Logical Statements with Applications

Jose Toledo Luna

2024-07-21

Table of contents

Logical Statements

There are only two logical values, `TRUE` and `FALSE`. In R, we can abbreviate `TRUE` with `T` and `FALSE` with `F`. They can be interpreted as any option corresponding to a binary choice. For example, yes/no, do/don't, satisfied/not satisfied or even `1/0`.

A basic way to define a logical statement is using a **relational operator** to compare two expressions. For example, we may ask ourselves *"is x less than a certain number ?"* or using a real world example from the `mtcars` dataset *"how many cars have more than 18 miles per gallon?"*

Relational operators

The table below summarizes some of the **relational operators** available in R:

Operator	Interpretation	Basic Example	Result
==	Equal to	5 == 5	`TRUE`
!=	Not equal to	4 != 5	`TRUE`
>	Greater than	4 > 5	`FALSE`
<	Less than	4 < 5	`TRUE`
<=	Less than or equal to	4 <= 5	`TRUE`
>=	Greater than or equal to	4 >= 5	`FALSE`

From the table above we consider single numbers as our two expression to compare, but we can extend this idea to vectors, data.frames, matrices of various data types. When applying relational operators to vectors it is important to know they are being compared *element-wise*.

We first start off by comparing a vector with a single number

```
c(1,3,5,7,9) < 5
```

```
#> [1]  TRUE  TRUE FALSE FALSE FALSE
```

Interpretation: Is 1 less than 5? is 3 less than 5? is 5 less than 5? is 7 less than 5? is 9 less than 5?

The output from the above example is a `logical` vector

```
class(c(1,3,5,7,9) < 5)
```

```
#> [1] "logical"
```

with `TRUE/FALSE` if the given condition was satisfied or not. What if we were given the question *"How many values of x are smaller than some number?"*

```
sum( c(1,3,5,7,9) < 5 )
```

```
#> [1] 2
```

we can then apply the `sum()` function to count how many `TRUE` were in our logical vector. This will be very useful when we have very large vectors and we can't count how many `TRUE` were in our vector manually.

Below are some examples applying relational operators to compare two vectors of the same length

```r
c(1,2,3,4) < c(5,4,3,2)
```

```
#> [1]  TRUE  TRUE FALSE FALSE
```

Interpretation: Is 1 less than 5? is 2 less than 4? is 3 less than 3? is 4 less than 2?

```r
c(1,2,3,4) <= c(5,4,3,2)
```

```
#> [1]  TRUE  TRUE  TRUE FALSE
```

Interpretation: Is 1 less than or equal to 5? is 2 less than or equal to 4? is 3 less than or equal to 3? is 4 less than or equal to 2?

Another topic to consider is comparing two strings. While this can be a more advance topic we only consider the simplest scenario in which we compare case-sensitive strings.

```r
string1 <- 'Hello'
string2 <- 'hello'
```

while the above strings contain the same characters in the same order, if we compare them directly they will be considered different

```r
string1 == string2
```

```
#> [1] FALSE
```

Interpretation: are string1 and string2 identical?

However, if were are interested in seeing if they contain the same characters regardless of the case sensitivity, we can use `tolower()` function as follows

```r
tolower(string1)
```

```
#> [1] "hello"
```

```r
tolower(string2)
```

```
#> [1] "hello"
```

`tolower()` will convert any upper-case character in a vector into lower-case character.

```
tolower(string1) == tolower(string2)
```

```
#> [1] TRUE
```

Since all the characters are now lower-case, and both strings contain the same characters in the same order then they are now identical.

For more advanced examples in comparing strings check out the following blog post *(Optional)*

Logical operators

In practice, we often need to use multiple conditions to make certain decisions. For example, you have a personal rule that if there is no homework *AND* you don't have class, then you will go out with your friends. Now, explore what happens to this rule when *OR* is used instead of *AND*, also what happens when negation (*NOT*) is added to one or both clauses.

