While loop assignment

1. Write a function named, isPrime. The function should take a single argument, n.   
   The argument n is expected to be a positive whole number.   
   The function should "stop" with an error if n is either not a whole number or is less than 1.   
   The function should return TRUE if n is a prime number and FALSE if n is not prime.   
   Theoretically, for this type of problem, you could use a for loop or a while loop. However, for this assignment, you are required to use a while loop.   
     
   Examples:  
    > isPrime(3.5)  
    Error in isPrime(3.5) : n must be a positive whole number  
     
    > isPrime(0)  
    Error in isPrime(0) : n must be a positive whole number  
     
    > isPrime(29)  
    [1] TRUE  
     
    > isPrime(35)  
    [1] FALSE
2. Write a function named, getAllPrimesUpToN.   
   The function should take a single argument, n.   
   The function should return a vector of all the prime numbers from 1 up until n.  
   The function should "stop" with an error if n is either not a whole number or is less than 1.   
   Theoretically, for this type of problem, you could use a for loop or a while loop. However, for this assignment, you are required to use a while loop.   
     
   Examples:

> getAllPrimesUpToN(20)

[1] 1 2 3 5 7 11 13 17 19

> getAllPrimesUpToN(100)

[1] 1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83

[25] 89 97

1. Write a function named, getFirstNPrimes. This function should take a single argument, n.   
   The argument n is expected to be a positive whole number.   
   The function should "stop" with an error if n is either not a whole number or is less than 1.   
   The function should return a vector that contains the first n prime numbers.   
     
   For example:

> getFirstNPrimes(6)

[1] 1 2 3 5 7 11

> getFirstNPrimes(10)

[1] 1 2 3 5 7 11 13 17 19 23

HINTS:

* 1. This problem requires that you loop through as many numbers as necessary until you find N prime numbers. Each time you find a new prime number you add it to the answer. Once you have found "n" prime numbers, the loop should stop. This type of problem cannot be done with a for loop, since it is impossible to know in advance how many numbers you will need to check. You will need to use a while loop.
  2. In order to check to see if a particular number is prime, you can call the function isPrime that you created earlier (see above). This is probably the easier (and arguably better) approach.   
       
     Alternatively, instead of calling isPrime, you could theoretically use two loops, one inside the other (i.e. "nested" loops). The outer loop will need to be a while loop that keeps track of what number you are going to check next (this is the loop mentioned above in part a). The inner loop can be a for loop or a while loop - the job of the inner loop would be to check to see if the number you are up to is a prime number.  
       
     You may use either approach to answer this question.

1. Write a function named, rollDice that simiulates the rolling of one or more dice.   
   The function should take a single argument, numberOfDice that indicates the number of dice being thrown.   
   numberOfDice is expected to be a positive whole number. If it is not then the function should stop and display an error message.  
   The function should return a vector of random whole numbers. The value of each number should be randomly chosen to be 1,2,3,4,5 or 6. There should be as many numbers in the vector as indicate by the argument numberOfDice.  
     
   For example:

> rollDice(1)

[1] 3

> rollDice(1)

[1] 6

> rollDice(2)

[1] 2 4

> rollDice(2)

[1] 6 1

> rollDice(5)

[1] 6 3 3 4 6

> rollDice(5)

[1] 4 4 2 6 5  
  
HINTS: Use the runif function. Set the min and max arguments to runif to 1 and 7 respectively. Then use the trunc function to remove the decimal points. This works since the documentation for runif says the following:  
  
runif will not generate either of the extreme values unless max = min or max-min is small compared to min, and in particular not for the default arguments.  
  
Therefore, runif will not generate 7 (or 1 for that matter). Therefore once you truncate the result that you get from runif you will be left with numbers that are either 1,2,3,4,5 or 6.  
  
THINKING DEEPER (you don't have to do this, but it's something to think about): If you follow the hints above, the resulting function, is not 100% "fair". Note that according to the documentation mentioned above if you specify min as 1 and max as 7, then runif will never generate 7.0000 or 1.00000 either. Therefore, technically, the result of your function is very slightly less likely to generate 1s than other numbers. Think about how you could theoretically fix this - One way is that you could set min to 0 (zero) and max to 7 and then truncate the result with trunc. If you get a zero, you keep repeating this process until you got a number that wasn't 0.

