Logical Expressions

Chapter 3

Stats 20: Introduction to Statistical Programming with R

UCLA

Contents

Acknowledgements: Vivian Lew and Juana Sanchez

Learning Objectives

After studying this chapter, you should be able to:

- Use relational operators to compare values and create logical vectors.
- Use logical vectors as vector indices for subsetting and assignment.
- Use Boolean operators to combine logical expressions.

1 Relational Operators

1.1 Definitions

It is often necessary in programming to test relations between values. A **relational operator** is a binary operator that compares values in vectors. Applying a relational operator will produce a logical expression, which outputs `TRUE` if the expression is true and `FALSE` if the expression is false. Many logical expressions in R use these relational operators.

A list of the relational operators in R is below:

- `<` : Less than
- `>` : Greater than
- `<=` : Less than or equal to
- `>=` : Greater than or equal to
- `==` : Equal to
- `!=` : Not equal to

Caution: Notice that the logical comparison `==` uses a *double* equal sign, not a single equal sign. The single equal sign `=` is reserved for object assignment and setting default function arguments.

Some examples:

```
3 < 4 # Is 3 less than 4?
```

```
[1] TRUE
```

```
7 > 7 # Is 7 (strictly) greater than 7?
```

```
[1] FALSE
```

```
7 <= 7 # Is 7 less than or equal to 7?
```

```
[1] TRUE
```

```
3 - 4 >= 0 # Is 3 - 4 greater than or equal to 0?
```

```
[1] FALSE
```

```
TRUE == FALSE # Is TRUE equal to FALSE?
```

```
[1] FALSE
```

```
9 * 3 != 2 # Is 9 * 3 not equal to 2?
```

```
[1] TRUE
```

Note: When comparing logical values, `TRUE` is coerced into `1` and `FALSE` is coerced into `0`.

Caution: Be careful with tests of equality (`==` and `!=`). Because R uses finite (double) precision when storing numbers, rounding errors may produce unexpected results. A test of equality will only output `TRUE` if the two values are represented *exactly* in R.

To illustrate this, consider this example:

```
49 * (4 / 49) == 4 # Is 49 * (4 / 49) exactly equal to 4?
```

```
[1] FALSE
```

To help get around rounding errors when comparing values, it can be helpful to use the **`round()`** function, which rounds a numeric input. The second argument is called `digits`, which specifies the number of digits (decimal places) to use.

```
pi # Outputs pi to 6 digits
```

```
[1] 3.141593
```

```
pi == 3.141593 # Is pi equal to 3.141593?
```

```
[1] FALSE
```

```
round(pi, digits = 6) # Round pi to 5 digits
```

```
[1] 3.141593
```

```
round(pi, 6) == 3.141593 # Is rounded pi equal to 3.14159?
```

```
[1] TRUE
```

```
round(49 * (4 / 49), 8) == 4 # Is 49 * (4 / 49) (rounded to 8 digits) equal to 4?
```

```
[1] TRUE
```

1.2 Vectorization

Relational operators are vectorized, meaning that the relational operator will be applied to each element of a vector individually.

```
# Is 3 or 8 greater than or equal to 3?
c(3, 8) >= 3
```

```
[1] TRUE TRUE
```

```
# Is 1, 4, or 9 exactly equal to 9?
c(1, 4, 9) == 9
```

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

```
# Is 1, 4, or 9 not (exactly) equal to 9?
c(1, 4, 9) != 9
```

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

When comparing vectors, the corresponding elements of each vector are compared.

```
# Is 3 less than 1, and is 8 less than 4?
c(3, 8) < c(1, 4)
```

```
[1] FALSE FALSE
```

```r
# Is 1 greater than 5, is 4 greater than 6, and is 9 greater than 7?
c(1, 4, 9) > c(5, 6, 7)
```

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

1.3 Recycling

Relational operators also recycle vectors in the same way that arithmetic operators do.

```r
c(1, 4) == c(5, 3, 7, 4)
```

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

```r
c(1, 4, 9, 3) >= c(5, 4)
```

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

```r
c(1, 4, 9, 3, 8) > c(5, 6, 7)
```

```
Warning in c(1, 4, 9, 3, 8) > c(5, 6, 7): longer object length is not a multiple
of shorter object length
```

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

Question: What values are being compared in these examples?

Notice that the length of the logical output vector has the same length as the longer vector in the relational statement.

1.4 The `any()`, `all()`, and `identical()` Functions

The relational operators compare vectors elementwise using vectorization, so the result from an input vector is a logical output vector. If we want to know if any or all of the individual logical values are `TRUE`, we can use the `any()` and `all()` functions.

