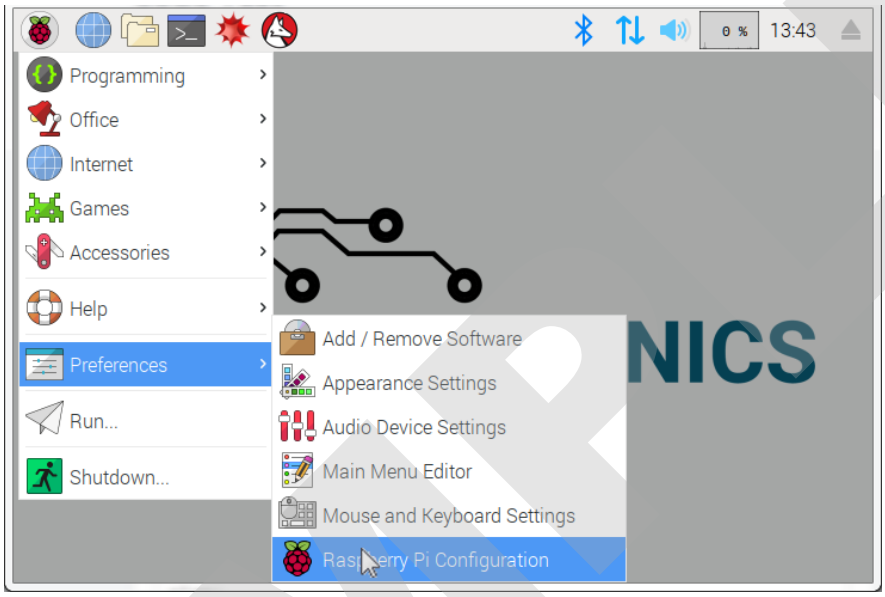
Double-check all connections with this photo before proceeding to the next step.

Step #5

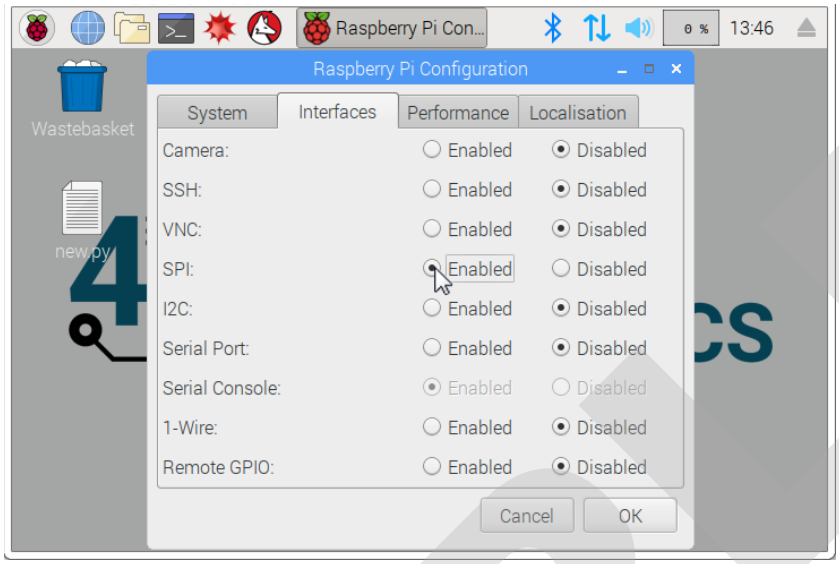
Now that the reader is connected it's time to power on the Pi and enable hardware SPI.

Power on your Raspberry Pi.

Once it's up and running click on the raspberry in the top-left corner, select Preferences, and then select Raspberry Pi configuration from the bottom of the list.



Once inside the configuration utility, select the Interfaces tab, and select the Enabled radio button next to SPI. This will turn on hardware SPI in the Raspberry Pi.

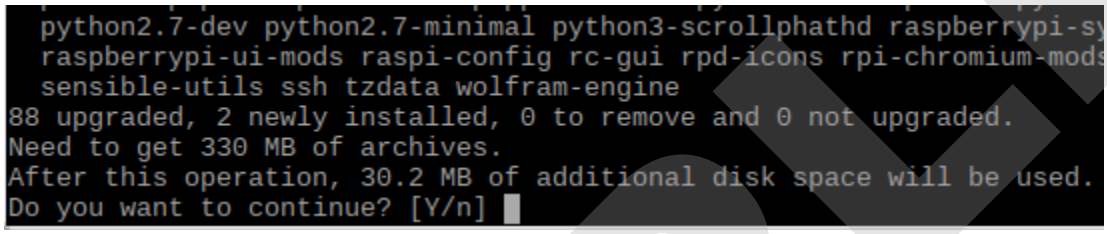


Reboot your Raspberry Pi to allow the SPI setting change to take effect.

Step #6

Before installing the required libraries, make sure your Pi is fully updated so the libraries will have access to the latest versions of the Raspberry pi software packages.

Open a Terminal window by clicking the terminal button in the top menu bar. Once open, use the command **`sudo apt-get update`** to ensure your Pi knows the latest version numbers of all packages. Once that completes, run **`sudo apt-get upgrade`** to upgrade any required packages to the latest version. Answer **`y`** to any questions about free disk space that will be consumed by the upgrades.



Step #7

Now that your Pi is fully updated, it's time to clone the required libraries from GitHub.

In your existing Terminal window, type the following commands, pressing enter after each command:

First, ensure you are still located in the /home/pi directory:

```
cd ~
```

Next, clone SPI-Py from the 42 Electronics GitHub repository:

```
git clone https://github.com/42electronics/SPI-Py.git
```

Now change directories into SPI-Py:

```
cd SPI-Py
```

The last step is to execute the setup.py install script for python3:

```
sudo python3 setup.py install
```

SPI-Py will now be installed for use in the Python 3 environment, which you can run from within Thonny. If you encounter any errors during this process, start over from the beginning of this step.

Step #8

The last library to clone will be MFRC522-python. This library does not require the install step, just cloning from GitHub.

In your existing Terminal window, type the following commands, pressing enter after each command:

First, move yourself back to the /home/pi directory:

```
cd ~
```

Next, clone MFRC522-python from the 42 Electronics GitHub repository:

```
git clone https://github.com/42electronics/MFRC522-python.git
```

If you encounter any errors during this process, start over from the beginning of this step.

LESSON B-11

ACTIVITY #2:
READING AND WRITING TAGS

In this activity you will connect the read information from and write information to the tags included in your kit.

Step #1

You now have a local copy of the MFRC522-python repository.

Open File Manager by clicking on the folder icon in the top menu bar. File manager will open in the /home/pi directory. Double-click on the MFRC522-python directory to view its contents.

Some example programs called read.py and write.py have been included to allow you to quickly begin reading and writing tags.

Double click the file named **read.py** and it will open in Thonny:

read.py

```
#!/usr/bin/python3

import RPi.GPIO as GPIO
import SimpleMFRC522

reader = SimpleMFRC522.SimpleMFRC522()

try:
        id, text = reader.read()
        print(id)
        print(text)
finally:
        GPIO.cleanup()
```

Run **read.py** by clicking the run button in Thonny. Place the card near the reader and its id number and text value will be displayed in the console output.

The text field will be represented by 48 squares because this field is completely empty from the factory.

Scan the blue tag to ensure it scans properly, and that it displays a different id value than the card.

Shell

```
>>> %Run read.py

  648266784720
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪

>>>
```

Step #2

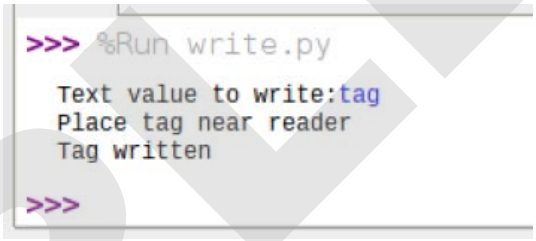
The cards can be read but you need to fix the empty text value fields by writing new data to the tags.

Navigate back to the File Manager and double-click on write.py. Write.py will open as a new tab in Thonny.

