Forming an Algorithm

by Sophia

WHAT'S COVERED

In this lesson, you will review various problems and find patterns to help build a solution to those problems prior to writing code. Specifically, this lesson covers:

1. Introduction to Algorithms

With any problem that we're building a program for, we must define what the algorithm needs to be. Recall that the **algorithm** is the logical step-by-step plan that we need to build to solve the problem. When you design an algorithm, it's also important to think about different ways you can solve the problem and to find the best way to do it. As you learn about new functionality in a programming language, you'll find potentially better ways to optimize a solution.

THINK ABOUT IT

Before we get started, we should think about the big picture of the algorithm. In the end, what is the final goal of the problem? Consider other aspects like how often this program is meant to run, when the program creation deadline is, or how complex the problem is. For example, if a program is only meant to run once, it probably would be acceptable if it weren't fully optimized. However, if a program is meant to run millions of times a day, the optimization of a solution would be crucial. Think about a mobile app that you use on a regular basis. Would you continue to use that app if it took five minutes for the app to even load? Most would probably not.}}

Once we have the big picture in mind, we then want to think about the individual stages and steps that will need to be broken down. There are some basic criteria that you should think about, as we discussed in the programming mindset lesson.

STEP BY STEP

1. To start, we need to ask what the inputs into the problem are and where they are coming from. This should give us a starting point to the problem.
2. Next, we should look at what the outputs of the problem are. This is different than just the items that we output throughout the problem; rather, we should ask what the end result of the program should be. That could be a single output line, but it could also be a file, an email, or a report being generated.
3. Next, we need to think about what the order of the steps should be. The order is important as it helps define the actions that need to occur.

4. Within the program, we should look at what types of decisions need to be made. You'll remember in our prior lesson that we had Sophie needing to bring different items with her depending on the day or weather forecast. These types of decisions add a layer of choice and options based on the input and other variables.

NOTE: Also consider if there are any areas of the problem that could be repeated.
You will learn about solutions for addressing repeatability in the coming lessons. For now, look for patterns in your algorithm. Finding patterns can help reduce the number of steps it takes to get to your expected solution. These patterns can be converted to code later to help with processes that need to be repeated over and over again. So, rather than having to write for every single situation (step), you can streamline the code with the use of loops, functions, classes, and methods. You'll want to keep this in the back of your mind as you work through your algorithm to avoid having to rewrite step(s) many times. Don't worry too much yet about what loops, functions, classes, and methods are. We will be identifying all those possible coding solutions later. Just know for now that you can optimize your code by finding repeating patterns.

EXAMPLE
On the topic of repeatability and looking for patterns, here is a real-world example. Think about going to an ATM machine to withdraw money. You're initially presented with a menu option. Once you've completed a transaction like withdrawing money, you're presented with the menu again. You can complete as many transactions as you want before the program ends. Rather than having to completely exit out of the program after a single transaction, the menu is repeated over and over again until a selection is entered to exit out of the program.}}

TERM TO KNOW

Algorithm
A logical step-by-step plan that we need to build for solving a problem.

2. Guessing Game

EXAMPLE
In this example, we'll create a simple guessing game that allows the user to guess a random value between 1 and 100. The user will be able to keep guessing the number until the number is found. After each guess, the program will tell the user if the guess is too high or too low. Once the user has guessed the number, the program will tell the user how many times it took the user to guess and exit the program.

To start, we'll have to define the input elements of the program. In this case, we need to have a random number that is generated between 1 and 100. Let's call it `randomNumber`. We'll also need to store the input from the user for the guess. Let's call it `guess`. Lastly, we'll need a counter, to count the number of guesses that the user made before they guessed the number correctly. We'll call that variable `numberOfGuesses`.

Next, we'll determine what the output at the end of the program should be. This should be the `numberOfGuesses` output to the screen once the user has guessed the number correctly.

THINK ABOUT IT

Do we have any elements of conditions? We do this in this program. One is to check if the `guess` is higher than the `randomNumber`, after which we should increase the `numberOfGuesses` and inform the user. If the `guess` is lower than the `randomNumber`, we should also increase the `numberOfGuesses` and inform the user. If the `guess` is equal to the `randomNumber`, we'll end the program after outputting the result to the user.

Do we have any repeating elements? In this case, we do. We want to keep running the check and output to the user until the `guess` is equal to the `randomNumber`.

Let’s take a look at what this would look like in code. Don’t try to decipher the code below. You will learn later what functions do and how to write lines of code. Just see if you can follow the logic and read the comments (identified by the hashtags).