The table below summarizes some of these logical operators

Operator	Interpretation	Basic Example	Result
!	**NOT** *If the condition is true, logical NOT operator returns as false*	! (5 == 5)	`FALSE`

&	**AND** *(element-wise) Returns true when both conditions are true*	TRUE `&` TRUE
		TRUE `&` FALSE
		FALSE `&` TRUE
		FALSE `&` FALSE
			`TRUE`
			`FALSE`
			`FALSE`
			`FALSE`

&&	**AND** *(single comparison) Same as above but for single comparison*	*(same as & above)*	*(same as & above)*

|	**OR** *(element-wise) Returns true when at-least one of conditions is true*	TRUE `|`TRUE
		TRUE `|` FALSE
		FALSE `|` TRUE
		FALSE `|` FALSE
			`TRUE`
			`TRUE`
			`TRUE`
			`FALSE`

Operator	Interpretation	Basic Example	Result
||	**OR** *(single comparison) Same as above but for single comparison*	*(same as | above)*	*(same as | above)*

The difference between *element-wise* and *single comparison* can be seen in the examples below

```
c(TRUE,TRUE,FALSE,FALSE) | c(TRUE,FALSE,TRUE,FALSE)
```

```
#> [1]  TRUE  TRUE  TRUE FALSE
```

Interpretation: TRUE or FALSE, TRUE or FALSE, FALSE or TRUE, FALSE or FALSE

Element-wise will return a vector of logical values, one for each pair of logicals combined. Whereas, *single comparison* only compares the first two elements of the logical vectors and will return a single logical value

```
age <- 20
age == 18 || age <= 21
```

```
#> [1] TRUE
```

Interpretation: Is age 18 OR less than or equal to 21 ?

```
age > 10 && age < 30
```

```
#> [1] TRUE
```

Interpretation: Is age greater than AND less than 30?

Consider a more complicated example of holding office in the United States. The president must be a natural-born citizen of the United States, be at least 35 years old, and have been a resident of the United States for 14 years

```
candidate_age <- 40
candidate_birth <- 'United States'
candidate_residance_years <- 10
```

We have a candidate who is 40 years old, was born in the United States but for some reason they have only been a resident of the United States for 10 years. Clearly, this candidate is not eligible to become our next president. We demonstrate this using logical operators

```
candidate_age >= 35
```

```
#> [1] TRUE
```

Interpretation: Is the candidate at least 35 years old?

```
candidate_birth == 'United States'
```

```
#> [1] TRUE
```

Interpretation: Is the candidate born in United States?

```
candidate_residance_years >= 14
```

```
#> [1] FALSE
```

Interpretation: Has the candidate been a resident for at least 14 years?

Putting all of the above together,

```
(candidate_age >= 35) && (candidate_birth == 'United States') && (candidate_residance_years >
```

```
#> [1] FALSE
```

Interpretation: TRUE AND TRUE AND FALSE

Since one of the conditions fails the entire statement will be false.

Subsetting

Vectors

Now that we have an idea of how to construct logical statements, we can apply them to subset our data based on a given condition

Consider the following vector `dat` with 18 values

```
dat <- c(11, 13, 18, 3, 2, 24, 10, 8, 5,
         13, 3, 23, 7, 25, 17, 20, 11, 17)
```

We will subset `dat` based on the following conditions:

1. How many values are bigger than 10?

```
dat > 10
```

```
#>  [1]  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
#> [13] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
sum(dat > 10 )
```

```
#> [1] 11
```

while knowing how many values are bigger than 10 is useful, we may only want to keep those values and not the ones that are smaller than 10.

2. Keep the values that are bigger than 10?

If given a vector, the way to subset it based on a condition is as follows: `vector[ condtion ]`. Our condition is all the values that are bigger than 10, that is `dat > 10`

```
dat[ dat > 10 ]
```

```
#>  [1] 11 13 18 24 13 23 25 17 20 11 17
```

3. How many values are exactly 11 ?

Our condition is `dat == 11`,this should only return two `TRUE`, and after using the `sum()` function to count them we obtain

```
sum(dat == 11)
```

```
#> [1] 2
```

If we wanted to extract these values from `dat` we would run

```r
dat[ dat == 11 ]
```

```
#> [1] 11 11
```

Next we use the `birth` dataset for the following examples

4. How many females were in this dataset?

```r
birth_dat <- read.csv(file = "/Users/jtoledo/Desktop/Projects/csuf-math-338/data/births.csv")
```

