More Nested Loops

Prime Number Example

Consider the problem of listing the set of prime numbers within some given range. A prime number is a positive integer greater than one that is only divisible by one and itself. By definition, we can test to see if a number n is prime by dividing it by 2, 3, 4, …, n-1 and seeing if each of those results leaves a remainder. Thus, our algorithm (set of finite steps that solves a problem) to determine whether or not a positive integer n is prime or not is as follows:

Divide n by 2, 3, 4, …, n-1

If any of these divisions leaves a remainder of zero, answer that the number is not prime.

If none of these divisions leaves a remainder of zero, answer that the number is prime.

The basic structure of this algorithm indicates that we'll use a single loop to go through all the numbers we are dividing by. We also need to keep track of whether we've found a divisor of a number or not, so that when we exit our loop, we know what to answer. Furthermore, we may as well exit the loop as soon as a divisor is found instead of waiting to divide by the other numbers.

Once we've laid down this logic, we might even sketch out the following code segment:

```
# Try all possible divisors
isPrime = True
for div in range(2, n):
    if n%div == 0:
        isPrime = False
        break

# If none are found, print.
if isPrime:
    print(num,"is prime.")
else:
    print(num,"is not prime.")
```

Notice how we use isPrime as a flag. Though we don't use this flag in the loop condition, it nonetheless is linked to whether or not the loop will run. This technique of using a flag is common. The flag stores whether or not some event occurred. If the event ever does occur, we change the value of the flag to indicate this. In this case, the flag indicates whether or not a proper divisor of n has been found. (We use 1 to represent that a divisor has not been found and 0 to represent that one has.)

Also, notice how we check for divisibility. We don't use division, but we use mod and look at the remainder.

Finally, notice the use of the break statement which allows us to exit this loop as soon as a divisor is found.

Since the value of isPrime corresponds to whether or not n is prime, we can simply use that as our whole Boolean expression, since its type is Boolean.

Unfortunately, this does NOT solve our original problem of listing all the prime numbers in some given range. But, we can use this framework for a basic solution that looks like this:

For each number in our range to test

> Check if this number is prime.
> If so, output it. (If not, don't.)

Because checking whether or not a value is prime has a loop in it, technically this algorithm has a nested loop structure. But, because we've broken the problem down into separate pieces, we never had to think about the problem in that fashion.

Now, all we have to do is loop through all our possible values of n that we are testing for primality! Here is the whole program:

```
def main():

    start = int(input("What is the starting point of your range?\n"))
    end = int(input("What is the ending point of your range?\n"))

    print("All the primes in between",start,"and",end,"are:")

    # Try each number in the range.
    for num in range(start, end+1):

        # Assume prime initially.
        isPrime = True

        # Try all possible divisors
        for div in range(2, num):
            if num%div == 0:
                isPrime = False
                break

        # If none are found, print.
        if isPrime:
            print(num,end=" ")

    print()

main()
```

Notice that the bulk of the logic is exactly the same. In fact, the inner for loop is exactly the same! The only difference is that the if statement at the end doesn't have an else associated with it because we have no action to execute if the number isn't prime. Furthermore, the outer loop is very basic; just looping through each different value of n we would like to test. The if statement before the first loop simply avoids us from checking any values less than 2 and allows us to get rid of the if statement in the old version where we set is_prime to 0 for any value of n less than 2. Finally, because there are slight errors in precision with double values and we want to make sure that if the sqrt(n) is an integer, that it gets tested, since in all of these cases the number is composite. To ensure that the Boolean expression is true for this value of try_div, we add a bit of padding to the sqrt(n) to ensure than a slight error in its calculation on the low side doesn't kick us out of the loop. In general, it's very important to attempt to take into account the slight errors in calculations that occur with floats and doubles in this manner.

In general, most problems that have nested loops can be broken down into smaller pieces like this problem so that you don't have to think about the loops in a nested fashion. Attach one part of the problem as we did in this example and then treat that solution as a "unit". Then, consider the larger problem at hand and see if the unit of code you created can help you solve it. When you put these pieces together, you may very well have a nested loop structure, even if you didn't think about it that way to begin with.