HINT

Comments: The “#” (hashtag) is used in Python code to identify a comment. Commented lines of code are ignored by the compiler during run time. A programmer adds comments throughout the code to explain how the program or function works. This makes it easier for the programmer or anyone else looking at the code to identify the intention of that section of code. Keep in mind that some programs are hundreds of lines of code or more, so commenting is a necessary practice. We will talk more about commenting and its use in a later lesson.

For this code example, we will only bold the actual compiled code that is running.

```
#We are importing a module that we need to be able to generate random numbers
import random

#We are creating a random number between 1 and 100
#and assigning it to randomNumber
randomNumber = random.randrange(1, 100)

#We are creating the variable numberOfGuesses and
#assigning it to 1 for the first guess.
numberOfGuesses = 1

#We are asking the user to enter in a value
#between 1 and 100 and store it in guess
guess = int(input("Guess a number between 1 to 100: "))

#Here is the start to our loop that will keep
#running as long as the guess is
#not the same as the randomNumber
while guess != randomNumber:
  numberOfGuesses += 1 #We increment the numberOfGuesses by 1 each time the loop runs
  if guess > randomNumber: #If the guess is larger than the random number
    print(guess, "is too high.")
    guess = int(input("Guess a lower number: "))
  elif guess < randomNumber: #If the guess is lesser than the random number
    print(guess, "is too low.")
    guess = int(input("Guess a higher number: "))
  else: #If the guess is the same (since we've checked both ranges already)
    print("You got it! ")

#Once the guess is equal to the randomNumber, we can output this line
print("You guessed correctly", numberOfGuesses, "guesses!")
```

Below is a screenshot of the code from a programming editor application.

```
#We are importing a module that we need to be able to generate random numbers
import random

#We are creating a random number between 1 and 100
#and assigning it to randomNumber
randomNumber = random.randrange(1, 100)

#We are creating the variable numberOfGuesses and
#assigning it to 1 for the first guess.
numberOfGuesses = 1

#We are asking the user to enter in a value
#between 1 and 100 and store it in guess
guess = int(input("Guess a number between 1 to 100: "))

#Here is the start to our loop that will keep
#running as long as the guess is
#not the same as the randomNumber
while guess != randomNumber:
    numberOfGuesses += 1 #We increment the numberOfGuesses by 1 each time the loop runs
    if guess > randomNumber: #If the guess is larger than the random number
        print(guess, "is too high.")
        guess = int(input("Guess a lower number: "))
    elif guess < randomNumber: #If the guess is lesser than the random number
        print(guess, "is too low.")
        guess = int(input("Guess a higher number: "))
    else: #If the guess is the same (since we've checked both ranges already)
        print("You got it! ")

#Once the guess is equal to the randomNumber, we can output this line
print("You guessed correctly", numberOfGuesses, "guesses!")
```

TERMS TO KNOW

Comments
The “#” (hashtag) is used in Python code to identify a comment. Commented lines of code are ignored by the compiler during run time. A programmer adds comments throughout the code to explain how the program or function works.

Module
A self-contained piece of code that can be used in different programs.

3. Algorithm to Guess the Guessing Game

Did you know there’s also an algorithm that can be used to guess the number in this game in seven guesses or less each time? You could get lucky and guess it sooner, of course, but this algorithm will be consistent. On the first guess, you'll want to choose the middle number, which is 50. This will eliminate 50% of all of the possible numbers, as the actual number will either be higher or lower than that. If 50 is the correct value, great! But you had a 1 in 100 chance on that number. Depending on if the number is higher or lower than 50, you'll select the next middle value. So, if the program says that 50 is too low, you'll guess 75, as that's the value between 50 and 100. By doing so, you're again eliminating 50% of all of the numbers that are left. Keep doing that each time and with each guess, you'll eliminate 50% of all of the remaining numbers until you have the final number.