First we extract the values from the `Gender` column and store them in a variable called `gender_vec`

```r
gender_vec <- birth_dat$Gender
```

```r
unique(gender_vec)
```

```
#> [1] "Male"   "Female"
```

> **Warning**
>
> Recall strings are case-sensitive, so you must spell 'Female' exactly as it appears above

Then we subset this vector to only include females

```r
females_vec <- gender_vec[gender_vec == 'Female']
```

```r
unique(females_vec)
```

```
#> [1] "Female"
```

Now our vector only contains females, we can use `length()` to count how many females were in this dataset

```r
length(females_vec)
```

```
#> [1] 961
```

An easier approach would be to simply create the variable `gender_vec` and count how many females are in that vector

```r
sum(gender_vec == 'Female')
```

```
#> [1] 961
```

Data Frames

Considering *example 4* in the vectors section of subsetting, we are extracting solely the values from a specific column based on a given condition. However, in some scenarios we may want to preserve all other information *(columns)* from our dataset after subsetting our data.

Data frames have the following structure `data[rows,columns]`. The first argument inside the brackets will specify the rows and the second argument will specify the columns. We can apply all of the subsetting techniques we covered in the vectors within the rows, columns, or both rows and columns `data[condition for rows, condition for columns]`

For example, if we wanted to subset the `births` dataset to only include females

```r
is_female <- birth_dat$Gender == 'Female'
```

```r
birth_dat[is_female, ]
```

Interpretation: Subset the rows to only include females, keep all the other columns

```
#>     X Gender Premie weight Apgar1 Fage Mage Feduc Meduc TotPreg Visits
#> 5   5 Female     No    119      8   30   19    12    12       2     12
#> 9   9 Female     No    126      7   31   31    12    12       2      8
#> 10 10 Female     No    131      8   29   28     9     9       3      9
#>      Marital Racemom Racedad Hispmom Hispdad Gained     Habit MomPriorCond
#> 5  Unmarried   Black Unknown NotHisp Unknown     20 NonSmoker         None
#> 9    Married   White   White Mexican Mexican     30 NonSmoker         None
#> 10   Married   White   White Mexican Mexican     33 NonSmoker         None
#>    BirthDef DelivComp BirthComp
#> 5      None      None      None
#> 9      None      None      None
#> 10     None      None      None
```

You will notice that we only applied a condition to the rows argument and not the columns argument. In the case where one of the arguments is left blank, then no condition will be applied to the respective argument.

For practice, consider the following examples

1. Create a new data frame containing the columns: Gender, weight, and Habit

We can use `colnames()`

```
colnames(birth_dat)
```

```
#>  [1] "X"            "Gender"       "Premie"       "weight"       "Apgar1"
#>  [6] "Fage"         "Mage"         "Feduc"        "Meduc"        "TotPreg"
#> [11] "Visits"       "Marital"      "Racemom"      "Racedad"      "Hispmom"
#> [16] "Hispdad"      "Gained"       "Habit"        "MomPriorCond" "BirthDef"
#> [21] "DelivComp"    "BirthComp"
```

to make sure we have the correct spelling of the appropriate columns we want to keep.

```
birth2 <- birth_dat[ , c('Gender','weight','Habit')]
```

Interpretation: Keep all the rows, but only keep the columns: Gender, weight, and Habit

```
head(birth2,3)
```

```
#>   Gender weight     Habit
#> 1   Male    116 NonSmoker
#> 2   Male    126    Smoker
#> 3   Male    161 NonSmoker
```

We created a character vector with the names of the columns we wanted to keep and used it as the condition in the columns argument.