1. Write a function named keepRollingUntilSnakeEyes. The function should not take ANY parameters. The function should keep calling rollDice(2) inside of a loop to simulate multiple rolls of two dice. The loop should stop when the roll is two ones (i.e. "snake eyes"). The function should display the values each roll as shown below. The function should return the total number of rolls that were made. See the examples below. Note that in the first two examples below, the last value displayed is the value that is "returned". In the third example below, the return value is captured in a variable and is displayed in a separate command. (also see the next question).   
     
   HINTS:
   1. Use the cat function to display the messages.
   2. Use a variable to keep track of how many rolls took place
   3. keep looping until you get a 1 and a 1

EXAMPLES:   
  
> keepRollingUntilSnakeEyes() # the return value appears after all the messages

roll #1 was: 2 and 1

roll #2 was: 3 and 5

roll #3 was: 6 and 5

roll #4 was: 4 and 1

roll #5 was: 5 and 5

roll #6 was: 6 and 1

roll #7 was: 2 and 3

roll #8 was: 6 and 3

roll #9 was: 5 and 4

roll #10 was: 6 and 3

roll #11 was: 6 and 3

roll #12 was: 4 and 5

roll #13 was: 4 and 1

roll #14 was: 4 and 2

roll #15 was: 1 and 1

[1] 15

> keepRollingUntilSnakeEyes() # the return value appears after all the messages

roll #1 was: 1 and 1

[1] 1

> numRolls <- keepRollingUntilSnakeEyes() # return value is captured in a variable

roll #1 was: 2 and 1

roll #2 was: 4 and 3

roll #3 was: 3 and 6

roll #4 was: 3 and 6

roll #5 was: 4 and 3

roll #6 was: 6 and 4

roll #7 was: 6 and 2

roll #8 was: 3 and 2

roll #9 was: 4 and 1

roll #10 was: 1 and 1  
  
> numRolls # we can now display the value that was returned from the function  
[1] 10

1. Modify the function that you created in the previous question, keepRollingUntilSnakeEyes. In this new version you should define a single argument named, showOutput. The default value of showOutput should be FALSE. If showOutput is TRUE then the messages should be displayed. If showOutput is FALSE then the messages should NOT be displayed. In either case, as with the last question, the functions should return total number of rolls. For example:

> keepRollingUntilSnakeEyes() # this will not show output

[1] 48

> keepRollingUntilSnakeEyes(showOutput = FALSE) # nor will this

[1] 80

> keepRollingUntilSnakeEyes(FALSE) # nor will this

[1] 1

> keepRollingUntilSnakeEyes(TRUE) # this WILL show output

roll #1 was: 4 and 4

roll #2 was: 4 and 6

roll #3 was: 5 and 4

roll #4 was: 4 and 3

roll #5 was: 5 and 6

roll #6 was: 5 and 5

roll #7 was: 2 and 3

roll #8 was: 3 and 1

roll #9 was: 1 and 3

roll #10 was: 3 and 4

roll #11 was: 3 and 2

roll #12 was: 1 and 2

roll #13 was: 6 and 3

roll #14 was: 1 and 1

[1] 14

> keepRollingUntilSnakeEyes(showOutput = TRUE) # this WILL show output

roll #1 was: 2 and 4

roll #2 was: 6 and 1

roll #3 was: 4 and 2

roll #4 was: 5 and 4

roll #5 was: 5 and 6

roll #6 was: 1 and 1

[1] 6

1. Do all of the following steps:  
   1. Write a function named playManyTimes that calls the function keepRollingUntilSnakeEyes in a loop. playManyTimes should take an argument, n, that indicates the number of times the game should be played. playManyTimes should return a vector that contains the number of rolls it took each time the keepRollingUntilSnakesEyes function was called. For example:

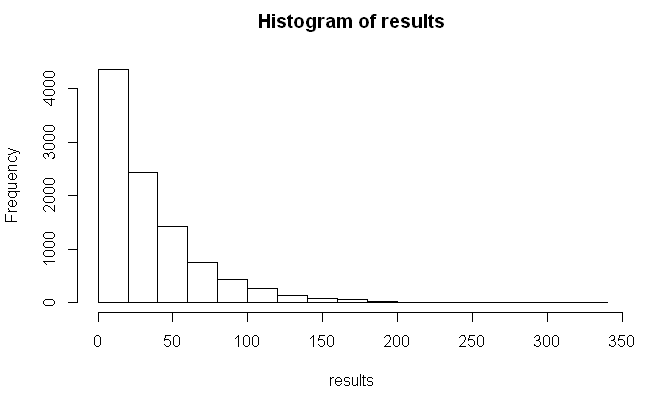
> playManyTimes(3)

