Movement and Other Output

Setting up a Finch program

Before you begin programming the Finch, there are a couple of things you'll need to do before you can start. First, your program will always need the following line at the top of the program:

```
from finchAPI import *
```

This statement brings in the resources that you need. This statement looks up the file "finchAPI.py" and imports all of the code from the file into the current program. Don't worry about understanding the specifics of this: all that you need to know is that it allows you to access many of the resources that you'll need.

When you run a program that has this line at the top, you may get a message that says "Finch is not plugged in". If you see this, check that the Finch is securely connected to your computer and restart the program. You may also get a red error message. It probably means that your program can't find the file "finchAPI.py". Make sure that your program is in the same directory as the file "finchAPI.py". If you have any further questions, refer back to the Setting Up unit.

If you are on a mac, you must make Python programs as a separate file inside the directory with finchAPI. That is, you cannot use the shell to program the Finch dynamically. This is because the macOS security system prevents you from directly modifying application files, which includes placing files anywhere in the application directory.

`Try it yourself`: place `from finchAPI import *` into a program and run it. If you get any error messages, ask your instructor for help. If nothing happens, you are free to move ahead..

Moving the Finch

The first thing you might want to do with any robot is to make it move. To move the Finch, use the functions **`forward()`**, **`backward()`**, **`turnLeft()`**, and **`turnRight()`**. **`forward()`** moves the Finch forward, **`backward()`** moves the Finch backward, **`turnLeft()`** turns the Finch counterclockwise, and **`turnRight()`** turns the Finch clockwise. Each of these will continuously move the Finch once the call is made. The code will continue to execute while the Finch is moving. To stop the movement, use **`stop()`**. **`stop()`** halts both wheels, stopping any movement that is currently happening.

```
forward()    #Moves the Finch forward
backward()   #Moves the Finch backward
turnLeft()   #Turns the Finch counterclockwise
turnRight()  #Turns the Finch clockwise
stop()       #Stops all movement
```

Each of the above functions can also take an argument to specify the amount of movement. **`forward()`** and **`backward()`** take a number of inches to move. **`turnLeft()`** and **`turnRight()`** take an angle in degrees to turn. Unlike before, supplying the inches or angle will make the function wait for the movement to complete. Code will not continue to execute until the movement is finished. This offers an advantage of chaining movement in a simple way. The following code will move the Finch forward 100 inches, then backward 100 inches.

```
forward(100)     #Moves forward 100 inches
backward(100)    #Moves backward 100 inches
```

The speed of the Finch's motors are initially set to ¼ of its full speed. This is to ensure the accuracy of its movement. You can change the speed of the Finch by using **`setSpeed()`**. **`setSpeed()`** takes a single number as an argument, from 0 to 1. At 0, the Finch will not move at all. At 1, the Finch will move at top speed. Note that the real physical issues of moving, like motor force, wheel slippage, friction, and air resistance, results in less actual gain in movement than is programmed. The difference between .75 and 1 is not actually a 25% increase in movement speed.

```
setSpeed(0)      #Not moving at all
setSpeed(.25)    #Moving at 1/4 speed
setSpeed(.5)     #Moving at 1/2 speed
setSpeed(.75)    #Moving at 3/4 speed
setSpeed(1)      #Moving at full speed
```

If you find yourself needing to control each wheel manually, you can use **`setWheels()`**. **`setWheels()`** takes two numbers as arguments. This is unlike the functions you've seen previously where you only input a single argument. Any function that takes multiple arguments must have those arguments separated by commas to tell the function which input is which.The arguments must also be put in a specific order within parentheses. As illustrated by the **`setWheels()`** example functions below,, the value for the left wheel is first argument (on the left) and the value for the right wheel is second argument (on the right). These values range from -1 to 1. At -1, the wheel is moving backwards at full speed. At 1, the wheel is moving forward at full speed. This also overrides the set speed because it

manually controls the wheels.

```
setWheels(-1, -1)    #Moving backwards at full speed
setWheels(-.5, -.5)  #Moving backwards at half speed
setWheels(1, 1)      #Moving forwards at full speed
setWheels(.5, .5)    #Moving forwards at half speed
setWheels(1, -1)     #Turning right in place
setWheels(-1, 1)     #Turning left in place
setWheels(1, .5)     #Turning right gradually
setWheels(.5, 1)     #Turning left gradually
```

Notice how the first number and the second number have a comma between them. This separates the arguments. The parentheses surround both numbers, which indicates that they are both arguments to **`setWheels()`**. Further, the values are distinct. The combination (1, -1) causes the Finch to turn right while the combination (-1, 1) causes the Finch to turn left. This is because each argument operates on a different wheel.

`Try it yourself`: move the Finch in a square. Then, move the Finch in a circle. (Hint: you should be able to use a single line of code to move in a circle). If you feel like you still need more practice, play around with the movement commands in the shell.

Lighting the Nose

Lights are useful indicators on any machine. Most modern computers have several lights to indicate if it has low battery, if the battery is currently charging, or if the computer is in sleep mode. The Finch has a single light in its nose that is capable of changing colors. This light may prove useful in providing a visual indicator of what the Finch is currently doing. To control the light, use **`light()`**. This can be used in one of two ways.