Make sure there are no tags near the reader when you run write.py. The program will write any tag within range and you want to make sure the tags get programmed correctly, so you can write a program in Activity #3 that will recognize these values.

You will now write the text value of **`card`** to the white card.

With no tags near the reader, run write.py in Thonny. In the Shell window of Thonny, enter **`card`** as the value to be written to the tag, and press enter. Place the white card near the reader when prompted, the tag will be written, and **`Tag written`** will be printed to the Shell for confirmation.



If any errors occur, or the program does not run as expected, stop write.py in Thonny and run it again. Do not proceed to the next step until the white card has successfully been written with the text value **`card`**.

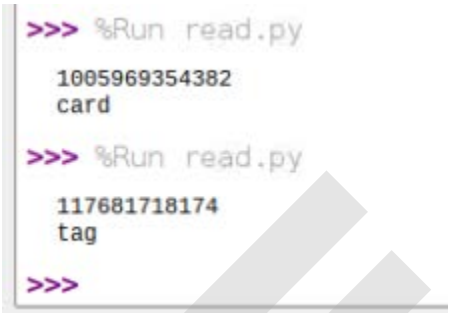
Step #3

Now that the card has been written you can work on the blue tag. Make sure to hold the metal keyrings on the tag to keep them from coming into contact with any metal connections on the circuit board of the reader.

Using the same process in the last step, write a text value of **`tag`** to the blue tag.

Step #4

Confirm the tags were properly written by reading the values back.



Switch tabs in Thonny back to read.py and run that program. Scan the white card to ensure it reports its ID number and the text string **`card`** when scanned.

Run the program again and scan the blue tag to ensure it reports its ID number and the text string **`tag`** when scanned.

You now know that the text values loaded correctly and now you can create a program that can make decisions based on those values.

Note:

The RFID reader's proper operation relies on good connections to power, ground, and data lines on the Raspberry Pi. Poor or intermittent connections can cause the reader not to operate properly. If your reader stops reading or writing for no reason, carefully remove and reinstall each of the jumper wire connections in J49 through J56. If the problem with your reader is due to a bad jumper wire connection, this should fix the problem.

The reader will read a tag very quickly, but if you try to swipe a tag by the reader extremely fast it is possible to pull the tag away from the reader before it's finished reading. This will result in a card read error that might look something like this:

```
>>> %Run read.py

  AUTH ERROR!!
  AUTH ERROR(status2reg & 0x08) != 0
  1005969354382
```

If this happens, just scan the tag again, more slowly, and everything should work as expected.

LESSON B-11

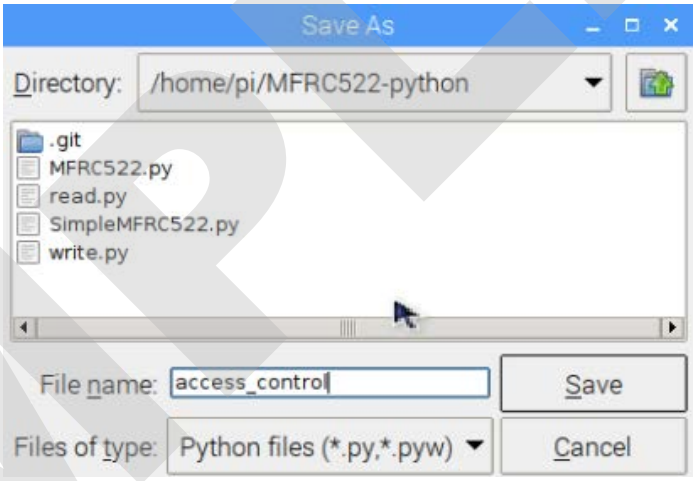
ACTIVITY #3:
CREATING AN ACCESS CONTROL PROGRAM

In this activity you will create a program that reads the text values from scanned tags, and prints a message letting the user know if access is granted, or not.

Step #1

Use the **`read.py`** program as a starting point, as it already contains everything needed to read tags.

With read.py open in Thonny, select **`File`**, and then select **`Save as`** from the dropdown menu. Enter a file name of **`access_control`** and click the **`Save`** button:



Note:

Programs that need to read RFID tags must be located inside the MFRC-Python directory, as they will need direct access to the SimpleMFRC522.py and MFRC522.py files used to communicate with the reader. Attempting to run programs that require access to these communication files from anywhere else, will result in Python errors.

Step #2

You now have a copy of **read.py** saved as **access_control.py** that can be modified without affecting the original file. The program is currently running a **try:** loop until it sees a tag, prints the **id** and **text** values, and runs a **GPIO.cleanup()** before exiting. Only one tag can be read before the program automatically exits. Let's add a **while True:** loop inside the **try:** loop to keep the program reading tags until you exit the program.

Add a **while True:** loop just below the **try:** loop. The addition is highlighted below:

```
reader = SimpleMFRC522.SimpleMFRC522()

try:
    while True:
        id, text = reader.read()
        print(id)
        print(text)
        time.sleep(.3)

finally:
        GPIO.cleanup()
```

Run the program and read the card and tag.

You will notice the program is now reading without exiting after each tag, but it's reading one tag multiple times because this program can loop very quickly, and the tag might be near the reader during more than one loop.

Another problem with the program is that there is no longer away to exit the program gracefully. Press the stop button in Thonny or CTRL-C to stop the program. This will result in errors because the program was busy communicating with the reader when the program was ended. You will fix the looping and exit problems in the next step:

Step #3

It's time to fix the duplicate read and exit issues that we created in the last step. You will add a **sleep** command to slow down the loop and modify the **finally:** command to catch keyboard exceptions.

Import the **time** module at the beginning of the program and add a **time.sleep(.3)** to the end of the loop. Also, change **finally:** to **except KeyboardInterrupt:** which will allow manually ending the program to trigger the **GPIO.cleanup()**. Additions and changes are highlighted below:

```
#!/usr/bin/python3

import RPi.GPIO as GPIO
import SimpleMFRC522
import time

reader = SimpleMFRC522.SimpleMFRC522()

try:
    while True:
        id, text = reader.read()
        print(id)
        print(text)
        time.sleep(.3)

except KeyboardInterrupt:
        GPIO.cleanup()
```

Run the program.

It can now scan tags reliably without duplicating reads and pressing stop in Thonny or CTRL-C will not generate errors, as **GPIO.cleanup()** is being triggered before the program exits.

Step #4

Now that the program is reading cards without exiting, it needs to strip the trailing whitespace from the text value that's being read from the card. To do this you will run the **`strip()`** command on the value of text that's read from the card.

Replace the two existing print statements with a **`strip()`** command that will strip trailing whitespace from **`text`** and save it as the new value of **`text`**. Changes highlighted below:

```
try:
    while True:
        id, text = reader.read()
        text = text.strip()
        time.sleep(.3)
```

The id number and text value will be read from the card and the text value will be stripped down to either **`'card'`** or **`'tag'`** based on the tag that was scanned.

Step #5

You can now add some if statements to check which tag is being read and print whether the tag is granted or denied access. Go ahead and grant access to the card, but deny access to the tag, by adding two if statements.

Add two if statements directly under the **strip()** command that will check the value of text and print **ACCESS GRANTED** or **ACCESS DENIED** based on which tag is read:

```
try:
    while True:
        id, text = reader.read()
        text = text.strip()
        if text == 'card':
            print('ACCESS GRANTED')
        if text == 'tag':
            print('ACCESS DENIED')

        time.sleep(.3)
```

Run the program and read both tags. The access messages will be printed each time a card is scanned to indicate ACCESS GRANTED or ACCESS DENIED.

This code is being kept very simple to illustrate the concept, but the print sections of the if statements in this program could be replaced with anything you like. By including GPIO pin setups at the beginning of your program you could light an LED when access is granted, play a noise through the piezo speaker when access is denied, or any combination of print statements and GPIO events that you might want.

In Lesson B-12, you will add more advanced program functionality, and circuitry that will allow for additional notification options.

- Leave this circuit built as it will be used as a starting point in Lesson B-12.
- Save the program to use as a starting point in Lesson B-12.

