CHAPTER 13

Examples of Flow Control

This chapter gives some examples of flow control as well as ways to do the examples using indexing. The first example uses nested **for** loops and **if/else** statements. The second example uses the **while** statement. The third example is of nested **for** loops. The fourth example uses a **for** loop, an **if** statement, and a **next** statement. The fifth example is of a **for** loop, a **repeat** loop, an **if** statement, and a **break** statement.

Nested 'for' Loops with an 'if/else' Statement

In this example, we do an element-by-element substitution into a matrix based on an **if/else** test.

First, a two-by-five matrix **x** is generated and the matrix is displayed. Next, two **for** loops cycle through the row and column indices of **x**. At each cycle, a set of **if/else** statements test whether the element in the matrix is greater than five.

If the value of the element is greater than five, the value of the element is replaced with one. If not, control goes to the **else** statement. Within the **else** statement, the value of the element is replaced by zero.

© Margot Tollefson 2019
M. Tollefson, *R Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-4405-0_13

Last, the resultant matrix is displayed. The example follows:

```
> x = matrix( 1:10, 2, 5 )

> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

> for ( i in 1:2 ) {
+   for ( j in 1:5 ) {
+     if ( x[i,j]>5 ) x[i,j]=1
+     else x[i,j]=0
+   }
+ }

> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    1    1
[2,]    0    0    1    1    1
```

Using Indices

Doing the same substitution without loops is easier. First, the matrix **x** is generated and displayed. Next, the elements in **x** are set equal to the new values based on the original values. Note that the order in which the substitution is done matters, since one is less than six. Last, the resultant matrix is displayed. The example follows:

```
> x = matrix( 1:10, 2, 5 )

> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

```
> x[ x<=5 ] = 0
> x[ x>5 ] = 1

> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    1    1
[2,]    0    0    1    1    1
```

On my computer, using a matrix with 43,830 rows and 35 columns, both methods took less than a second.

A 'while' Loop

In this example, a **while** loop is used to find how many iterations it takes for a sum of variables distributed randomly and uniformly between zero and one to be greater than five.

After initially setting the seed for the random number generator and setting **n** and **x** to zero, a **while** loop is started to increment **n** and to sum **x**. A number generated using the random number generator for the uniform distribution is added to **x** at each iteration. When **x** is greater than five, the looping stops. The values for **n** and **x** are printed out. The example follows:

```
> set.seed( 129435 )

> n=0
> x=0

> while ( x<=5 ) {
+   x = x + runif( 1 )
+   n = n + 1
+ }
```

```
> n
[1] 7

> x
[1] 5.179325
```

Using Indices

To do the same task using indices, a vector of uniform random variables is generated of length greater than what would be expected for the result of the sum.

Then, the function `cumsum()`, which creates a cumulative sum along a vector, is used to find when the sum is greater than five. Since the elements of **x** are always greater than zero, the accumulated sum always increases along the vector.

Next, the function `length()` is used to find the number of elements for which the sum is less than or equal to five. Then, the values for **n** and **x** are printed out, where **x** equals **x[n]**.

```
> set.seed( 129435 )

> x = runif( 25 )
> x = cumsum( x )
> n = length( x[x<=5] )+1
> x = x[n]

> n
[1] 7

> x
[1] 5.179325
```

Note that the random number generator is set to the same seed value for both parts of the example, so the results for the two match since the same first seven numbers are generated.

On my computer, if I substitute 1,000,000 for 5 in the preceding examples, and 3,000,000 for 25, the method using indices is almost instantaneous, while the method using looping takes about 5 seconds.

Nested 'for' Loops

Sometimes, the differences between each of the columns of a matrix are needed. In this example, nested **for** loops are used to find the differences.

First, a matrix **x** is generated with two rows and four columns and is assigned column names. Next, the matrix is displayed. Then, a matrix **xp** of zeroes with two rows and six columns is generated to hold the result of the differences, and the matrix is assigned blank column names.

Next, a counter **k** for the columns in the matrix **xp** is set to zero. As the two **for** loops increment, **k** will increase by one at each step.

Then, the two **for** loops are run. In the loops, the elements of **xp** are filled with differences between the different columns in **x**. The two loops loop through the columns in the matrix **x** in such a way that no column combinations are repeated and the two columns are never the same. At each step, the columns of **xp** are assigned names based on the names in **x**.

Last, the resulting matrix **xp** is displayed. The example follows:

```
> x = matrix( 1:8, 2, 4 )
> colnames( x ) = paste( "c", 1:4, sep="" )