`light()` can be supplied a single string that tries to match the word you've put in with a color to display. Light can take any of the following as colors: red, blue, green, cyan, magenta, yellow, and white. You may also supply the string **`"off"`** to turn off the light.

```
light("red")     #turns the light red
light("blue")    #turns the light blue
light("green")   #turns the light green
light("cyan")    #turns the light cyan
light("purple")  #turns the light purple
light("yellow")  #turns the light yellow
light("white")   #turns the light white
light("off")     #turns the light off
```

`light()` can also take three numbers as the red, blue, and green color components of the light, separated by commas and in that order. If you are familiar with the RGB color model, this might be interesting, but the above colors should be sufficient for most applications.

```
light(255, 0, 0)     #turns the light red
light(0, 255, 0)     #turns the light blue
light(0, 0, 255)     #turns the light green
light(0, 255, 255)   #turns the light cyan
light(255, 0, 255)   #turns the light purple
light(255, 255, 0)   #turns the light yellow
light(255, 255, 255)#turns the light white
light(0, 0, 0)       #turns the light off
```

`Try it yourself`: create a program that turns the light blue and moves the Finch forward a foot. When the movement is complete, turn the light red.

Making Some Noise

Like lights, sounds are also useful indicators on machines. While visual cues are excellent, non-invasive, ways of alerting people to some event, sounds draw attention to said event while similarly alerting people to the event. Fire alarms operate on this principle (imagine if fire alarms just lit up red). In the context of programming, sounds can be used to signal that some action has taken place within a program. For this purpose, the Finch has a buzzer that can generate sounds with frequencies between 20 to 20000 Hz, which is the audible frequency range for humans (but, you should avoid using sounds above 5000, as those can hurt your ears). To make a sound with the buzzer, use **`buzz()`**.

`buzz()` takes two numbers as arguments. The first number is the duration in seconds of the sound you want to play and the second number is the frequency of the sound in Hertz. These two arguments are separated by a comma syntactically. The following code plays a sound at the frequency of 1000 Hz for 1 second:

```
buzz(1, 1000)
```

Unlike when moving the Finch, **`buzz()`** does not wait for the sound to finish. Because of this, the Finch may continue moving and otherwise acting as it would while the sound is playing. If you want to wait for the sound to finish before continuing, use **`delayedBuzz()`** instead. Otherwise, this works exactly as **`buzz()`**.

```
delayedBuzz(1, 1000)
#Will wait for 1 second
print("Sound is over")
```

The Finch can also play music using it's buzzer. To do so, compose a string of notes separated by spaces. For example, **`"A B C A B C"`**. You can make notes sharp by adding a "#" and make notes flat by adding a "b", like "A#" and "Eb". Then, use **`sing()`**, with the notes and speed input as comma-separated arguments. The speed is how long a note is held for. The Finch will sing the song you've put in at the desired speed.

```
sing("A B C A B C", .25)
sing("A C D C", .5)
sing("C F# Eb", .1)
```

Try it yourself: create a program that plays a 440 frequency sound for a second before moving forward a foot. Once the Finch completes the movement, play a 880 frequency sound for a second.

Simultaneous Behaviour

Many robots will do things simultaneously. They'll be moving, checking sensors, playing sounds, lighting up, and much more, all at the same time. The Finch is also capable of doing this. Already, you've seen that the movement functions keep running code if no distance or angle is supplied. **buzz()** and **light()** also keep running the code while they're in effect. You can combine these to make complicated behaviour, like moving while playing a song.

You will also want to move a set distance or angle while doing other things. For example, you may want to move a certain distance while checking the sensors. To accomplish this, you can add **False** as an argument to any movement function, separated by a comma. This tells the program not to wait for the movement to finish. The movement functions are set to **True** by default, which is why the program will wait by default. Note that if you call another movement after a movement function with **False**, it will override the previous one:

```
forward(100, False)  #the program will not wait for the movement to finish
backward(100)        #this movement executes as it overides the previous movement
```

You can check if the Finch is currently moving by using the flag **isMoving**. **isMoving** will be **False** whenever the Finch is not executing a movement function and will be **True** whenever the Finch is executing a movement command.

```
print(isMoving)      #Will be False
forward(100, True)   #Will wait for the movement to finish
print(isMoving)      #Will be False BECAUSE THE LAST MOVEMENT WAITED
forward(100, False) #Will not wait for movement to finish
print(isMoving)      #Will be True because the Finch is still moving forward
```

If the Finch is engaged in simultaneous behaviour and needs to stop everything, the function **halt()** can be used. **halt()** will stop everything the Finch is currently doing, including, but not limited to, stopping all movement, turning off lights, silencing any noise, etc. This can be useful for an emergency measure.

Knowing when to use the **halt()** function will become much more apparent when we start receiving data from sensors continuously. You'll then be able to make the Finch

respond to its surroundings while doing other things. For now, know that this is possible and it will be revisited later.

`Try it yourself`: make the Finch move three feet while playing a song.

Key Points

- To move the Finch, use **`forward()`**, **`backward()`**, **`turnLeft()`**, and **`turnRight()`**. Each can take a number for a distance in inches or an angle to turn. When using a distance or angle, they an also take a boolean value which determines if the program should wait for the movement to finish. This is **`True`** by default, but adding **`False`** makes the program continue during movement. You can stop any movement currently happening using **`stop()`**.
- To light up the Finch's nose, use **`light()`**.
- To use the buzzer, use **`buzz()`**. If you want to wait for the sound to finish, use **`delayedBuzz()`**. If you want to play a song, use **`sing()`**.
- To stop everything that the Finch is doing, use **`halt()`**.

Exercises

1. Move the Finch in a triangle. Before a turn, slow down the Finch's movement, make the Finch play a short low noise and make the nose red. Before moving forward, speed up the Finch's movement, make the Finch play a short high noise and make the nose green. It is best to do this in small parts, tackling the movement first, then putting in the sounds and lights.