LESSON B-11

QUESTIONS FOR UNDERSTANDING

1. Is hardware SPI turned on by default on the Raspberry Pi, or does it require a menu change and reboot to become enabled?

2. What is the Python command that removes leading and trailing whitespace from a string?

3. Do all RFID tags work with all RFID readers?

Answers Can be Found on the Next Page

Answers

1. **Is hardware SPI turned on by default on the Raspberry Pi, or does it require a menu change and reboot to become enabled?**

   ANSWER: Hardware SPI requires a menu change and a reboot to be enabled.

2. **What is the Python command that removes leading and trailing whitespace from a string?**

   ANSWER: To remove both leading and trailing spaces, use the **`strip()`** command.

3. **Do all RFID tags work with all RFID readers?**

   ANSWER: No, not all readers and tags are compatible. It is important to check for compatibility between RFID readers and RFID tags.

LESSON B-18

RANGE SENSING GAME

Lesson Overview

- ► Level Shifting with Resistors
- ► Absolute Value in Python
- ► Modifying a File for Import Use

Concepts to Review

- ► Slide Switch (Lesson B-5)
- ► Voltage Dividers (Lesson B-9)
- ► Level Shifting Integrated Circuit (Lesson B-13)
- ► Ultrasonic Range Sensor (Lesson B-14)
- ► Running Modules as Imports vs. Directly (Lesson B-15)
- ► OLED Display (Lesson B-16)
- ► Capacitive Touch Sensors, GPIO Pin Level Sensing (Lesson B-17)

Materials Needed

- ❑ Raspberry Pi connected to Monitor, Keyboard, and Mouse
- ❑ Assembled Circuit from Lesson B-17
- ❑ Ultrasonic Range Sensor x1
- ❑ 1K-Ohm Resistors x2
- ❑ Long Male-to-Male Jumper Wires x2
- ❑ Short Male-to-Male Jumper Wires x5

LESSON B-18

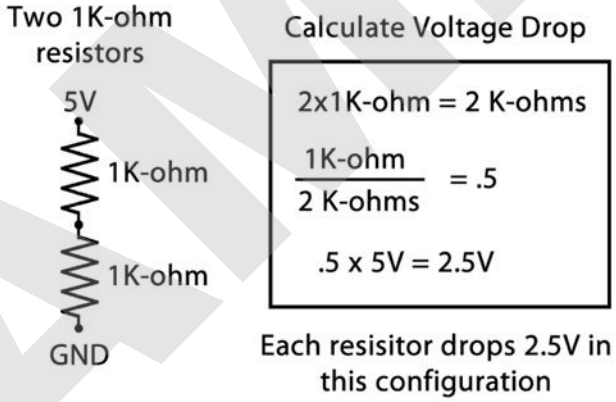
RANGE SENSING GAME

In this lesson you will learn how to use a voltage divider to do simple signal level shifting, the absolute value function, and how to modify a file so it can be used as an import for another program. You will then move on to build the final project.

Level Shifting with Resistors

In Lesson B-9 you learned how voltage dividers can be used to create new voltage levels. This same principle can also be used for level shifting from a higher voltage to a lower voltage.

The following voltage divider with two equal value 1K-ohm resistors will cut the supplied 5V signal in half:



The great thing about this circuit is that the 5V signal does not have to come from a constant supply. It can also be connected to the output from a 5V device, like the Ultrasonic Range Sensor, whose 5V output is not safe to connect directly to a GPIO pin. By using the output of the Ultrasonic Range Sensor to supply the input voltage, the output will be 2.5V which is safe to connect to a GPIO pin.

By watching the voltage divider output pin with the GPIO, you can determine if the 5V signal is present or not:

- 2.5V present means the Ultrasonic Range Sensor is applying 5V to the input
- 0V present means the Ultrasonic Range Sensor is not applying 5V to the input

You learned in Lesson B-17 that a GPIO pin will register as high for any voltage above 1.4V. So a 2.5V level will easily trigger a high in the GPIO pin:

- GPIO high means the Ultrasonic Range Sensor is applying 5V to the input
- GPIO low means the Ultrasonic Range Sensor is not applying 5V to the input

Using a voltage divider can be a very useful way to quickly reduce the voltage from a sensor, using only two resistors. When using several 5V sensors, it makes more sense to use a device like the 74LVC245 (Level Shifting) IC instead, due the amount and complexity of resistors that would be required to make a voltage divider for each channel.

Absolute Value in Python

An absolute value represents how far a value is away from zero. It essentially removes the positive or negative sign of a value, so the absolute value of -5 is 5. The same goes for the positive value. The absolute value of 5 is still just 5. The absolute value function in Python is:

```
abs()
```

The absolute value of will be taken of anything inside the parentheses:

`abs(-23)` becomes **`23`**

`abs(42)` becomes **`42`**

The second example doesn't seem to be very useful, but what if you don't know what the value in the parentheses will be until your program starts running:

```
abs(x-y)
```

If x is greater than **`y`** then the **`abs()`** function will have no effect on the value that this function outputs. However, if **`y`** is greater than **`x`**, the result of **`x-y`** will be negative and **`abs()`** will strip off the sign from the negative number.

In the upcoming activity, you will build a game that will be taking the difference of two numbers, without knowing which will be larger. The **`abs()`** function will be used to strip the sign from the result, leaving only the positive value of the difference between the two values.

Modifying a File for Import Use

When you import a file, that file is completely run from beginning to end. Any imports, variable assignments, function definitions, or anything else in that program, will run as if those lines were included in your program.

What if a file that you choose to import includes the following:

```
def thing1():
    x = 1

while True:
    print('Hello World')
```

On import, the imported file would define the function named **`thing1()`**, and then get stuck in the **`while True:`** loop, never returning to your program. This is obviously not ideal, and this is why back in Lesson B-15, you learned about the **`__name__`** variable that can be used to determine if a file has been imported, or has been executed directly:

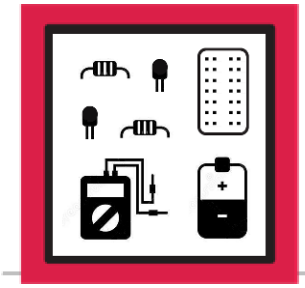
```
if __name__=="__main__":
```

Any code indented below this **`if`** condition will only run when the file was run directly and will be ignored if the file is being executed as an import to another file. You can modify the earlier example to allow for both imported and direct execution:

```
def thing1():
    x = 1

if __name__=="__main__":
    while True:
        print('Hello World')
```

Importing the file will now result in the code inside the if block being completely ignored, and when the program is executed directly, it will run from top to bottom, including everything contained in the **`if`** block.

In the following activities, the ultrasonic range sensing program that you created in Lesson B-14 will be modified so its function definitions can be used for import, without executing the main program it contains.



LESSON B-18

ACTIVITY #1:
MODIFYING ULTRASONIC.PY

In this activity, you will modify the **`ultrasonic.py`** program that you created in Lesson B-14 so its range sensing functions can be imported without running the main loop program that it contains.

Step #1

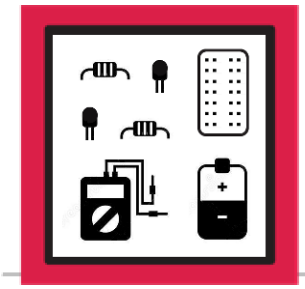
The first step is to open **`ultrasonic.py`** from your Desktop in Thonny. Once open, scroll down to the **`try:`** loop. You want to enclose this entire **`try:`** loop inside of an **`if __name__`** condition to ensure that it does not run when the file is used as an import. Add the following line above the **`try:`** block and add another level of indentation to the **`try:`** block as well as everything below, including the **`except:`** block:

```
if __name__=="__main__":
    try:
        while True:
            distance = average()
            print('%.1f' % distance)
            if distance < 20:
                GPIO.output(red, GPIO.HIGH)
                GPIO.output(green, GPIO.LOW)
                GPIO.output(blue, GPIO.LOW)
            elif 20 <= distance < 25:
                GPIO.output(red, GPIO.LOW)
                GPIO.output(green, GPIO.HIGH)
                GPIO.output(blue, GPIO.LOW)
            else:
                GPIO.output(red, GPIO.LOW)
                GPIO.output(green, GPIO.LOW)
                GPIO.output(blue, GPIO.HIGH)
            time.sleep(.25)

    except KeyboardInterrupt:
        GPIO.cleanup()
```

All of the gray boxes above are additional spaces that were added to realign the code below the new **`if`** block.

