Logic and Control

Statistics 427: R Programming

Module 7

2020

Logic and Control

R has many ways to use vectors, and comes with tools for extracting and manipulating (the good kind) information in vectors.

Logical Comparison Operators and Logical Vectors

In previous modules there have been some examples of **logical comparison operators** and **logical vectors** .

Logical Comparison Operators I

The *logical* part is in the answer to the comparison. The example shows the values of `x` that are less than or equal to `y`. The answers in that are given as `TRUE` (yes that element of `x` is less than or equal to corresponding element of `y`) or `FALSE` (no that element of `x` is not less than or equal to corresponding element of `y`).

```
x=c(3,0,-2,-5,7,2,-1)
y=c(2,4,-1,0,5,1,-4)
x<=y
```

```
[1] FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

Logical Comparison Operators II

Sign	Definition
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Logical Comparison Operators III

Character vectors can be used as well with the comparisons. They will compare the number of characters.

```
a=c('ann','gretchen','maria','ruth','wendy')
b=c('bruce','ed','robert','seth','thomas')
a>=b
```

```
[1] FALSE  TRUE FALSE FALSE  TRUE
```

Boolean Operations I

Now we begin to combine logical comparisons in Boolean operations. The most commonly known Boolean operations are "and", "or", and "not".

Sign	Definition
`&`	and
`|`	or
`!`	not

Boolean Operations II

```
x=c(3,0,-2,-5,7,2,-1)
y=c(2,4,-1,0,5,1,-4)
(x-y>-2) & (x-y<2)
```

```
[1]  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

```
(x!=y)
```

```
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Boolean Operations III

When you are using the index method with more than a one-dimensional vector, as in a data frame with at least 2 vectors, then we have to account for the row number and column number of the value you want to extract from the data frame. As in `x[r,c]` with `r` meaning row and `c` meaning column (reads as the elements of row and column of vector x).

Leaving either the `r` or `c` blank (like `x[,c]` or `x[r,]`) will choose *all* of the rows or columns of the data frame. Using a negative sign will choose all *except* the one(s) stated (like `x[,-1]` where all rows are chosen (it is blank in the `r` argument) and all *except* the first column is chosen).

Boolean Operations IV

```
x=c(3,7,5,-2,0,-8)
y=c(2,7,-5,4,-1,6)
z=c('jan','feb','mar','apr','may','jun')
# OR
zalt=month.name[1:6] # new function to extract first 6 months
monthly.numbers=data.frame(x,y,zalt)
monthly.numbers[4,1]
```

```
[1] -2
```

Boolean Operations V

```
monthly.numbers[4,2] # obs from 4th row and 2nd column
```

```
[1] 4
```

```
monthly.numbers[1:3,c(1,3)] # first 3 rows and first and third columns
```

```
  x     zalt
1 3  January
2 7 February
3 5    March
```

```
monthly.numbers[5,] # 5th row, all columns
```

```
  x  y zalt
5 0 -1  May
```

Boolean Operations VI

```
monthly.numbers[1:3,] # first 3 rows and first and all columns
```

```
  x  y     zalt
1 3  2  January
2 7  7 February
3 5 -5    March
```

```
monthly.numbers[-4,-2] # all rows but the 4th and all columns but the 2nd
```

```
   x     zalt
1  3  January
2  7 February
3  5    March
5  0      May
6 -8     June
```

Conditional Statements

`R` will execute a statement or statements conditionally using `if` and `if/else` statements.

The `if` function will execute statement(s) based on meeting a condition. By itself it may not provide enough flexibility if the condition could have another outcome.

`if/else` will execute one set of statement(s) based on meeting a condition, and other set(s) of statement(s) if the condition is not met. There can be several `else` statements within the `if` function.

if

General form of `if(cond) expr`
`cond`: the condition to be met
`expr`: executable statement(s)

```
if(cond){
executable statement 1
executable statement 2
```

⋮
}

`if/else`

General form of `if(cond) expr else altexpr`
`cond`: the condition to be met
`expr`: executable statement(s) if condition is met
`altexpr`: executable statement(s) if condition is not met or another condition is met

```
if(cond){
executable statement 1a
executable statement 1b
⋮
} else {
executable statement 2a
executable statement 2b
⋮
}
```

`if` example

This will only give feedback if the condition can be met and will produce output.