[1] 66 1 22

> playManyTimes(10)

[1] 6 27 35 106 38 51 100 1 1 26

* 1. Run **results <- playManyTimes(10000)** to capture the results of playing the game ten thousand times.
  2. Create a histogram of the results with the command, **h <- hist(results)** The histogram should look similar to the example shown below. You can see from this histogram that the function keepRollingUntilSnakeEyes is much more likely to return smaller numbers than to return larger numbers:



* 1. In the previous step the command **h <- hist(results)** captured the output of the hist function in the variable **h** You can display the contents of this variable to examine details about the histogram (see the output below) .The information in **h** is stored in a structure called a "list"., You can access the information in **h**. For example, the **counts** entry in **h** contains the number of values in the results variable that fell into each "bar" of the histogram. The sum of all these counts are 10,000,as should be expected. For example:  
       
     > h  
     $breaks  
      [1] 0 20 40 60 80 100 120 140 160 180 200 220 240 260 280 300 320 340  
       
     $counts  
      [1] 4369 2427 1434 758 432 258 136 78 55 25 12 5 6 1  
     [15] 3 0 1  
       
     $density  
      [1] 0.021845 0.012135 0.007170 0.003790 0.002160 0.001290 0.000680 0.000390  
      [9] 0.000275 0.000125 0.000060 0.000025 0.000030 0.000005 0.000015 0.000000  
     [17] 0.000005  
       
     $mids  
      [1] 10 30 50 70 90 110 130 150 170 190 210 230 250 270 290 310 330  
       
     $xname  
     [1] "results"  
       
     $equidist  
     [1] TRUE  
       
     attr(,"class")  
     [1] "histogram"  
       
       
     > sum(h$counts)  
     [1] 10000

1. The game of "craps" involves a player rolling a pair of dice repeatedly according to the rules shown below.
   1. The first roll:
      1. If the player rolls 7 or 11 he/she wins
      2. If the player rolls 2, 3 or 12, he/she loses
      3. if the player rolls any other number, that number becomes the "point"
   2. All other rolls
      1. If the player hasn't won or lost on the first roll, then the player keeps rolling until either he rolls a 7 or the "point" (i.e. the same value as the very first roll). If the player rolls a 7 he loses. If the player rolls the "point" he wins.

Write a function named, playCraps, that simulates the computer playing a single game of craps. The function should return TRUE if the player wins the simulated game and FALSE if the player loses the game. define a single argument named, showOutput. The default value of showOutput should be FALSE. If showOutput is TRUE then the messages should be displayed. If showOutput is FALSE then the messages should NOT be displayed. In either case, the function, playCraps, should return TRUE if the player wins and FALSE if the player loses. For example:

> playCraps(showOutput = FALSE)

[1] FALSE

> playCraps(showOutput = FALSE)

[1] TRUE

> playCraps(showOutput = FALSE)

[1] FALSE

> playCraps(showOutput = FALSE)

[1] TRUE

> playCraps(showOutput = TRUE)

roll #1: 6

roll #2: 11

roll #3: 5

roll #4: 5

roll #5: 6

WIN

[1] TRUE

> playCraps(showOutput = TRUE)

roll #1: 12

WIN

[1] TRUE

> playCraps(showOutput = TRUE)

roll #1: 7

WIN

[1] TRUE

> playCraps(showOutput = TRUE)

roll #1: 3

LOSE

[1] FALSE

> playCraps(showOutput = TRUE)

roll #1: 6

roll #2: 9

roll #3: 5

roll #4: 9

roll #5: 4

roll #6: 4

roll #7: 9

roll #8: 3

roll #9: 3

roll #10: 2

roll #11: 4

roll #12: 8

roll #13: 10

roll #14: 7

LOSE

1. Calculate the estimated probability of winning a game of craps by doing the following:
   1. Create a function named playCrapsManyTimes that takes a single argument, n.
   2. The function should return a vector that contains the results of calling the playCraps command n times.
   3. Use the function to simulate playing craps ten thousand times
   4. Calculate the percent of times that the player won the game (i.e. total TRUEs divided by total number of games played). Since TRUE is treated as 1 and FALSE as zero, it is possible to use the mean function to calculate this.
   5. For example the following shows that there is only approximately a 47.9% chance of winning the game of craps.:

> results <- playCrapsManyTimes(10000)

> mean(results)

[1] 0.47915