**An Introduction to Programming in R**

Wait! Wait! Don’t run away yet! I admit that learning how to write computer code can appear to be challenging at first glance. However, one of the beauties of R is that it is a fairly easy language to learn. Once you learn a few basic commands, you have the power to create some fairly complex programs. Note that our goal at this stage is not to write the most efficient code ever. Rather, our goal is to write code that works. We can learn about refining the code at a later time.

For now, we will cover 3 basic commands that allow for many simulation studies. These commands are: **if**, **while**, and **for.** We’ll also introduce some logic statements that will allow us to utilize these commands.

**Logic Statements**

The following statements are some commands used to make logic checks:

|  |  |
| --- | --- |
| Statement | What it means |
| > | Is this greater than…? |
| >= | Is this greater than or equal to…? |
| < | Is this less than…? |
| <= | Is this less than or equal to…? |
| == [Note that this is two equal signs.] | Is this equal to…? |
| && | And |
| || [Note that this is two vertical lines.] | Or |

For an example on how to use this, consider the following code:

x=1:6 # We store the numbers 1 through 6 in x.

x<4 # We make a logic check on each value to see if the

# number is less than 4.

R will return:

[1] TRUE TRUE TRUE FALSE FALSE FALSE

because clearly the first three numbers (1, 2, and 3) are less than 4 while the last three numbers (4, 5, and 6) are not.

**The “If” Statement**

The “If” statement asks R to run a logic check. If the statement is true, R will then run the rest of the code. If it is false, R will skip the code that follows the if statement. The general structure is:

if( *put logic check in here* ){

*put code to be run if logic check is true here*

}

Here’s a simple example of the “if” statement in action:

if (a < 4){

print("a is less than 4")

}

**Your turn:**

1. Set the value of a to be equal to 3. Then run the three lines of code above. What do expect to happen?

What actually happened?

2. Set the value of a to be equal to 4. Then run the three lines of code above. What do expect to happen?

What actually happened?

Note that you can combine logic checks using && or ||. For example:

if (a < 4 && a>3){

print("a is a number between 3 and 4")

}

Note that the logic check in the if statement will only return as “true” if both the first part (a<4) and the second part (a>3) are both true.

Similarly, if you use ||, the if statement will only return as “true” if either the first part or the second part is true.

**The “While” Statement**

The “While” statement is very similar to the “if” statement. Like the “if” statement, it will perform a logic check. However, unlike the “if” statement, it will keep running the code in the brackets until the logic check returns “false.” The general structure is as follows:

while( *put logic check in here* ){

*put code to be run if logic check is true here*

}

Here’s an example of the “while” statement in action. Suppose I want to keep squaring the number 2 until I get a value greater than 500000. That is, I want to find the first number in the sequence 2, 4, 16, 256, … that is greater than 500000. The following code will find it for us:

a = 2

while(a < 500000){

a = a^2

}

**Your turn:**

1. What is the first number in the sequence 2, 4, 16, 256, … that is greater than 500000?

2. What is the last number in the sequence 2, 4, 16, 256, … that is less than 500000?

3. What is the first number in the sequence 

that is greater than 10?

**The “For” Statement**

Unlike the “if” and “while” statements, the “For” statement does not perform a logic check. Instead, it repeats the code in the brackets for the number of times specified in the “for” statement. The general structure is as follows:

for( *statement that indicates how many times the code should be executed* ) {

*code that needs to be repeatedly executed*

}

A common command that is used with the “for” statement is:

i in 1:1000

or some other string of integers. The first time the “for” statement is executed, the value of “i” will be 1. The second time, it will be 2. This will continue with the value of i being equal to 1000 the last time the code is executed. Here’s a couple examples of the “for” statement in action:

a = 2

for(i in 1:5){

a = a^2

}

Note that the value of i does not come into play in this code. Instead, I simply take the value of a and square it 5 times, updating the value of a after each square. However, I could take advantage of the value of i as well:

a = 2

for(i in 1:5){

a = a + i

}

The first time through the code, a = 2 [old value of a] + 1 [current value of i].

The second time through the loop, a = 3 [updated old value of a] + 2 [current value of i]

**Your turn:**

1. What is the final value of a after the first for loop?

2. What is the final value of a after the second for loop?

3. Write a “for” loop that prints out the values of 

**Putting it all together**

Now that we have our three basic commands, let’s start to write some simulation code. Consider the following problem. You have a circular board game that has 10 squares on it. You are going to roll a fair die and move your playing piece that many squares forward on the board. You will keep rolling the die and moving your piece until you stop on every square. How many times will you roll the die?

