Using Functions: Navigation

Using Functions

By using functions, you can eliminate a lot of repetitive code. By eliminating repetitive code, you can create complex behaviours with only a few lines of code and reuse that code wherever applicable. One of your earlier exercises probably looked something like this:

```
forward(12, False)
emergencyBrake = False
while not isMoving and not emergencyBrake:
    if detectObstacle():
        emergencyBrake = True
turnLeft(90)
forward(12, False)
emergencyBrake = False
while not isMoving and not emergencyBrake:
    if detectObstacle():
        emergencyBrake = True
turnLeft(90)
forward(12, False)
emergencyBrake = False
while not isMoving and not emergencyBrake:
    if detectObstacle():
        emergencyBrake = True
turnLeft(90)
forward(12, False)
emergencyBrake = False
while not isMoving and not emergencyBrake:
    if detectObstacle():
        emergencyBrake = True
```

This can be painful to look at as a whole. It can also be annoying to repeatedly type that while loop when it does the exact same thing every time. Instead, you can make a function that does it for you, giving the code a nice, crisp look.

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

Now, we can just check if the emergency brake was triggered.

```
forward(12, False)
if checkObstacleWhileMoving():
    stop()
turnLeft(90)
forward(12, False)
if checkObstacleWhileMoving():
    stop()
turnLeft(90)
forward(12, False)
if checkObstacleWhileMoving():
    stop()
turnLeft(90)
forward(12, False)
if checkObstacleWhileMoving():
    stop()
```

Even after these changes, we have a lot of repeated code. If you look closely, you'll notice that the above code the same four chunks repeated. You can make yet another function that will simplify it even further.

```
def moveForwardAndTurn():
    forward(12, False)
    if checkObstacleWhileMoving():
        stop()
    turnLeft(90)
```

Our code for that behavior is now four lines of code.

```
moveForwardAndTurn()
moveForwardAndTurn()
moveForwardAndTurn()
moveForwardAndTurn()
```

To even further simplify, we can put this in a for loop that automates the repetition for us!

```
for i in range(0, 4):
    moveForwardAndTurn()
```

This is much better. The code performs the intended behavior, is easy to look at and understand, and is far easier to type out. Functions exist purely for the programmer's ease. They are tools that simplify otherwise complicated and messy code. Use functions whenever you want to make your code easier on the eyes and on your fingers.

`Try it yourself`: take any exercise or try it yourself you've completed thus far that uses repeated code or was hard to read at the end. Translate that code to use functions instead.

Responding to the Environment with Functions

The above code was a way to check for obstacles. his was best implemented using functions because of the complexity involved in avoiding obstacles. The function implemented above, **`moveForwardAndTurn()`**, handles obstacles by truncating the forward movement and turning in that spot instead of at the end. However, you may want to handle this differently and not have to retype the code for handling obstacles each and every time. The same is true of anything else in your environment, like temperature, light, and position. In fact, you may want to have a whole function dedicated to just responding to the environment.

```
def checkEnvironment():
    if detectLight() > .5:
        #do something
    else:
        #do somehing
    if detectTemperature() > 70:
        #do something
    else:
        #do something
    #etc
```

This allows you to compact your code that responds to the environment in a concise manner so you only have to write a single line when you want to do so.

Function Application: Navigation

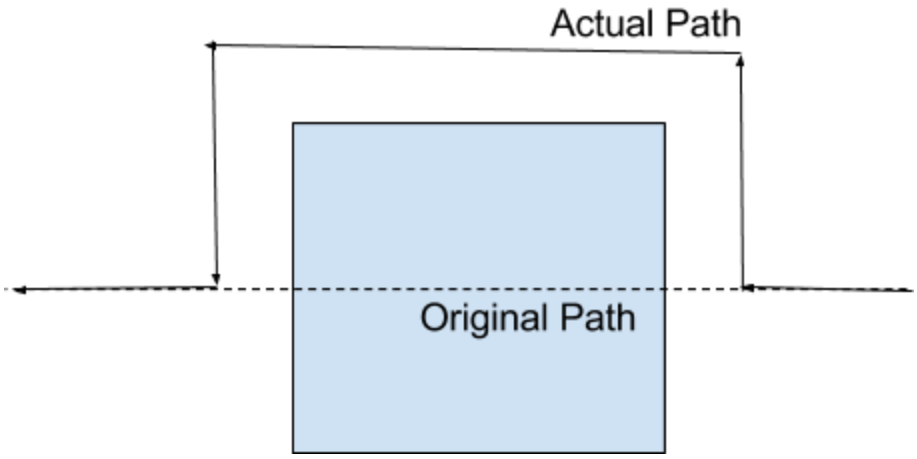
A navigation system is an example of a real-world application that uses functions. The Finch has several useful functions that control basic movement, but that alone is hardly enough to get the Finch to move from point A to point B while avoiding obstacles in between. Suppose I want to move forward in a line. This seems like it would only involve one step:.

```
forward()
```

However , that only works if you don't have an obstacle in the way. If you do, you will have to find a way to move around it. To deal with this in an orderly fashion, here is a function that handles an obstacle in an orderly fashion.

```
def goAroundObstacle():
    obstacle = True
    distanceLeft = 0        #Tracking distance
    #Going left
    while obstacle:
        if not detectObstacle():
            obstacle = False
        else:
            turnLeft(90)
            forward(6)
            distanceLeft = distanceLeft + 6
            turnRight(90)
    #At this point, obstacle is no longer in front of us
    obstacle = True
    #Going forward
    while obstacle:
        forward(6)
        distanceForward = distanceForward + 6
        turnLeft(90)
        if not detectObstacle():
            obstacle = False
        else:
            turnRight(90)
    #Returning to line of travel
    forward(distanceLeft)
    turnRight(90)
```

Calling this function will cause the Finch to go around a square obstacle. The function first assumes that there is an obstacle present and enters the first while loop. So long as the obstacle is still in front of the Finch, the Finch will move left 6 inches and check for an obstacle. Once it has done that, the Finch will move along the length of the obstacle, checking if the obstacle is still present to its left. If it's not, the Finch will return to the original path and return how far it has moved forward to get around the obstacle.



Here is an example use of this:

```
while not isTapped():
    forward()
    if detectObstacle():
        stop()
        goAroundObstacle()
```

This code will continue moving the Finch in a line until it is tapped. If the Finch encounters an obstacle, it will execute our **`goAroundObstacle()`** function to navigate around the obstacle. Note that **`goAroundObstacle()`** assumes that the obstacle is a square. For irregular shaped obstacles or an unknown environment, more complicated code would be needed. For now, recognize that this function allows one to navigate around a square obstacle easily.

`Try it yourself`: move the Finch in a circle. If it encounters an obstacle, turn in the opposite direction until there is no longer an obstacle. Then, go in a circle in the opposite direction.

Key Points

- Functions allow you to make code that is cleaner and easier to use.
- Functions allow you to have systematic ways of responding to your environment.
- Navigation is a complex system that is best implemented using functions.

Exercises

1. Write a function that has the Finch write out a letter of the alphabet. Then, write a function to have the Finch write out the initials of each team member.

2. Move the Finch forward until it encounters a wall. The, move the Finch along the wall until you tap it.