> x
     c1 c2 c3 c4
[1,]  1  3  5  7
[2,]  2  4  6  8

> xp = matrix( 0, 2, 6 )
> colnames( xp ) = rep( "", 6 )
> xp
```

```
[1,] 0 0 0 0 0 0
[2,] 0 0 0 0 0 0

> k=0

> for ( i in 1:3 ) {
+   for ( j in (i+1):4 ) {
+     k = k+1
+     xp[,k] = x[,i]-x[,j]
+     colnames( xp )[k] = paste( colnames(x)[i], "-",
                                 colnames(x)[j], sep="" )
+   }
+ }

> xp
     c1-c2 c1-c3 c1-c4 c2-c3 c2-c4 c3-c4
[1,]    -2    -4    -6    -2    -4    -2
[2,]    -2    -4    -6    -2    -4    -2
```

Note that the number of columns in **xp** equals p(p-1)/2, where **p** is the number of columns in **x**.

Using Indices

To do this problem using indices, two vectors of indices are created.

First, the initial matrix **x** is generated, assigned column names, and displayed. Then, two sets of indices of the same length, **ind.1** and **ind.2**, are created. The respective indices in the two sets are never the same, and all possible combinations are present and present only once.

Next, the resultant matrix **xp** is created by subtracting the columns of **x** in the second index set from the columns of **x** in the first index set. Next, the column names for **xp** are created and assigned using `paste()` and the two index sets.

Last, the matrix **xp** is displayed. The example follows:

```
> x = matrix( 1:8, 2, 4 )
> colnames( x ) = paste( "c", 1:4, sep="" )
> x
     c1 c2 c3 c4
[1,]  1  3  5  7
[2,]  2  4  6  8

> ind.1 = rep( 1:3, 3:1 )
> ind.1
[1] 1 1 1 2 2 3

> ind.2 = numeric( 0 )
> for( i in 2:4 ) ind.2 = c( ind.2, i:4 )
> ind.2
[1] 2 3 4 3 4 4

> xp = x[,ind.1] - x[,ind.2]
> colnames( xp ) = paste( "c", ind.1, "-c", ind.2, sep="" )

> xp
     c1-c2 c1-c3 c1-c4 c2-c3 c2-c4 c3-c4
[1,]    -2    -4    -6    -2    -4    -2
[2,]    -2    -4    -6    -2    -4    -2
```

Note that a **for** loop is used to create the second set of indices. Also, column indices are repeated in both sets of indices.

For large matrices, the second method is faster than the first. On my computer, column differences for two matrices each with 43,830 rows and 35 columns were found by the two methods. The two methods both gave the same 43,830-by-595 matrix. The looping method took over 1.0 minute, and the indexing method took less than 1.0 second.

A ‘for’ Loop, ‘if’ Statement, and ‘next’ Statement

In this example, standard normal random numbers are generated and compared to 1.965. Only those values that are less than or equal to 1.965 are kept.

First, the seed for the random number generator is set to an arbitrary value. Then, x is set equal to a numeric NULL value. In the **for** loop that comes next, for 10,000 iterations, a standard normal random number is generated at each iteration. If the number is larger than 1.965, the next loop starts. Otherwise, the number is added to a vector of numbers. A histogram is plotted of the final vector. See Figure 13-1 for the result. The example follows:

```
> set.seed( 69785 )

> x = numeric( 0 )

> for ( i in 1:10000 ) {
+   x2 = rnorm( 1 )
+   if ( x2>1.965 ) next
+   x = c( x, x2 )
+ }

> hist( x )