2. Split `birth_dat` into two parts: One for which the individual was a smoker and another for which they were not a smoker

The variable `Habit` contains information on whether or not the individual was a smoker.

```
unique(birth_dat$Habit)
```

```
#> [1] "NonSmoker" "Smoker"    ""
```

First we create a logical vector to determine if the individual was a smoker

```r
is_smoker <- birth_dat$Habit == 'Smoker'
```

```r
is_smoker[1:5]
```

```
#> [1] FALSE  TRUE FALSE FALSE FALSE
```

Interpretation: Return TRUE if Habit is smoker, otherwise FALSE

We use the negation logical operator to obtain all the non-smokers from our logical vector `is_smoker` without having to create a new variable

```r
!is_smoker[1:5]
```

```
#> [1]  TRUE FALSE  TRUE  TRUE  TRUE
```

To subset our data into keeping only the smokers we input our logical vector `is_smoker` into the rows argument

```r
smokers <- birth_dat[is_smoker, ]
```

Interpretation: Only keep the rows in which the individual is a smoker

```r
head(smokers,3)
```

```
#>     X Gender Premie weight Apgar1 Fage Mage Feduc Meduc TotPreg Visits
#> 2   2   Male     No    126      8   30   18    12    12       1     14
#> 16 16 Female    Yes     78      8   35   26    14    15       2      9
#> 19 19   Male     No    121      9   25   24    10    10       4     11
#>      Marital Racemom Racedad Hispmom Hispdad Gained  Habit MomPriorCond
#> 2  Unmarried   White Unknown NotHisp Unknown     50 Smoker At Least One
#> 16   Married   White   White NotHisp NotHisp     25 Smoker         None
#> 19 Unmarried   White   White NotHisp NotHisp     50 Smoker         None
#>    BirthDef    DelivComp BirthComp
#> 2      None         None      None
#> 16     None At Least One      None
#> 19     None         None      None
```

To subset our data into keeping only the non-smokers we input our logical vector `!is_smoker` into the rows argument

```r
not_smokers <- birth_dat[!is_smoker, ]
```

Interpretation: Only keep the rows in which the individual is NOT a smoker

```r
head(not_smokers,3)
```

```
#>   X Gender Premie weight Apgar1 Fage Mage Feduc Meduc TotPreg Visits Marital
#> 1 1   Male     No    116      9   28   34     6     3       2     10 Married
#> 3 3   Male     No    161      8   28   29    12    12       3     14 Married
#> 4 4   Male     No    133      9   26   23     8     9       3     10 Married
#>   Racemom Racedad   Hispmom   Hispdad Gained     Habit MomPriorCond BirthDef
#> 1   White   White   Mexican   Mexican     30 NonSmoker         None     None
#> 3   White   White OtherHisp OtherHisp     65 NonSmoker         None     None
#> 4   White   White   Mexican   Mexican      8 NonSmoker         None     None
#>      DelivComp BirthComp
#> 1         None      None
#> 3 At Least One      None
#> 4 At Least One      None
```

3. What is the average weight of babies with at least one birth defect?

The variable `BirthDef` determines if the baby had no birth defects or had at least one defect

```r
unique(birth_dat$BirthDef)
```

```
#> [1] "None"         "At Least One"
```

Create a logical vector to determine if the baby had at least one defect

```r
has_defect <- (birth_dat$BirthDef == 'At Least One')
```

> **Note**
>
> We must spell "At Least One" with correct upper/lower cases including spaces

```r
has_defect[1:5]
```

```
#> [1] FALSE FALSE FALSE FALSE FALSE
```

Subset our data to include rows with babies with at least one defect, then select only the `weight` column. Lastly compute the mean.

```
mean( birth_dat[has_defect,'weight'] )
```

```
#> [1] 115.8
```

Interpretation: Average weight of babies with at least one birth defect

Missing Data

Missing data (or missing values) appear when no value is available in one or more variables of an observation. A common example can look something like this

StudentID	Major	GPA
12345	math	3.8
23456	NA	3.2
23405	biology	NA

where we do not know the major of the second student and we also do not know the major from the third student *(denoted by `NA`)*

Identifying the rows and columns where missing values occur is necessary before addressing the issue of missingness. Although it is easy to observe in the example mentioned above, in most cases, dealing with larger datasets requires a more programmatic approach

Vectors

In R, `NA` stands for "Not Available" and is used to represent missing values in a dataset. `NA` can be used for any data type in R, such as numeric character, or logical.