```r
x=3; rm(y); rm(z) # removing because they were use earlier
if (x<=2){
  y=5
  z=5
}
```

`y; z` will give an error since they did not meet the condition

`if/else` example

```r
rm(x) # removing because was stored as something else earlier
x=3
if (x<=2){
  y=5
  z=5
} else {
  y=6
  z=6
}
y; z
```

```
[1] 6
```

```
[1] 6
```

Computer Dice I

Simulation to illustrate the `if` statement. The simulation will generate a vector of desired length with just 0s and 1s in it (0=failure, 1=success), each element being the outcome of a random event. Many things could be simulated like the atom of a radioisotope either decays in a unit of time or is does not decay in a unit of time, a baseball player either gets a hit in a defined at-bat (at-bat excludes walks, hit by the pitcher, and sacrifices) or does not get a hit during the at-bat, and an adult bird either survives during the coming year or it does not survive the coming year.

Computer Dice II

For each experiment (atom isotope decaying or not, hitting during at-bat or not, bird survives or dies) we "roll" the dice to get an outcome of either 0 or 1. If there is a success (1) we can assign it probability p and if it is a failure (0) we can assign it a probability of $1 - p$.

$$P(1) = p \; and \; P(0) = 1 - p$$

Computer Dice III

We will use the uniform random number generator as our "dice". The generator will produce a randomly picked number between 0 and 1. We will draw a uniform random number and compare it to p. If the value of the uniform random number is less than or equal to p, we will say that a 1 occurred on the current repetition otherwise we will say that a 0 has occurred. With the use of the word `if` in the previous sentence, that tells us that there are two outcomes (conditions) that could be met.

We will use `runif()` function to return a vector of randomly generated values from a uniform distribution, set an `if/else` to sort out the values in relation to p.

[`runif()` will produce different values every time you run it]

Computer Dice IV

We will create a function (it will be our generator) with `runif()` and `if/else`

```
gen=function(n,p){
  u=runif(n)
  x=numeric(n)
for (i in 1:n){
  if(u[i]<=p){
    x[i]=1
  } else {
      x[i]=0
    }
}
  return(x)
}
gen(30,.25)
```