> box()
```

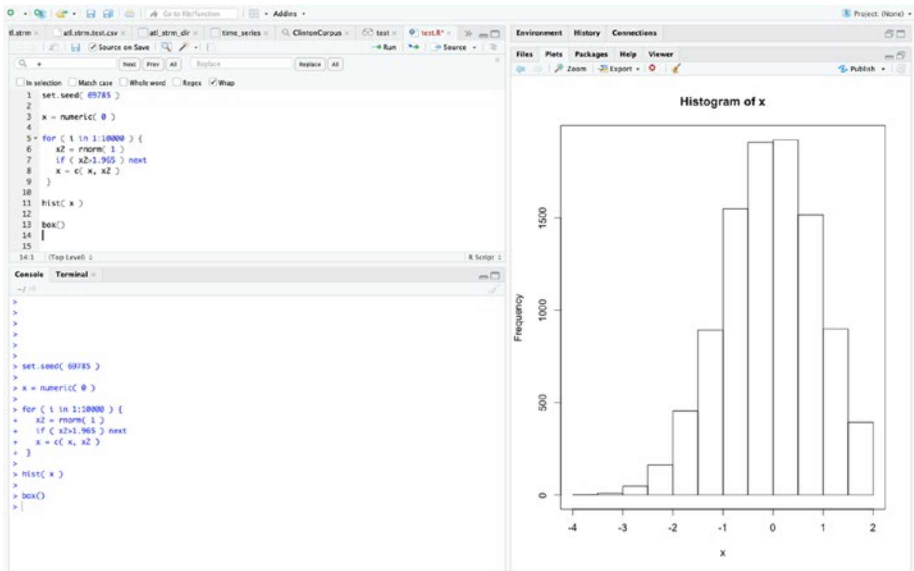

Figure 13-1. *Using a loop to generate a histogram of random standard normal variates that are less then 1.965*

Using Indices

Using indices is much simpler. First, the random number generator seed is set to the same value as for the previous example. Next, a vector of standard normal random variables of length 10,000 is generated. Next, only those values in the vector that are less than or equal to 1.965 are kept. Last, a histogram of the vector is generated. The histogram is shown in Figure 13-2. The example follows:

```
> set.seed( 69785 )

> x = rnorm( 10000 )
> x = x[ x<=1.965 ]

> hist( x )

> box()
```

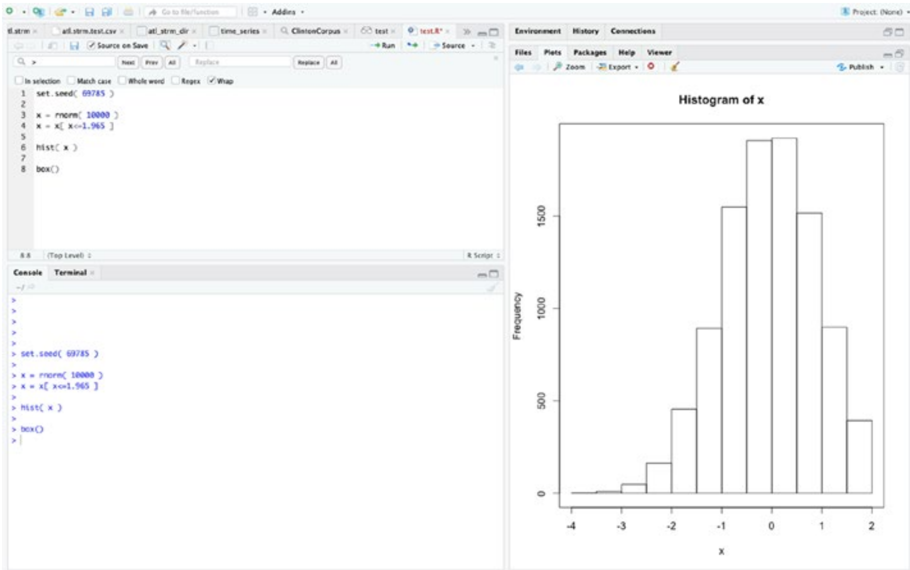

***Figure 13-2.** Using indices to generate a histogram of random standard normal variates that are less then 1.965*

Note that the two histograms are the same since the seeds are the same and the same 10,000 numbers are used.

If 10,000 is increased to 100,000 above, on my computer the method using loops takes about 34 seconds while the method using indices takes less than 1 second.

A ‘for’ Loop, a ‘repeat’ Loop, an ‘if’ Statement, and a ‘break’ Statement

In this example, random samples of size 100 of standard normal numbers are generated within a **repeat loop**. The **repeat loop** is within a **for** loop that goes through 10,000 iterations.

For each sample, the sum of the sample is divided by the ten and then compared to 1.965. (Since the expected value of the generated numbers is zero, the standard error is one, and the numbers are independent, the sample sum divided by ten is a standard normal variate.) If the value is less than 1.965, then the **repeat** loop continues. Otherwise, the **repeat** loop stops, the number of times through the loop is recorded, and the next **for** loop starts. At the end, the vector of the numbers of times through the loop is plotted in a histogram, and the mean and median of the numbers of times is found.

First, the seed for the random number generator is set. Then, a vector **n.hist** is created to hold the results, with a place for each iteration of the **for** loop. Next, the **for** loop opens, and the counter **n** is set to zero. Then, the **repeat** loop opens.

At the beginning of the **repeat** loop, the counter **n** is incremented by one. Then, the sample is taken, divided by ten, and summed. The result is set equal to **x**. Next, the value of **x** is compared to 1.965 in an **if** statement. If the value is greater than 1.965, then **n.hist** for index **i** is set equal to the counter **n** and a **break** statement breaks the function out of the **repeat** loop. Otherwise, the **repeat** loop continues looping.

At the end, `hist()` is run to create a histogram of **n.hist**, `mean()` is run to find the mean of **n.hist**, and `median()` is run to find the median of **n.hist**. See Figure 13-3 for the histogram. The example follows:

```
> set.seed( 69785 )

> n.hist = numeric( 10000 )
```

```
> for ( i in 1:10000 ) {
+   n=0
+   repeat{
+     n=n+1
+     x=sum( rnorm( 100 )/10 )
+     if ( x>1.965 ) { n.hist[i]=n; break }
+   }
+ }

> hist( n.hist, breaks=25, xlim=c( 0, 500) )

> box()

> mean( n.hist )
[1] 40.4769

> median( n.hist )
[1] 28
```

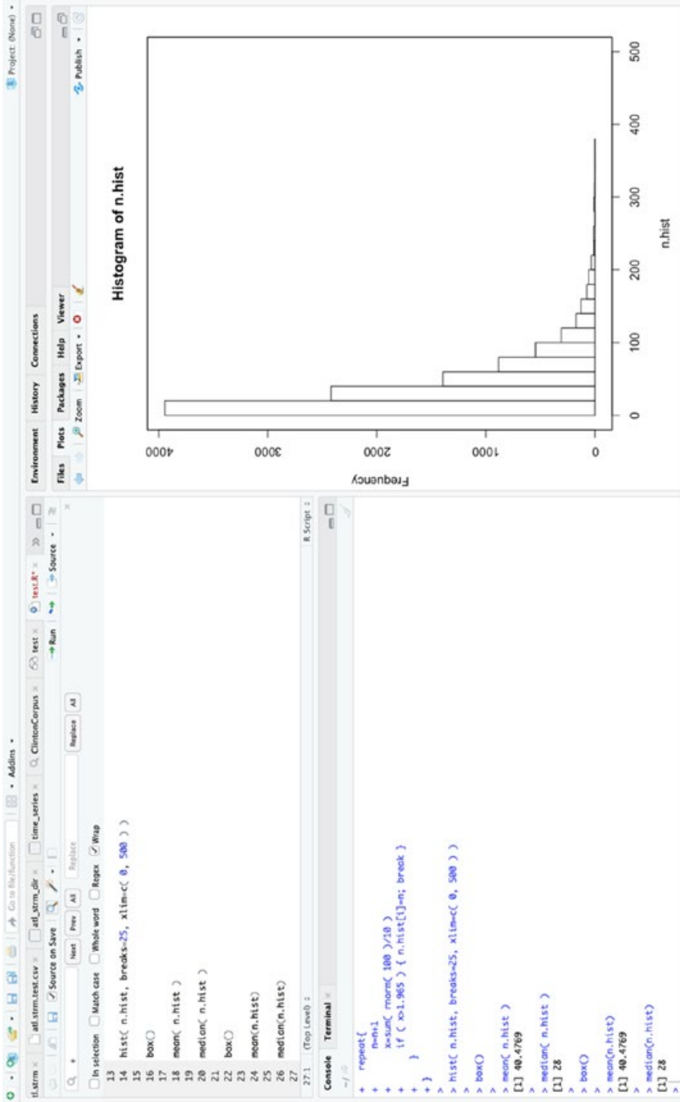

***Figure 13-3.** The numbers of times needed until the result exceed 1.965 for sums of 100 standard normal variable divided by 10—using a **for** loop*

Note that the mean is close to 40, which is the expected number of trials necessary on average to see an event with a probability of 0.0247 of occurring. However, the median is much smaller since the distribution is highly skewed.

Using Indices

To do this example using indices, we found the **repeat** loop necessary, but that the **for** loop could be dispensed with.

Once again, the random number generator seed is set—to the same number as in the first part of the example—and **n.hist** is defined **numeric** with 10,000 elements. Then, the counter **n** is set to zero, the counter **cl.sv** is set to zero, and the counter **n.col** is set to 10,000.

Next, the **repeat** loop opens. The matrix **x** is defined as a matrix with 100 rows and **n.col** columns (initially 10,000). The elements of **x** are randomly generated standard normal numbers and the number of elements is the product 100 and **n.col**.

Next, the function `apply()` is used to sum each column of the matrix, and the result is assigned to **x**. Then, **x** is divided by 10. Next, the length of the vector containing those elements of **x** that are larger than 1.965 is found and assigned to **x**.

Then, **x** is added to **cl.sv** so that **cl.sv** contains the number of columns for which a result larger than 1.965 has been found. Then, **n** is incremented by one. Next, **x** values of **n.hist** are set equal to **n**, where **cl.sv** and **x** are used to say where along the vector **n.hist** to put the value of **n**.

Next, **n.col** is decremented by the value of **x**. The **repeat** loop continues until **n.col** equals zero. At each iteration, **n** increases by one.

The histogram of **n.hist** is generated using `hist()`, the mean of **n.hist** using `mean()`, and the median of **n.hist** using `median()`. See Figure 13-4 for the histogram. The example follows:

```
> set.seed( 69785 )
```

```
> n.hist = numeric( 10000 )
> n = 0
> cl.sv = 0
> n.col = 10000

> repeat{
+   x = matrix( rnorm( n.col*100  ), 100, n.col )
+   x = apply( x, 2, sum )
+   x = x/10
+   x = length( x[ x>1.965 ] )
+   cl.sv = cl.sv + x
+   n = n+1
+   n.hist[ ( cl.sv-x+1 ):cl.sv ] = n
+   n.col = n.col-x
+   if (n.col==0) break
+ }

> hist( n.hist, breaks=25, xlim=c( 0, 500) )


> box()


> mean(n.hist)
[1] 40.5015

> median(n.hist)
[1] 28
```

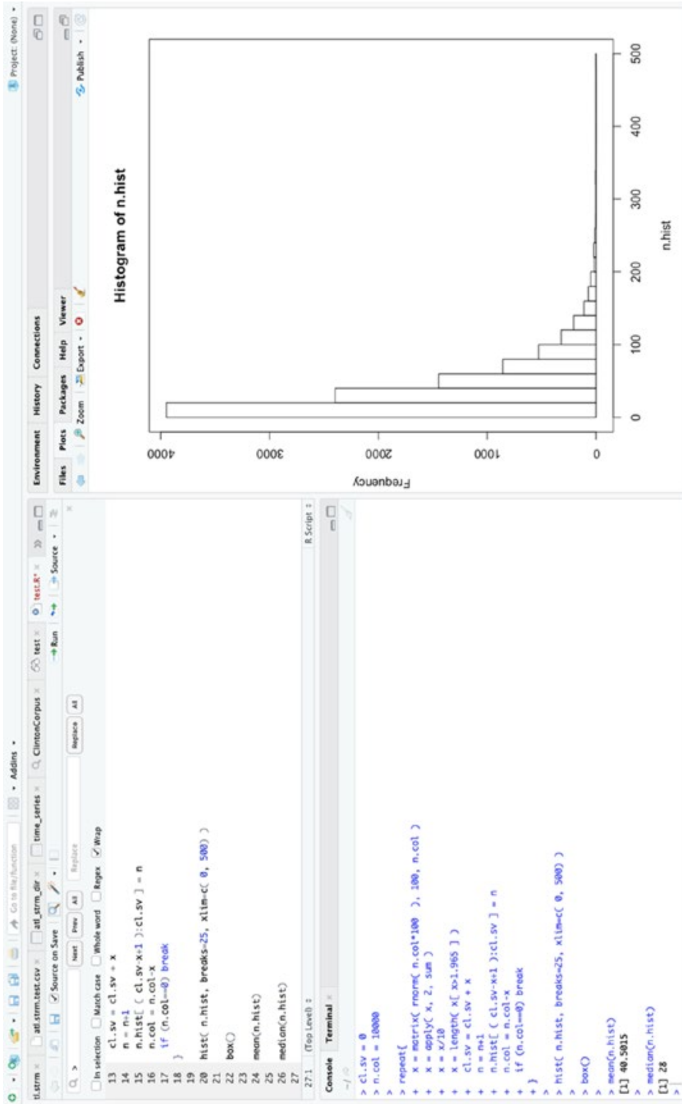

***Figure 13-4.** The numbers of times needed to exceed 1.965 for sums of 100 standard normal variable divided by 10—using indices*

Once again, the mean is close to 40 and the median is 28.

Both methods use about the same amount of time. If 10,000 is replaced by 100,000 above, then the looping method takes about 44 seconds and the indexing method takes about 45 seconds on my computer.

Since the process of generating the random samples is different between the two methods, the results for the two methods are not identical even though the seed for the random number generator is the same.