The type of `NA` is a logical value

```
typeof(NA)
```

```
#> [1] "logical"
```

and can be coerced into any other data type. For example, consider the following numeric vector

```
typeof(c(1,2,3))
```

```
#> [1] "double"
```

but now with a missing value as the third element, it will preserve the original data type

```
typeof(c(1,2,NA,4))
```

```
#> [1] "double"
```

or even a character vector

```
typeof(c("a","b",NA,"c"))
```

```
#> [1] "character"
```

In the following, we will show several examples how to find missing values. The most common approach is to use the function `is.na()`

```
is.na(c(1,2,NA,4,NA))
```

```
#> [1] FALSE FALSE  TRUE FALSE  TRUE
```

Interpretation: For each element does this element contain `NA`

which will return a logical vector of the same length as the input vector, `TRUE` in the position which `NA` is located in. We can use the function `which()` in order to find out the actual position of `TRUE`

```
which( is.na(c(1,2,NA,4,NA)) )
```

```
#> [1] 3 5
```

Interpretation: Which position(s) are the logical values `TRUE` located

The output will then be an integer vector denoting the positions in which there were missing values. Applying the concepts learned in **Subsetting**, we can exclude any values which are missing. For example,

```
x <- c(1,2,NA,4,NA)
is.na(x)
```

```
#> [1] FALSE FALSE  TRUE FALSE  TRUE
```

```
!is.na(x)
```

```
#> [1]  TRUE  TRUE FALSE  TRUE FALSE
```

Interpretation: For each element does this element NOT contain `NA`

```
x[!is.na(x)]
```

```
#> [1] 1 2 4
```

Interpretation: Only keep the elements which DO NOT contain `NA`

If we only want to find out if there any `NA` values, we can utilize the function `anyNA()`

```
anyNA(c(1,2,NA,4,NA))
```

```
#> [1] TRUE
```

The above command will output `TRUE` if there are any `NA` in the vector and `FALSE` if there is not a single missing value

```
anyNA(c(1,2,3))
```

```
#> [1] FALSE
```

In conclusion, a common approach to check for missing data in R, we can use `is.na()` or `anyNA()`. If we want to know the position of the missing values, we should use `is.na()`. However, if we are only concerned with whether there are any missing values or not, and not their position, then we can use `anyNA()`

Data Frames

Now, working with data frames we would like to verify if there are any missing observations throughout the entire dataset

```r
student_dat <- data.frame(
  'StudentID' = c('12345', '23456', '23405'),
  'Major' = c("math",NA,"biology"),
  'GPA' = c(3.8,3.2,NA))
```

```
#>   StudentID   Major GPA
#> 1     12345    math 3.8
#> 2     23456    <NA> 3.2
#> 3     23405 biology  NA
```

When the function `is.na()` is applied to a data frame, the output will be a matrix containing logical values. The logical values in the matrix will depend on whether there were any missing values or not in the data frame

```r
is.na(student_dat)
```

```
#>      StudentID Major   GPA
#> [1,]     FALSE FALSE FALSE
#> [2,]     FALSE  TRUE FALSE
#> [3,]     FALSE FALSE  TRUE
```

If we wanted to find out the position(s) of the missing values for each column we will utilize the `apply()`. The basic syntax for `apply()` is

`apply(X, MARGIN, FUN)`

- `x`: an array or matrix
- `MARGIN`: take a value or range between 1 and 2 to define where to apply the function
- `MARGIN=1`: the manipulation is performed on rows
- `MARGIN=2`: the manipulation is performed on columns
- `MARGIN=c(1,2)`: the manipulation is performed on rows and columns
- `FUN`: tells which function to apply, according to the specified `MARGIN`

```r
apply(X = is.na(student_dat), MARGIN = 2, FUN = which)
```

```
#> $StudentID
#> integer(0)
#>
#> $Major
#> [1] 2
#>
#> $GPA
#> [1] 3
```

Interpretation: From each column `MARGIN =2`, which values `(FUN = which)` from student_dat are missing `is.na(student_dat)`

The output of using `apply(...,MARGIN =2)` will be a list containing the row(s) in which missing values were found from each `column`.

In our case there were no rows with missing data in the first column, the second row contained a missing value from the column *Major* and the third row contained a missing value from the column *GPA*