Step #2

Save your updated file so you can use it as an import later when building the game program in Activity #3.



LESSON B-18

ACTIVITY #2:
ADDING THE ULTRASONIC RANGE SENSOR

In this activity, you will add the ultrasonic range sensor to the capacitive touch circuit you built in Lesson B-17, Activity #3.

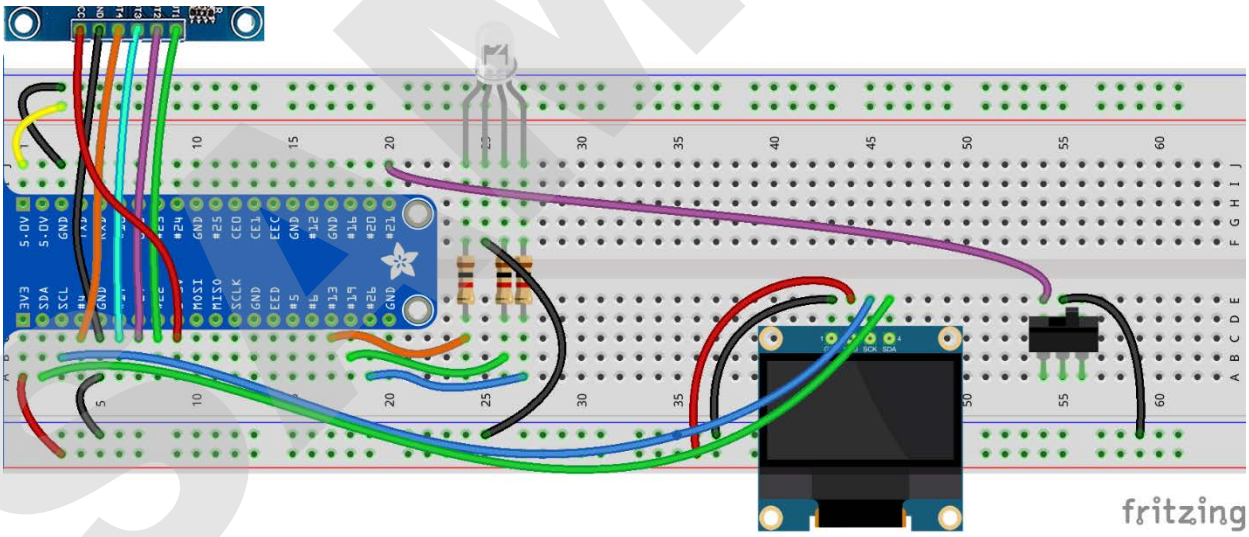
Step #1

Shut down the Pi and disconnect power before proceeding.

Once that's done, the first circuit modification will be to get 5V power and ground over to the P2/N2 power rails so it can be used to power the Ultrasonic Range Sensor. Make the following two connections using short jumper wires:

Short jumper wire – 5V – between J1 and P2-3

Short jumper wire – GND – between J3 and N2-3



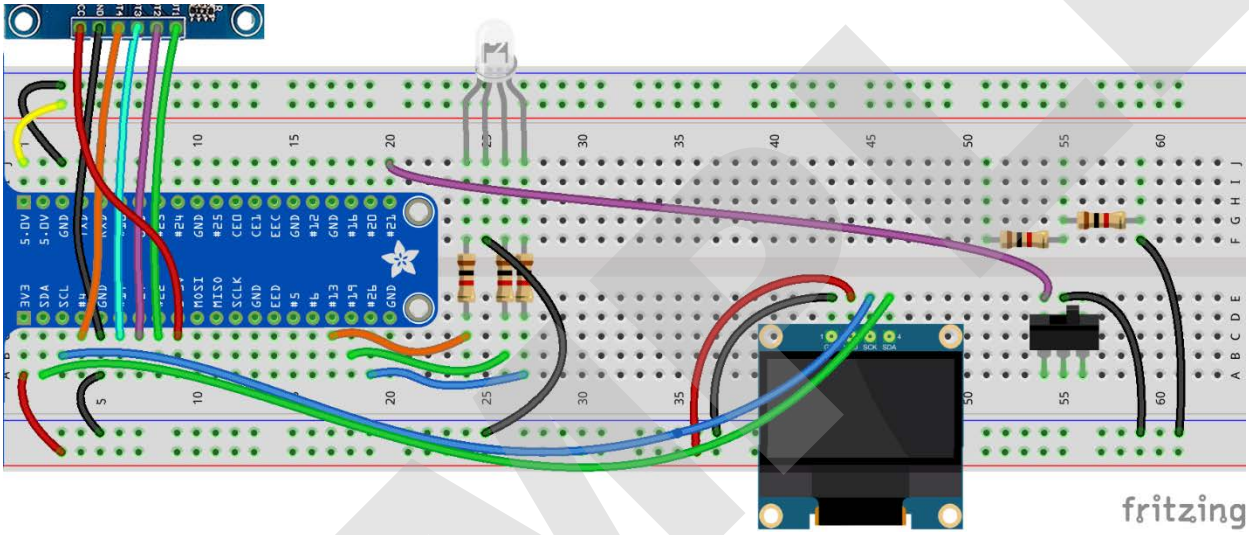
Step #2

Next, add a voltage divider made of two 1K-Ohm resistors that will be used to level-shift the Echo output of the ultrasonic range sensor. Add two resistors and a short jumper wire between the points below:

1K-Ohm resistor – between G55 and G59

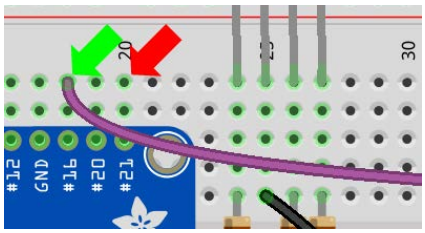
1K-Ohm resistor – between F51 and F55

Short jumper wire –GND - between F59 and N2-61



Step #3

The ultrasonic.py file that will be used to capture distance readings will be looking for echo signals on GPIO21, but the slide switch is currently in that position. Move the slide switch connection from GPIO21 over to GPIO16:



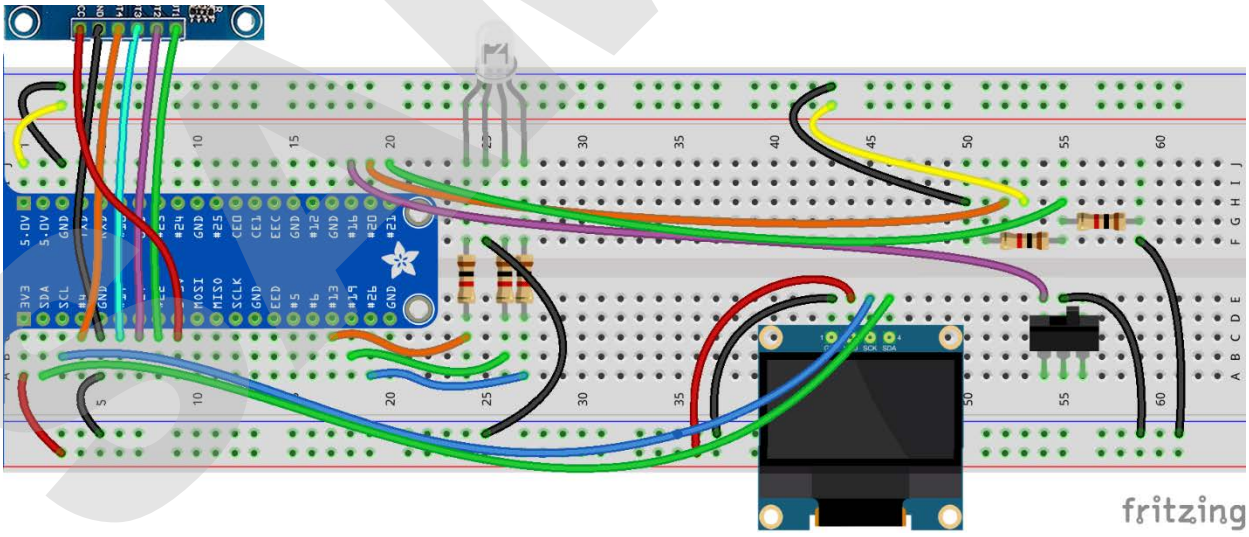
The voltage divider will take care of bringing the Echo line from the ultrasonic range sensor down to a safe level, but you will need to make a few more connections before you can use it for ranges. Make the following four connections between the points listed below:

Short jumper wire – 5V – between H53 and P2-43

Short jumper wire - GND – between H50 and N2-42

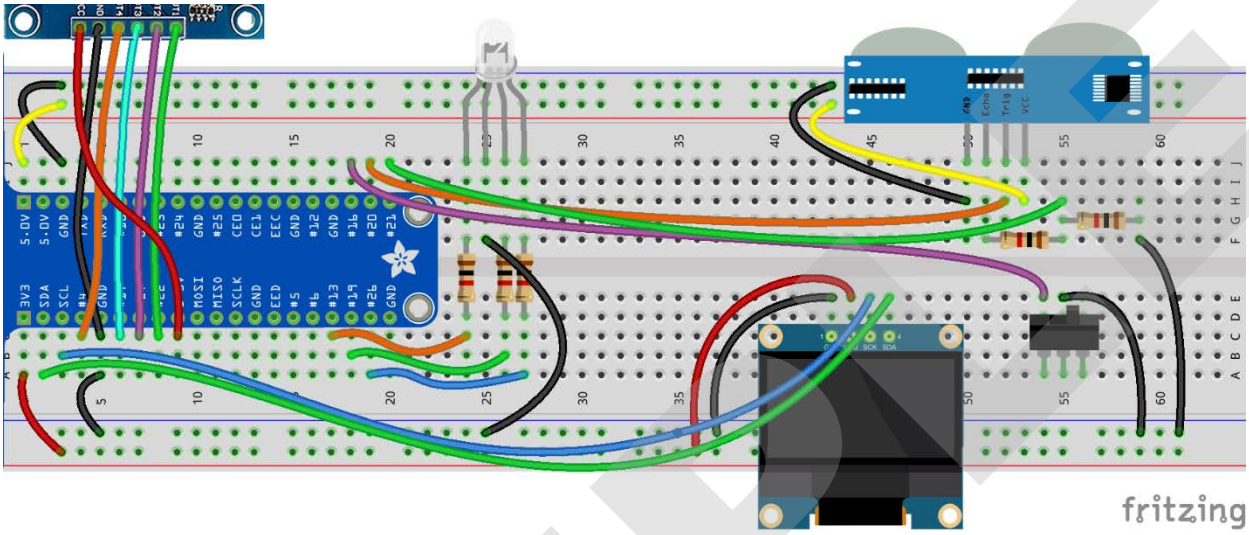
Long jumper wire – Trigger – between H52 and J19

Long jumper wire – Echo – between H55 and J20

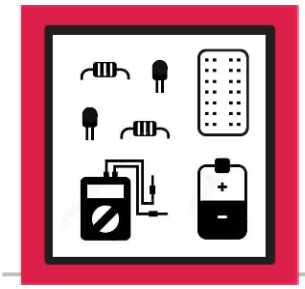


Step #4

The last step is to add the ultrasonic range sensor. Connect it to the breadboard at J50 through J53 with the sensor pointing away from the OLED display. Ensure the sensor is properly oriented and connected to the correct locations, or it could be damaged:



Power the Raspberry Pi on so it can be used to create a program to use with your new circuit.



LESSON B-18

ACTIVITY #3:
BUILDING THE RANGE GAME

In this activity, you will build a program to create the game outlined below:

The game will ask you to place your hand or another obstacle at a pre-determined distance from the Ultrasonic Range Sensor. The game will consist of the following actions:

- The user selects the desired difficulty level using the slide switch. Easy mode allows for 20cm of error while Hard mode only allows for 10cm. The program waits 2 seconds before the position of the slide switch determines which difficulty level will be used for that round.
- The screen displays a random target distance between 20cm and 100cm.
- The user is asked to press capacitive touch pad 1-4 to start the distance capture process. Each pad represents the number of seconds before the capture occurs, which can be used to add more difficulty. Less time to get ready before the capture makes it more difficult.
- The user range is captured and compared to the target distance. The RGB LED will turn green if the user distance was within 20cm in Easy mode, or 10cm in Hard mode. Errors above these amounts will result in the RGB LED turning red.
- The program loops back to the beginning.

This program will be the largest you've written yet, but it will borrow large blocks from programs in previous activities. So, don't be intimidated! Build the program one block at a time, just like you would with a shorter program.

Step #1

The first step will be to open Thonny and create a new program called **`range_game.py`**. Save the file to your Desktop so it will have access to the **`ultrasonic.py`** file that you modified in Activity #1.

As you go through the steps below, save your program often to avoid losing an of your work.

Step #2

The first area in the program will be the imports and there are quite a few. You will be using **time**, **RPi.GPIO**, **random**, **ultrasonic**, **Adafruit_SSD1306**, and the **PIL** imports you used for the OLED display. Here are the import lines to add:

```
import time, RPi.GPIO as GPIO, random, ultrasonic, Adafruit_SSD1306
from PIL import Image, ImageDraw, ImageFont
```

Step #3

Next, you will assign the input and output pins using a lot of variables.

This will help you later if you want to modify this program and move an input or output to another pin. This list will contain pin assignments for the slide switch, RGB elements, and Capacitive Touch inputs.

The lists for **rgb** and **cap** will allow those groups of pins to be configured as inputs and outputs as a group, using only one line each:

```
import time, RPi.GPIO as GPIO, random, ultrasonic, Adafruit_SSD1306
from PIL import Image, ImageDraw, ImageFont

slide = 16
rgb = [13,19,26]
red = 13
green = 19
blue = 26
cap = [22,27,17,4]
cap1 = 22
cap2 = 27
cap3 = 17
cap4 = 4
```

Step #4

The next step will be the GPIO configuration. Here are the configuration steps that need to be completed:

- Set GPIO pin mode to **BCM**
- Configure **slide** as an input with pull-up
- Configure **cap** as an input (this will take care of all four touchpads)
- Configure **rgb** as an output (this will take care of all three RGB elements)
- Set the output of **rgb** to low or 0 (this will ensure the RGB LED is off initially in case your program has errors and exits improperly)

Here is the code to accomplish these five tasks:

```
cap2 = 27
cap3 = 17
cap4 = 4

GPIO.setmode(GPIO.BCM)
GPIO.setup(slide, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(rgb, GPIO.OUT)
GPIO.setup(cap, GPIO.IN)
GPIO.output(rgb, GPIO.LOW)
```

Step #5

There is one more configuration step that must be accomplished but it can't be included with the GPIO configuration. It's the code to configure all the settings required for your OLED display to operate.

This block of code is pulled directly from the program you created in Lesson B-16, Activity #2, and each line of code is broken down in the section titled SSD1306 Display Driver. If you're unsure about anything below, please refer to that section for more information:

```
GPIO.setup(cap, GPIO.IN)
GPIO.output(rgb, GPIO.LOW)

disp = Adafruit_SSD1306.SSD1306_128_64(rst=None)
disp.begin()
width = disp.width
height = disp.height
image = Image.new('1', (width, height)) # '1' converts image to 1-bit color
draw = ImageDraw.Draw(image)
font = ImageFont.truetype('/usr/share/fonts/truetype/freefont/FreeSans.ttf',18)
```

Step #6

There are a couple lines of code that will be used every time the display needs to be updated. They are **`disp.image(image)`** and **`disp.display()`**.