Idea of a Simulation

Now that we can repeat statements many times and generate random numbers, we have the power to run basic simulations. One practical use of computers is that they can be used to simulate items from real life. A nice property of a simulation is that it can be done really quickly and often times is very cheap to do. Actually rolling dice a million times would take a great deal of time, and actually running a military training exercise may be costly in terms of equipment, for example. As you learn more programming tools, you'll be able to simulate more complicated behavior to find out answers to more questions. In this section we'll run a simple simulation from the game Monopoly.

There are 40 squares on a Monopoly board and whenever a player rolls doubles, they get to go again. If they roll doubles three times in a row, they go to jail. There are a few more ways that affect movement, but these are more complicated to simulate. Let's say we wanted to know how many turns it would take on average to get around the board. We have the tools to simulate this. One simplification we'll make is that we'll stop a turn at three doubles in a row and we won't put the player in jail. We are making this simplification so that our code will be manageable. We will still get a reasonable result without having to add complicated code. Whenever writing simulations, we must make decisions about how accurately we want to carry out our simulations. If an extremely accurate answer is necessary, we must spend more time to make sure that the details of our simulation match reality. But, if we are running our simulation just to get a ballpark figure, as is the case in this simulation, certain simplifications are warranted. (It's relatively rare for us to roll three doubles in a row, so if our model is inaccurate in this one instance, it won't affect our overall results much. Technically, we'll roll three doubles in a row once every 216 rolls or so, which is less than 1% of the time.)

For our simulation, we want to run several trials, so we'll create an outside loop that loops through each trial. In a single trial we will accumulate rolls of the dice until the sum gets to 40. As we are running the trial, we will count each turn. After the first turn that exceeds a sum of 40 for all the dice rolls in that trial, we stop the trial and count the number of turns it took us to "get around the board." We add this to a variable and will use this sum to calculate our end average. Here is the program in full:

```
# Arup Guha
# 7/6/2012
# Simulates how many turns it takes to go around a Monopoly board
# (without special squares...)
import random

def main():

    # Initialize necessary variables.
    i = 0
    NUMTRIALS = 100000
    sumturns = 0

    # Run each trial
    while i < NUMTRIALS:

        # Set up one trip around the board.
        spot = 0
        turns = 0

        # Stop when we get around the board.
        while spot < 40:

            # Start this turn.
            turns = turns + 1

            # Make sure we don't go more than three times.
            doublecnt = 0
            while doublecnt < 3:

                # Do this roll and update.
                roll1 = random.randint(1,6)
                roll2 = random.randint(1,6)
                roll = roll1 + roll2
                spot = spot + roll

                # Get out if we didn't get doubles.
                if roll1 != roll2:
                    break

                doublecnt = doublecnt + 1

        # Update number of total turns.
        sumturns = sumturns + turns
        i = i+1

    # Print out our final average.
    average = sumturns/NUMTRIALS
    print("Average is",average)

main()
```

Note: When we run this code, it takes several seconds to complete, since so much work is being done in the simulation. Also, it's important to note that python, due to its interpreted nature, runs slower than other languages. A comparable simulation in C takes less than a second.

Though this program is more complex that ones we've previously seen, if you take it apart line by line, there is nothing complicated in its logic. The key is to break down each subtask into logical steps and then convert those steps into code. It this program, we have three levels of loops. The outer most level takes care of multiple trials while the middle loop takes care of a single trial and the inner most loop takes care of a single turn, which can have multiple dice rolls, due to doubles. In regular Monopoly, when someone rolls doubles three times in a row, they go to square number 10, or jail. From there, they lose a turn and can get start rolling again, if they pay $50. If we ignore the monetary consequence, we can simulate this relatively easily, by checking to see if doublecnt is 3, resetting spot to 10 in that case, and adding an extra one to turns.

A great way to improve your programming skills is to think about some event from real life and see if you can write a simulation for it. In this program, we found out that on average, it takes about 5.3 to 5.4 turns to circle the Monopoly board!