Here's a demonstration of this algorithm:

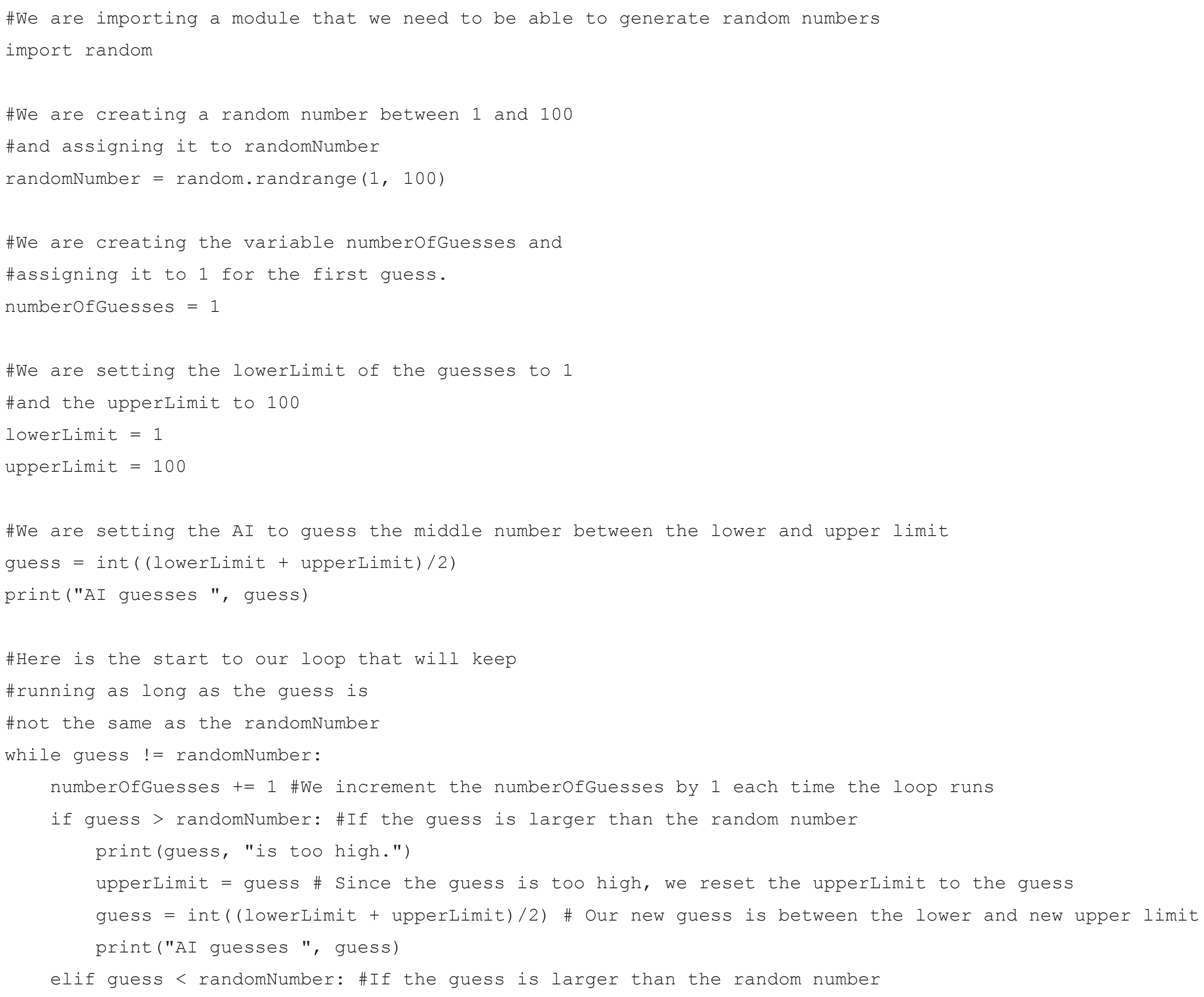
```
Guess a number between 1 and 100: 50
50 is too high.
Guess a lower number: 25
25 is too high.
Guess a lower number: 12
12 is too high.
Guess a number lower: 6
6 is too high.
Guess a number lower: 3
3 is too low.
Guess a higher number: 5
5 is too high.
Guess a lower number: 4
You guessed correctly 7 guesses!
```