Instead of using these two lines every time you need to update the display, create a function named **`update_display()`** that can be called every time an update is needed:

```
draw = ImageDraw.Draw(image)
font = ImageFont.truetype('/usr/share/fonts/truetype/freefont/FreeSans.ttf',18)

def update_display():
    disp.image(image)
    disp.display()
```

You are now just over 30 lines into the program. Ensure your program matches the program on the next page before continuing to the next step.

```
import time, RPi.GPIO as GPIO, random, ultrasonic, Adafruit_SSD1306
from PIL import Image, ImageDraw, ImageFont

slide = 16
rgb = [13,19,26]
red = 13
green = 19
blue = 26
cap = [22,27,17,4]
cap1 = 22
cap2 = 27
cap3 = 17
cap4 = 4

GPIO.setmode(GPIO.BCM)
GPIO.setup(slide, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(rgb, GPIO.OUT)
GPIO.setup(cap, GPIO.IN)

disp = Adafruit_SSD1306.SSD1306_128_64(rst=None)
disp.begin()
width = disp.width
height = disp.height
image = Image.new('1', (width, height))
draw = ImageDraw.Draw(image)
font = ImageFont.truetype('/usr/share/fonts/truetype/freefont/FreeSans.ttf',18)

def update_display():
    disp.image(image)
    disp.display()
```

Step #7

Now that all the configuration steps have been completed, it's time to start building the main program loop. Start with a **`try:`** loop that contains a **`while True:`** loop, and an **`except KeyboardInterrupt:`** that contains the following:

disp.clear()	clears the image buffer
disp.display()	pushes the image buffer to display to so it's clear before shutdown
GPIO.cleanup()	resets all GPIO pins to their default state

The code to add these elements is listed below:

```
def update_display():
    disp.image(image)
    disp.display()

try:
    while True:

except KeyboardInterrupt:
    disp.clear()
    disp.display()
    GPIO.cleanup()
```

Any new code in upcoming steps will be inserted in the **`while True:`** loop.

Step #8

The first step in the main program loop is to display the difficulty level based on input from the slide switch. You will control the timing of this portion of the program by looping 20 times with a 0.1 delay in each loop, for a total time of 2 seconds allowed for difficulty selection.

Inside this **skill_selection** loop, you will nest two **if/else** conditions:

If the slide switch is low or **False**, then draw a rectangle to clear the display, draw the strings **"Difficulty?"** and **"Easy"** to two lines of the image buffer, push the image buffer to the screen, and set a variable named **skill** equal to **2**. This variable will be used later to determine the size of the error window that will result in a green LED.

If the slide switch is not low, the **else:** block will execute, just like the block above but **"Hard"** will be printed to the second line of the display, and skill will be **set** equal to **1**.

Here is the code to accomplish the program functions outlined above:

```
try:
    while True:
        for skill_selection in range(0,20):
            if GPIO.input(slide) == False:
                draw.rectangle((0,0,width,height), outline=0, fill = 0)
                draw.text((0, 0), "Difficulty?",  font=font, fill=255)
                draw.text((0, 22), "Easy",  font=font, fill=255)
                update_display()
                skill = 2
            else:
                draw.rectangle((0,0,width,height), outline=0, fill = 0)
                draw.text((0, 0), "Difficulty?",  font=font, fill=255)
                draw.text((0, 22), "Hard",  font=font, fill=255)
                update_display()
                skill = 1
            time.sleep(.1)
```

Make sure your indentation matches the code above. This is crucial for each block of the program to operate as expected.

Step #9

Now that the difficulty has been selected, the random target number can be selected and displayed on the screen to let the player know the target distance for this round. First, set a variable named **`target`** to a random integer between 20 and 100 by using the **`randon.randint()`** function.

Next, show the target distance on the display by using % notation to print **`'Target = %s" %target`** on the first line of the display. The second and third lines of the display should read **`"Press pad"`** and **`"to start"`**. An **`update_display()`** will be used to send this new information to the display:

```
                update_display()
                skill = 1
            time.sleep(.1)

        target = random.randint(20,100)

        draw.rectangle((0,0,width,height), outline=0, fill=0)
        draw.text((0, 0), "Target = %s" %target,  font=font, fill=255)
        draw.text((0, 22), "Press pad",  font=font, fill=255)
        draw.text((0, 44), "to start",  font=font, fill=255)
        update_display()
```

Be careful again with the indentation on this section of the program. It should be at the same indentation level as the **`for:`** loop. You want it to run after, and not as part of the **`for:`** loop.

Step #10

The user has now been informed of the target distance and has been prompted to press a touchpad to start the capture process. Now hold the program in a **`while:`** loop until a touchpad is pressed and goes high.

This can be done by using a **`while:`** loop that requires **`cap1`**, **`cap2`**, **`cap3`**, and **`cap4`** all to be **`False`** to keep the loop running. As soon as any of the GPIO pins connected to those pads goes high, the loop will exit and continue with the rest of the program. A **`time.sleep`** of **`0.05`** will be added inside the loop to keep the program from using too many resources while waiting for input from a touchpad:

```
        draw.text((0, 44), "to start",  font=font, fill=255)
        update_display()

        while GPIO.input(cap1) == GPIO.input(cap2) == GPIO.input(cap3) ==
GPIO.input(cap4) == False:
            time.sleep(0.05)
```

Even though the **`while:`** condition didn't fit on one line in this document, it should be one continuous line in your code from **`while`** all the way to **`False`**:. The **`time.sleep(0.05)`** should be indented so it runs each time the **`while`** condition is met.

Step #11

If the **`while:`** loop stops running, that means one of the capacitive touch pads had been triggered, but you don't know which one.

Create some **`if`** conditions following the **`while:`** loop that will check to see if each pad is high or **`True`**, and assign a variable named **`delay`** equal to that pad's number. If **`cap1`** was pressed then **`delay = 1`**, and if **`cap4`** was pressed then **`delay = 4`**, etc.

```
            time.sleep(.05)

        if GPIO.input(cap1) == True:
            delay = 1
        if GPIO.input(cap2) == True:
            delay = 2
        if GPIO.input(cap3) == True:
            delay = 3
        if GPIO.input(cap4) == True:
            delay = 4
```

Since the **`while:`** loop exited you know that one of the pads was pressed. This code will quickly check each pad and assign the value of delay based on which pad was pressed.

Step #12

Now that you have the selected delay time stored as a variable, you can display a message letting the user know the target distance, and a message about when the range will be captured. There is, however, a small problem with this plan. The string "**`Capturing in X seconds`**" works for selections **`2`**, **`3`**, and **`4`**, but not for **`1`**. Since you don't want to display "1 seconds", the second line will have to be customized based on the value of **`delay`** to maintain correct grammar.

The first two lines of the message will be very similar to the **`Press pad to start`** block of code form Step #9. The first line will display the target distance and the second will display **`"Capturing range"`**. The third line of the display will need to be customized using an **`if/else`** block.

If delay is 1 then the third line should be **`"in %i second" %delay`** to maintain proper grammar for the single second. If **`delay`** is anything else, then the third line should be **`"in %i seconds" %delay`** to properly display multiple seconds.

At the end of this block you will update the display and insert a delay equal to the value of the **`delay`** variable by using **`time.sleep(delay)`**:

```
        if GPIO.input(cap4) == True:
            delay = 4

        draw.rectangle((0,0,width,height), outline=0, fill=0)
        draw.text((0, 0), "Target = %s" %target,  font=font, fill=255)
        draw.text((0, 22), "Capturing range",  font=font, fill=255)
        if delay == 1:
            draw.text((0, 44), "in %i second" %delay,  font=font, fill=255)
        else:
            draw.text((0, 44), "in %i seconds" %delay,  font=font, fill=255)
        update_display()
        time.sleep(delay)
```

Step #13

The delay is over, and you are now ready to capture the range to the user. You can do this by accessing the **`average()`** function inside your **`ultrasonic.py`** file, and setting the result equal to a variable named **`distance`**.

Next, you will create a variable named **`diff`** that will hold the absolute value of the random **`target`** variable minus the **`distance`** returned from the **`average()`** function. This will be expressed as **`diff = abs(target-distance)`**.

