Continuous Behaviour

Continuous Behaviour

Using loops, you can make the Finch do things continuously. Now, you can make it drive in a square as many times as you want by simply putting it in a loop. But, this is most powerful when you make the Finch run until it reaches some exit condition. Here is an example.

```
while not isTapped():
    forward(10)
    turnLeft(90)
```

This simple program will cause the Finch to travel in a square until you tap it. In this program, the exit condition is being tapped. Until it is tapped, it will just keep driving forward 10 inches and turning left 90 degrees.

This method of coding allows you to complex behaviours that occur constantly until the user decides to quit. Many modern programs operate this way: operating systems require the user to shut down, browsers say open until they are closed, and your router keeps the internet on until you shut it off.

`Try it yourself`: move the Finch in a triangle. At each turn, print out the temperature and light level. If the Finch is ever on its tail, exit the program.

Combining Conditionals and Iteration

Now that you can make the Finch behave continuously and conditionally, you can create very complex behaviours by combining them. Putting conditionals in loops allows you to create a branching behavioural scheme that operates until some condition is met. This strategy will be incredibly important moving forward. Many programs will use this strategy to leverage the full power of the Finch. This example demonstrates how you can use this to continue checking the light until you've moved into a spot where the light is greatest.

```
checks = 0  #How many times we've checked
lightLevel = detectLight()  #Our initial light level
while checks < 4:   #Until we've checked 4 times
    forward(12)
    if detectLight() > lightLevel:
        #If light is greater here, reset
        lightLevel = detectLight()
        checks = 0
    else:
        #If light is not greater here, go back
        backward(12)
        #Preparing for next check
        turnLeft(90)
        checks = checks + 1
```

The code uses two variables: **`checks`** and **`lightLevel`**. **`checks`** is the number of times you've moved away from our best point. The best point is our initial point at first, so we initialize **`lightLevel`** to our current light. We then move from the center, checking the light. If the light is greater, we make that our new center, **`lightLevel`** to the light there and resetting setting **`checks`** to 0. If it is no, we move back to our starting point, turn 90 degrees for our next check, and mark that we've checked one more time. We stop the loop when we've done so 4 times with no improvement.

The result is that the Finch will look for the place with the greatest amount of light in the room. The Finch will do so until it's convinced, to an accuracy of about of foot, that it has found the brightest spot in the room. The conditional creates the checking behaviour, which is how the Finch comes to know the brightest spot. The loop allows the Finch to perform this check continuously, so that it may check more than just a single spot. By combining loops and conditionals then, we've created this complex behaviour.

`Try it yourself`: create a program that finds the place with the greatest temperature in the room.

Simultaneous Behaviour: Revisited

Recall that you can move the Finch, use the buzzer, and light the nose all at the same time. The movement functions, **`forward()`**, **`backward()`**, **`turnLeft()`**, and **`turnRight()`**, do not stop the code from executing when they are not supplied with a distance or angle. If they are, you can continue the code by adding an extra boolean value input to the function like so:

```
forward(10, False)
backward(15, False)
turnLeft(90, False)
turnRight(180, False)
```

Note that calling movement functions that do not wait, one after another, will override each other, defaulting to the most recent function call. In this case, the Finch will only turn right 180 degrees. To prevent overriding, you can check **`isMoving`**, a flag that is **`True`** when the Finch is moving and **`False`** when the Finch is not moving. Particularly, you'll want to use a loop with a conditional like so:

```
emergencyBrake = False
while not isMoving and not emergencyBrake:
    if detectObstacle():
        emergencyBrake = True
```

This technique will also be very valuable. While the Finch is moving and doing other things, you can check the sensors and make the Finch respond to its environment. Just set the movement and execute a loop that checks the sensors until **`isMoving`** is **`False`**.

`Try it yourself`: turn the nose green. Then, move in a square. Stop if the Finch detects an obstacle and turn the nose red.

Key Points

- Use loops to build a continuous behavior.
- Combining loops and conditionals allows you to build complex behaviours.
- By using movement functions that continue executing code, you can make the Finch operate while making it respond to its environment.

Exercises

1. Make the finch move along a path with n sides each of length d where the user can enter both n and d.

   Hint: for a square with 4 sides, each turn needs to be 90° = 360° / 4, and a triangle with three sides, the finch will need to turn 120° = 360° / 3.

2. Make the finch patrol back and forth in a straight line between two points, stopping, changing the color of the nose light and emit a noise if it detects something blocking it.
3. Have the finch follow a path, but only in the dark. The finch should wait for darkness before it tries to move.