With this in mind, we could build in an algorithm—which we'll call AI—to do this for us using that same algorithm. Again, don't try to decipher the code; just see if you can follow the logic. The changes from the previous code are in red text.

```
#We are importing a module that we need to be able to generate random numbers
import random

#We are creating a random number between 1 and 100
#and assigning it to randomNumber
randomNumber = random.randrange(1, 100)

#We are creating the variable numberOfGuesses and
#assigning it to 1 for the first guess.
numberOfGuesses = 1

#We are setting the lowerLimit of the guesses to 1
#and the upperLimit to 100
lowerLimit = 1
upperLimit = 100

#We are setting the AI to guess the middle number between the lower and upper limit
guess = int((lowerLimit + upperLimit)/2)
print("AI guesses ", guess)

#Here is the start to our loop that will keep
#running as long as the guess is
#not the same as the randomNumber
while guess != randomNumber:
  numberOfGuesses += 1 #We increment the numberOfGuesses by 1 each time the loop runs
```

```
    if guess > randomNumber: #If the guess is larger than the random number
        print(guess, "is too high.")
        upperLimit = guess # Since the guess is too high, we reset the upperLimit to the guess
        guess = int((lowerLimit + upperLimit)/2) # Our new guess is between the lower and new upper limit
        print("AI guesses ", guess)
    elif guess < randomNumber: #If the guess is larger than the random number
        print(guess, "is too low.")
        lowerLimit = guess #Since the guess is too low, we reset the lowerLimit to the guess
        guess = int((lowerLimit + upperLimit)/2) #Our new guess is between the new lower and upper limit
        print("AI guesses ", guess)
    else: #If the guess is the same (since we've checked both ranges already)
        print("AI got it! ")
#Once the guess is equal to the randomNumber, we can output this line
print("AI guessed correctly ", numberOfGuesses, "guesses!")
```

Again, here's a screenshot of the code now with the AI algorithm included from a programming editor application.



```
#We are importing a module that we need to be able to generate random numbers
import random

#We are creating a random number between 1 and 100
#and assigning it to randomNumber
randomNumber = random.randrange(1, 100)

#We are creating the variable numberOfGuesses and
#assigning it to 1 for the first guess.
numberOfGuesses = 1

#We are setting the lowerLimit of the guesses to 1
#and the upperLimit to 100
lowerLimit = 1
upperLimit = 100

#We are setting the AI to guess the middle number between the lower and upper limit
guess = int((lowerLimit + upperLimit)/2)
print("AI guesses ", guess)

#Here is the start to our loop that will keep
#running as long as the guess is
#not the same as the randomNumber
while guess != randomNumber:
    numberOfGuesses += 1 #We increment the numberOfGuesses by 1 each time the loop runs
    if guess > randomNumber: #If the guess is larger than the random number
        print(guess, "is too high.")
        upperLimit = guess # Since the guess is too high, we reset the upperLimit to the guess
        guess = int((lowerLimit + upperLimit)/2) # Our new guess is between the lower and new upper limit
        print("AI guesses ", guess)
    elif guess < randomNumber: #If the guess is larger than the random number
```

```
        print(guess, "is too low.")
        lowerLimit = guess #Since the guess is too low, we reset the lowerLimit to the guess
        guess = int((lowerLimit + upperLimit)/2) #Our new guess is between the new lower and upper limit
        print("AI guesses ", guess)
    else: #If the guess is the same (since we've checked both ranges already)
        print("AI got it! ")
#Once the guess is equal to the randomNumber, we can output this line
print("AI guessed correctly ", numberOfGuesses, "guesses!")
```

Let's take a peek at the results when running the program:

```
AI guesses 50
50 is too low.
AI guesses 75
75 is too low.
AI guesses 87
87 is too low.
AI guesses 93
93 is too high.
AI guesses 90
AI guessed correctly 5 guesses!

AI guesses 50
50 is too high.
AI guesses 25
25 is too high.
AI guesses 13
12 is too high.
AI guesses 7
7 is too high.
AI guesses 4
4 is too high.
AI guesses 2
2 is too low.
AI guesses 3
AI guessed correctly 7 guesses!

AI guesses 50
AI guessed correctly 1 guesses!

AI guesses 50
50 is too high.
AI guesses 25
25 is too low.
AI guesses 37
37 is too high.
AI guesses 31
31 is too high.
AI guesses 28
```

```
28 is too low.
AI guesses 29
29 is too low.
AI guesses 30
AI guessed correctly 7 guesses!
```

There were some lucky guesses along the way, but the algorithm did function as expected.

SUMMARY

In this lesson, we were **introduced to algorithms** as a logical step-by-step plan that is built to create a solution to a problem. We also had a chance to see some conditional and repeatable elements in the sample **Guessing Game** code. Finally, we saw that this basic guessing game can be enhanced by using an **algorithm (AI) to guess the guessing game** that can produce consistent tries. It's important to note that it's not the underlying code that matters at this point, but rather the logical steps behind the algorithm.

Best of luck in your learning!

TERMS TO KNOW

Algorithm
A logical step-by-step plan that we need to build for solving a problem.

Comments
The "#" (hashtag) is used in Python code to identify a comment. Commented lines of code are ignored by the compiler during run time. A programmer adds comments throughout the code to explain how the program or function works.

Module
A self-contained piece of code that can be used in different programs.