Now you will calculate the range of values that will get a green LED based on the difficulty that's been selected. Create a variable named **`window`** that equals **`10 * skill`**. This means:

Easy mode, **`skill = 2`** so **`window`** will equal 10 X 2 or **`20`**

Hard mode, **`skill = 1`** so **`window`** will equal 10 X 1 or **`10`**

Add these program tasks with the following lines of code:

```
        update_display()
        time.sleep(delay)

        distance = ultrasonic.average()
        diff = abs(target-distance)
        window = 10 * skill
```

Step #14

It's now time to light the LED using the **diff** and **window** variables. If **diff** is smaller than or equal to **window**, the user distance is good, so the LED should turn green. If not, the user distance that round was larger than the window, so the LED should turn red. This can be accomplished by using a simple **if/else** block:

```
        distance = ultrasonic.average()
        diff = abs(target-distance)
        window = 10 * skill

        if diff <= window:
            GPIO.output(green, GPIO.HIGH)
        else:
            GPIO.output(red, GPIO.HIGH)
```

Step #15

The program is almost done.

Finally, you will display the target and player distances, along with the difference between the two, add a 5 second delay so there is time to view the results, and turn off the LED so it's ready for the next round.

The display block is exactly like other blocks above it:

- Blank the image buffer with a black rectangle
- Line 1 will display "Target =" along with the value of **`target`**
- Line 2 will display "Player =" along with the value of **`distance`**
- Line 3 will display "Diff =" along with the value of **`diff`**
- Update the display

You will use **`time.sleep(5)`** for the delay and the **`rgb`** list to turn off all LED elements:

```
        else:
            GPIO.output(red, GPIO.HIGH)

        draw.rectangle((0,0,width,height), outline=0, fill=0)
        draw.text((0, 0),  "Target = %s" %target,  font=font, fill=255)
        draw.text((0, 22),  "Player = %.0f" %distance,  font=font, fill=255)
        draw.text((0,44),   "Diff = %.0f" %diff,  font=font, fill=255)
        update_display()
        time.sleep(5)
        GPIO.output(rgb, GPIO.LOW)
```

Step #16

Double check your program and indentation against the code below:

```
import time, RPi.GPIO as GPIO, random, ultrasonic, Adafruit_SSD1306
from PIL import Image, ImageDraw, ImageFont

slide = 16
rgb = [13,19,26]
red = 13
green = 19
blue = 26
cap = [22,27,17,4]
cap1 = 22
cap2 = 27
cap3 = 17
cap4 = 4

GPIO.setmode(GPIO.BCM)
GPIO.setup(slide, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(rgb, GPIO.OUT)
GPIO.setup(cap, GPIO.IN)

disp = Adafruit_SSD1306.SSD1306_128_64(rst=None)
disp.begin()
width = disp.width
height = disp.height
image = Image.new('1', (width, height)) # '1' converts image to 1-bit color
draw = ImageDraw.Draw(image)
font = ImageFont.truetype('/usr/share/fonts/truetype/freefont/FreeSans.ttf',18)

def update_display():
    disp.image(image)
    disp.display()

try:
    while True:
        for skill_selection in range(0,20):
            if GPIO.input(slide) == False:
                draw.rectangle((0,0,width,height), outline=0, fill = 0)
                draw.text((0, 0), "Difficulty?",  font=font, fill=255)
                draw.text((0, 22), "Easy",  font=font, fill=255)
                update_display()
                skill = 2
            else:
                draw.rectangle((0,0,width,height), outline=0, fill = 0)
                draw.text((0, 0), "Difficulty?",  font=font, fill=255)
                draw.text((0, 22), "Hard",  font=font, fill=255)
                update_display()
                skill = 1
            time.sleep(.1)

        target = random.randint(20,100)

        draw.rectangle((0,0,width,height), outline=0, fill=0)
```

```
        draw.text((0, 0), "Target = %s" %target,  font=font, fill=255)
        draw.text((0, 22), "Press pad",  font=font, fill=255)
        draw.text((0, 44), "to start",  font=font, fill=255)
        update_display()


        while GPIO.input(cap1)==GPIO.input(cap2)==GPIO.input(cap3)==GPIO.input(cap4)==False:
            time.sleep(.05)

        if GPIO.input(cap1) == True:
            delay = 1
        if GPIO.input(cap2) == True:
            delay = 2
        if GPIO.input(cap3) == True:
            delay = 3
        if GPIO.input(cap4) == True:
            delay = 4

        draw.rectangle((0,0,width,height), outline=0, fill=0)
        draw.text((0, 0), "Target = %s" %target,  font=font, fill=255)
        draw.text((0, 22), "Capturing range",  font=font, fill=255)
        if delay == 1:
            draw.text((0, 44), "in %i second" %delay,  font=font, fill=255)
        else:
            draw.text((0, 44), "in %i seconds" %delay,  font=font, fill=255)
        update_display()
        time.sleep(delay)

        distance = ultrasonic.average()
        diff = abs(target-distance)
        window = 10 * skill

        if diff <= window:
            GPIO.output(green, GPIO.HIGH)
        else:
            GPIO.output(red, GPIO.HIGH)

        draw.rectangle((0,0,width,height), outline=0, fill=0)
        draw.text((0, 0),  "Target = %s" %target,  font=font, fill=255)
        draw.text((0, 22),  "Player = %.0f" %distance,  font=font, fill=255)
        draw.text((0,44),   "Diff = %.0f" %diff,  font=font, fill=255)
        update_display()
        time.sleep(5)
        GPIO.output(rgb, GPIO.LOW)

except KeyboardInterrupt:
    disp.clear()
    disp.display()
    GPIO.cleanup()
```

Step #17

Now that your program is complete, run the program. Select the difficulty by using the slide switch and start the capture process by pressing one of the Capacitive Touch pads.

If your program is not working as expected, identify which area of the program needs to be checked. Each block of code is performing a very specific function, so identify what part of the game is not working properly and check out the block of code that is controlling that behavior.

If you still can't get your program to work, you can download a copy of this program from the Level B Resource Page.

LESSON B-18

QUESTIONS FOR UNDERSTANDING

1. Does signal level-shifting require an IC or can it be done with two resistors?

2. Can every file be used an import without causing any problems in your main program?

3. What function can be used to take the absolute value of a variable or expression?

Answers Can be Found on the Next Page

Answers

1. **Does signal level-shifting require an IC or can it be done with two resistors?**

   ANSWER: Signal level shifting can be done with two resistors but if multiple 5V devices are being used, using an IC for level shifting is recommended.

2. **Can every file be used an import without causing any problems in your main program?**

   ANSWER: No. Everything in the imported file will run on import unless it's inside an **`if __name__=="__main__":`** condition. Loops or other types of code in the imported file could cause problems when imported and must be enclosed in this **`if:`** condition to isolate it during your import.

3. **What function can be used to take the absolute value of a variable or expression?**

   ANSWER: The **`abs()`** function will return the absolute value of a variable or expression.