```
 [1] 0 0 0 1 0 0 1 0 1 0 1 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
```

Computer Dice V

Each time you do this the values may vary a bit since you are taking *random* samples.

An average major league baseball (MLB) player usually has a batting average around .260, meaning that during any given at-bat, the player has a probability of 0.26 (26% chance) of getting a hit. The issue is that in any short sequence of at-bats, the player could have a great streak or a bad one. The at-bats number is dependent on the sample size and short sequences can really hurt or help the player, as management tends to make decisions based on short-run outcomes, rather than long-run outcomes, which probabilities are supposed to be based upon.

There IS Crying In Baseball

Using `gen` fxn and loop for MLB batting average

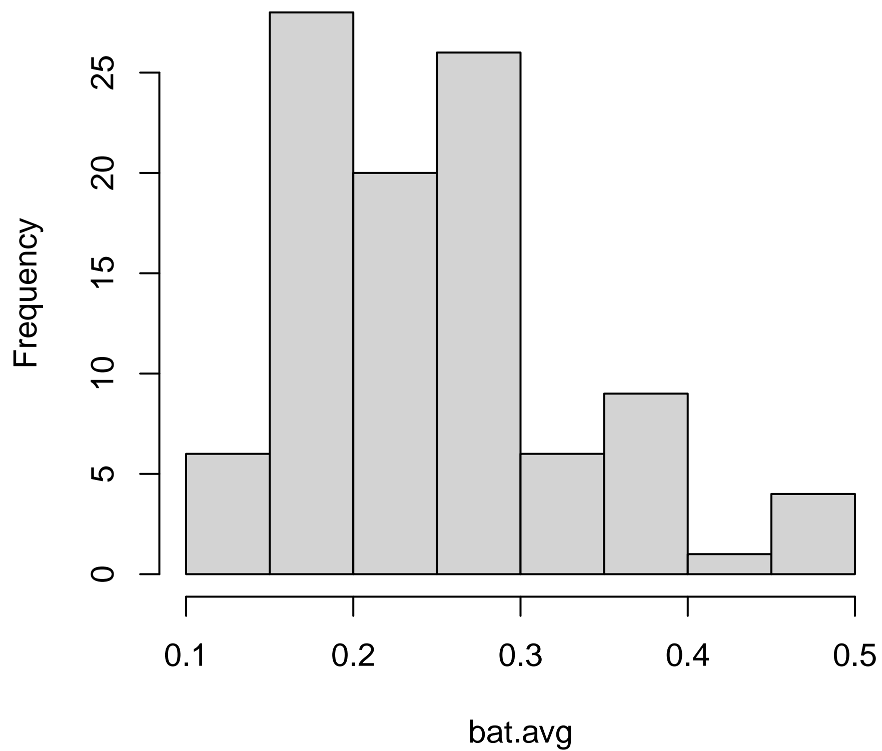
```
n=30; p=.26 # avg at-bats in a set and bat avg
n.sets=100 # number of at-bats to sim
bat.avg=numeric(n.sets) # will contain new values

for (i in 1:n.sets){
  bat.avg[i]=sum(gen(n,p))/n # number of hits in n at-bats divided by n
}
bat.avg
```

```
  [1] 0.3666667 0.2000000 0.2000000 0.2000000 0.1666667 0.2333333 0.2000000
  [8] 0.1333333 0.1666667 0.1666667 0.4666667 0.2333333 0.3000000 0.1333333
 [15] 0.2000000 0.3666667 0.3000000 0.3333333 0.2666667 0.2666667 0.2000000
 [22] 0.2333333 0.3000000 0.4000000 0.2333333 0.2333333 0.2000000 0.2666667
 [29] 0.2333333 0.3666667 0.4000000 0.1666667 0.2000000 0.1666667 0.3333333
 [36] 0.1333333 0.2333333 0.3000000 0.2333333 0.1666667 0.2000000 0.1666667
 [43] 0.4000000 0.2333333 0.2333333 0.2333333 0.3000000 0.3666667 0.2333333
 [50] 0.2666667 0.2333333 0.2333333 0.1666667 0.3000000 0.2666667 0.3333333
 [57] 0.3333333 0.2333333 0.1000000 0.2000000 0.3000000 0.1666667 0.2666667
 [64] 0.3000000 0.1666667 0.5000000 0.2333333 0.1666667 0.5000000 0.2666667
 [71] 0.1333333 0.1000000 0.2666667 0.2666667 0.2000000 0.4333333 0.2666667
 [78] 0.5000000 0.3000000 0.2666667 0.2666667 0.2666667 0.3000000 0.1666667
 [85] 0.3333333 0.3666667 0.2000000 0.3666667 0.2000000 0.2333333 0.2333333
 [92] 0.2666667 0.2666667 0.2000000 0.3333333 0.2333333 0.2333333 0.1666667
 [99] 0.1666667 0.2666667
```

There IS Crying In Baseball histogram

```
hist(bat.avg)
```



There IS Crying In Baseball stemplot

```
stem(bat.avg) # for the heck of it
```

```
  The decimal point is 1 digit(s) to the left of the |

  1 | 003333
  1 | 77777777777777
  2 | 0000000000000033333333333333333333
  2 | 7777777777777777
  3 | 0000000000333333
  3 | 777777
  4 | 0003
  4 | 7
  5 | 000
```