This problem is very hard to do by hand, but relatively simple when we simulate it with computer code. Before we start writing our code, let’s think about what we need.

1. We need our board.

2. We need to know what squares we have or haven’t landed on.

3. We need to know where our playing piece is located on the board.

4. We need to know what we rolled with our die.

5. We need to keep track of how many times we rolled our die.

6. We need to tell the computer when to stop.

Let’s start by setting up our board. We’ll create a vector with 10 values. For simplicity, suppose we code each of the squares with a 0 if we haven’t landed on the square and a 1 if we have landed on the square. To start, all of the values are equal to 0.

board = rep(0, 10)

Let’s start with the playing piece off of the board, in position 0. In the future, it will be anywhere from position 1 to position 10.

piece = 0

To roll the die, let’s use the **sample** command, which allows us to pick a random number from a collection of numbers. We’ll use a standard six-sided die, so we want to choose from the numbers 1 through 6. We’ll also tell R that we only roll the die once each time.

die = sample(1:6, 1)

At the beginning, we haven’t rolled the die yet, so the number of rolls is 0.

rolls = 0

How will we know when to stop rolling? Well, if the board squares are equal to 0 when they haven’t been landed on and they are equal to 1 when they have been landed on, this suggests that when the sum of all the squares is equal to 10 we should stop. This gives us a good idea of a stopping condition for a “while” statement.

while(sum(board) < 10) {

}

Let’s start to put our code together:

board = rep(0, 10) # Set up the board.

piece = 0 # Starting spot for the piece.

rolls = 0 # Starting number of rolls.

while(sum(board) < 10) { # This tells me when to stop.

die = sample(1:6, 1) # Roll the die!

piece = piece + die # Move the piece to its new location.

if(piece > 10){ # Did I move the piece off the board? (For

# example, the code put it on the 12th square.)

piece = piece - 10} # If I did, move it back on the board.

board[piece] = 1 # Update the board to indicate that I’ve landed on

# the square that my piece is on.

rolls = rolls +1 # Update my number of rolls.

}

If I now type in “rolls” I can get the total number of rolls it took. Of course, this is just one run of the code. Suppose I wanted to run this many times and keep track of how many times it took me to land on every square. This is where the “for” loop comes into play.

Let’s suppose we want to run the simulation 1000 times. Each time we run the code, we want to keep track of the number of rolls. To do this, let’s create a variable called “keeptrack”.

keeptrack = NA # We create an empty variable.

for(i in 1:1000){ # Create a “for” loop to run the code 1000 times.

board = rep(0, 10) # Here’s the old code, nested inside the for loop.

piece = 0

rolls = 0

while(sum(board) < 10) {

die = sample(1:6, 1)

piece = piece + die

if(piece > 10){

piece = piece - 10}

board[piece] = 1

rolls = rolls +1

}

keeptrack[i] = rolls # In the ith position of the keeptrack variable, we

# record the number of rolls it took for this run.

} # End the for loop.

Now, keeptrack stores all of the roll counts for each of the 1000 runs. That was done in seconds. (Imagine rolling a die and recording the results for this game 1000 times!) You can use the keeptrack variable to explore the distribution of the number of rolls it takes to land on every square of the board. Note how each time the for loop repeats the code, we reset the board, the position of the piece, and the number of rolls.

**Your turn:**

1. Find the mean and standard deviation of the number of rolls it takes to land on every square on the board.

2. Find the minimum and maximum number of rolls required to land on every square on the board in your simulation. What are the theoretical minimum and maximum?

3. Create a histogram of the number of rolls it takes to land on every square.

4. **Challenge!** Modify the code so that instead you roll the die a total of 15 times. Then, count how many unique squares on the board you’ve landed on. Regarding the number of squares you’ve landed on, find the:

a) mean

b) standard deviation

c) minimum (and theoretical minimum)

d) maximum (and theoretical maximum)

Also, create a histogram that shows how many squares you’ve landed on for the runs of your simulation.

5. **Thought challenge!** Think of a board game where a modification of this code might be useful to answer a question about how the game plays. Give the game and the relevant question.