42 ELECTRONICS

INTRO TO ROBOTICS LEVEL B

Scope and Sequence

Lesson 1

Administrative and File Management

- Understanding the Raspbian File System
- Completing File Tasks in the Graphical User Interface (GUI)
    - Creating a Folder
    - Creating a File
    - Renaming a File or Folder
    - Changing Permissions on a File or Folder
    - Root User
    - Deleting Files or Folders
    - Navigating Folders Using File Manager
- Completing File Tasks Using the Command Line
    - Working Directory
    - Listing Directory Contents
    - Creating a File
    - Creating a Directory
    - Moving to Another Directory
    - Copying a File or Folder
    - Moving or Renaming a File or Folder
    - Deleting a File or Folder
- Other Useful Command Line Tools
    - Confirming Raspbian Version
    - Taking a Screen Capture
- Activities:
    - Activity #1: GUI File and Folder Operations
    - Activity #2: Command Line File and Folder Operations

Lesson 2

Functions

- Understanding Functions
  - Keeping Code Organized
  - Grouping Related Program Blocks
  - Global and Local Variables
  - Functions with Arguments
  - Using Functions in Programs
- Activities
  - Activity #1: Simple Function Program
  - Activity #2: Calling a Function with a Loop
  - Activity #3: Functions with Passed Arguments

Lesson 3

Program Layout Options and Advanced String Concepts

- Program Layout Options
  - While True Loops
  - Try, Except, Finally Program Layout
- Advanced String Concepts
  - Determining the Length of a String
  - Accessing the Value of a Specific String Position
  - Accessing Values from Multiple String Positions
  - Uppercase and Lowercase
  - Replacing Characters in a String
- Activities
  - Activity #1: While True Loop
  - Activity #2: Error Handling Program
  - Activity #3: Practice with Strings

Lesson 4

Pulse Width Modulation

- Import Methods
  - Standard Module Import
  - Importing a Module Using an Alias
  - Importing Multiple Modules
  - Importing Specific Functions from a Module
- Analog Signals vs Digital Signals
- Pulse Width Modulation (PWM)
  - PWM Duty Cycle and Frequency
  - Hardware PWM vs Software PWM
  - GPIO Commands to Control PWM
- Activities:
  - Activity #1: Building an LED and Piezo Circuit
  - Activity #2: Controlling an LED with PWM
  - Activity #3: Adjusting LED Brightness

Lesson 5

Switches and Correcting for Switch Bounce

- Pull-up and Pull-down Options
- Slide or Toggle Switch
- Switch Bounce
- Using a Delay to Save System Resources
- Activities:
  - Activity #1: Adding Switches to the Circuit
  - Activity #2: Counting Button Presses
  - Activity #3: Tuning a Loop Using a Delay
  - Activity #4: Adding LED Confirmation Using the Slide Switch

Lesson 6

Logical Operators

- Logical Operators in Python
- Using a Pushbutton Switch as a Toggle
- Using a Pushbutton Switch to End a Program
- Activities:
    - Activity #1: Using a Pushbutton Switch as a Toggle
    - Activity #2: Use And/Or to Control Different Events
    - Activity #3: Add Stop Button Functionality

Lesson 7

Working with a 3x4 Matrix Keypad

- Matrix Style Input Panel
    - Program Flow for Checking a Matrix
    - Storing Values as Strings Versus Integers
    - Membrane Type Switches
    - Matrix Keypad Warning
- Variables and Scope
- Activities:
    - Activity #1: Adding the Switch Matrix
    - Activity #2: Creating a Program to Display Keypad Presses
    - Activity #3: Adding Exit Functionality to the Keypad Program

Lesson 8

Github and Python 2 vs Python 3

- Library Files
  - What is Github?
  - Cloning a Library from Github
  - When to Clone a Remote Library
  - Completing the Library Installation
  - Viewing Code Inside Files on Github
- Python 2 vs Python 3
  - Thonny and Python
- Activities:
  - Activity #1: Cloning and Installing a Github Repository
  - Activity #2: Exploring the Adafruit MCP3008 Repository
  - Activity #3: Viewing Code on Github

Lesson 9

Analog Signal Processing with the Raspberry Pi

- Voltage Dividers
- Analog Input on the Raspberry Pi
- Integrated Circuits (IC)
  - IC Datasheets
  - IC Pin Numbering
  - IC Pinouts
  - ESD and IC Handling Precautions
- Digital Communication
  - The MCP3008 A/D Converter
- Activities:
  - Activity #1: Connecting the MCP3008
  - Activity #2: Reading a Value from the MCP3008
  - Activity #3: Adding a Voltage Divider
  - Activity #4: Updating the Raspberry Pi's Software

Lesson 10

Potentiometers, Phototransitors, and Advanced List Commands

- Variable Resistors
- Light Sensors
- Advanced List Commands
    - Creating a List Using the Split Command
    - Adding Items to a List
    - Finding Items in a List
    - Determining if an Item is Part of a List
    - Finding the Length of a List
    - Sorting a List
    - Printing all Items in a List Along with Their Index Values
    - Removing Items from a List
    - Clearing All Items from a List
- Activities
    - Activity #1: Adding the Potentiometer and Phototransistor
    - Activity #2: Incorporating the LED as an Indicator
    - Activity #3: Working with Lists

Lesson 11

RFID Systems

- RFID Technology
    - How RFID Works
    - RFID for Access Control
    - Security Concerns for RFID
    - MFRC522 Tag Reader
    - RFID Tags
        - Reading Text from the RFID Tag
- Removing Trailing Spaces in Python
- Determining Program Type: Python Code SHEBANG or #!
- Reading and Writing Tags
- Activities:
    - Activity #1: Adding the RFID Reader and Software
    - Activity #2: Reading and Writing Tags
    - Activity #3: Creating an Access Control Program

Lesson 12

RFID II

- Using a File for Input
  - Input File Types and Formatting
  - Opening a File in Your Program
  - Reading Values from a File
- Multithreaded Operation
  - Important Notes About Multithreaded Operation
- Formatting and Displaying Time and Date
- Breaking Out of a Loop
- Activities:
  - Activity #1: Read an External File in Python
  - Activity #2: Tag Reader Indicator
  - Activity #3: Adding Date and Time Messages

Lesson 13

Level Shifting and Infrared Sensors

- Signal Level Shifting
  - Hardware Level Shifting
- Infrared (IR) Obstacle Sensor
  - Alignment of Obstacle Sensors
- Infrared Line Sensor
  - Alignment of Line Sensors
- Activities
  - Activity #1: Level Shifter and the RGB LED
  - Activity #2: Adding the IR Obstacle Sensor
  - Activity #3: Adding the IR Line Sensor

Lesson 14

Ultrasonic Range Sensing and NUMPY

- Ultrasonic Rangefinders
  - Ultrasonic Signals
  - Finding Range with an Ultrasonic Sensor
  - Using Distance Information in Programs
  - Error Handling
- NumPy
- Integers and Floats
- Printing with Notation
- Activities:
  - Activity #1: Adding the Ultrasonic Range Sensor
  - Activity #2: Creating a Program to Read Ranges
  - Activity #3: Averaging the Range Values Using NumPy
  - Activity #4: Adding the RGB LED as a Distance Indicator

Lesson 15

I2C and Temperature Sensing

- I2C Communication
  - I2C Addressing
  - Enabling I2C on the Raspberry Pi
  - List of I2C Devices
  - Detecting Attached I2C Devices
- The BMP280 Temperature Sensor
  - BMP280 Wiring
  - BMP280 Commands
  - Temperature Conversion
- The WGET Command
- Running Modules as Imports vs Directly
- The SYSTEMEXIT() Command
- Activities:
  - Activity #1: Enabling the I2C Interface
  - Activity #2: Adding the BMP280 Temperature Sensor
  - Activity #3: Creating a Program to Display Temperature
  - Activity #4: Adding the RGB LED to Indicate Temperature

Lesson 16

OLED I2C Display

- OLED Display Hardware
  - Resolution and Pixels
  - OLED Technology
  - I2C Communication
- SSD1306 Display Driver
  - Required Modules
  - Size Configuration
  - Starting the Display
  - Variables that Simplify Communication with the Display
  - Drawing on the Display
- Unicode Characters
- Activities
  - Activity #1: Modify the Circuit
  - Activity #2: Create a Program to Display Text
  - Activity #3: Modify the Program to Display Sensor Information

Lesson 17

Capacitive Touch Sensor

- Capacitors
  - Construction
  - Current Limiting
  - Charge Time
    - Calculating Charge Time
  - Parallel vs. Series
- Capacitive Touch Sensor
- Male-to-Female Jumper Wires
- GPIO Pin Level Sensing
- Activities:
  - Activity #1: Powering a Circuit Using a Capacitor
  - Activity #2: Measuring Capacitor Discharge Time
  - Activity #3: Using the Capacitive Touch Sensor

Lesson 18

Range Sensing Game (final project)

- Level Shifting with Resistors
- Absolute Value in Python
- Modifying a File for Import Use
- Activities:
    - Activity #1: Modifying Ultrasonic.py
    - Activity #2: Adding the Ultrasonic Range Sensor
    - Activity #3: Building the Range Game