The **`any()`** function inputs a logical vector and outputs `TRUE` if any of the values is `TRUE`. The **`all()`** function inputs a logical vector and outputs `TRUE` if *all* of the values are `TRUE`.

```r
# Create a vector of the integers from 1 to 10
first_ten <- 1:10
first_ten
```

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

```r
# Are any of the values greater than 7?
any(first_ten > 7)
```

```
[1] TRUE
```

```r
# Are any of the values greater than 70?
any(first_ten > 70)
```

```
[1] FALSE
```

```r
# Are all of the values greater than 70?
all(first_ten > 70)
```

```
[1] FALSE
```

```r
# Are all of the values greater than 0?
all(first_ten > 0)
```

```
[1] TRUE
```

The `all()` function can be used to test whether two vectors are equal. For example:

```
# Create a vector of the integers from 1 to 5 using c()
first_five <- c(1, 2, 3, 4, 5)
# Create a vector of the integers from 1 to 5 using :
seq_five <- 1:5
# Are all the elements of first_five == seq_five TRUE?
all(first_five == seq_five)
```

```
[1] TRUE
```

Another test for vector equality is to use the `identical()`, which inputs any two R objects and outputs `TRUE` if they are *exactly* identical. For example:

```
identical(seq(1, 10), 1:10)
```

```
[1] TRUE
```

This shows that the `seq()` function and the colon operator `:` do generate identical sequences.

Side Note: Use caution with `identical()`. The `identical()` function tests whether two R objects are exactly identical objects in R, down to their storage mode and all their attributes. Notice the following example:

```
first_five
```

```
[1] 1 2 3 4 5
```

```
seq_five
```

```
[1] 1 2 3 4 5
```

```
identical(first_five, seq_five)
```

```
[1] FALSE
```

The issue lies in the fact that the integers `1, 2, 3, 4, 5` used in generating `first_five` are actually stored as floating point numbers (doubles), while `seq()` and `:` generate integers.

```
typeof(first_five)
```

```
[1] "double"
```

```
typeof(seq_five)
```

```
[1] "integer"
```

This technical point seldom comes up in practice, but it is important to be aware of in case it causes an unexpected issue.

Side Note: Add an `L` to the end of an integer to explicitly refer to the integer-stored version of the number For example, the number `4` is stored as a double, but `4L` is stored as an integer.

1.5 Special Values

1.5.1 `NA`, `NULL`, and `NaN`

The special values `NA`, `NULL`, and `NaN` are incomparable using relational operators.

```
c(7, NA, 4) > 6
```

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

```r
c(TRUE, FALSE) > NULL
```

```
logical(0)
```

```r
c(1, 4, 9) <= NaN
```

```
[1] NA NA NA
```

Since comparing special values is not possible, we instead are often interested in whether elements in a vector are special values or not. This is particularly useful in finding where entries in a dataset are missing values. The **`is.na()`**, **`is.nan()`**, and **`is.null()`** functions are used in this context.

- The `is.na()` function inputs an object and outputs `TRUE` if the corresponding elements that are `NA` or `NaN`.
- The `is.nan()` function inputs an object and outputs `TRUE` if the corresponding elements that are `NaN`.
- The `is.null()` function inputs an object and outputs `TRUE` if the object is the `NULL` object.

```r
is.na(c(7, NA, 4, NA, 3, Inf, NaN))
```

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

```r
is.nan(c(1, NaN, -4, Inf, NA))
```

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

```r
is.null(NULL)
```

```
[1] TRUE
```

```r
is.null(c(1, NULL, NA))
```

```
[1] FALSE
```

Note: Notice that `is.na()` and `is.nan()` are vectorized functions, but `is.null()` is not.

1.5.2 Inf

Since infinity represents a value that is larger than any finite number, `Inf` can be compared against numbers in an expected and intuitive way.

```r
c(1, 4, Inf) < Inf
```

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

```r
-Inf < exp(100)
```

```
[1] TRUE
```

```r
exp(1000) == Inf
```

```
[1] TRUE
```

Note: Even though `exp(1000)` is not technically infinite, `exp(1000) == Inf` outputs `TRUE`. Because R cannot represent extremely large values, the object `exp(1000)` itself outputs `Inf`, so R interprets `exp(1000)` as being the same object as `Inf`.

The vectorized functions **`is.infinite()`** and **`is.finite()`** can be used to test whether elements are infinite or finite.

```r
is.infinite(c(7, -Inf, 8, NA, NaN, Inf))
```

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

```r
is.finite(c(7, -Inf, 8, NA, NaN, Inf))
```

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

2 Logical Indexing

2.1 Subsetting

Logical vectors can also be used as indices for subsetting. Using a logical index will extract every entry that corresponds to a `TRUE` value in the index vector.

A common use of logical indices is to extract only certain elements that satisfy some condition or criterion. As an example, we will return to Chris Traeger's running times from the previous chapter.

```r
# Input the data into R
running_times <- c(51, 40, 57, 34, 47, 50, 50, 56, 41, 38)
running_times
```

```
 [1] 51 40 57 34 47 50 50 56 41 38
```

```r
# Is the running time longer than 40 (minutes)?
running_times > 40
```

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

```r
# Extract only the running times that are longer than 40 (minutes)
running_times[running_times > 40]
```

```
[1] 51 57 47 50 50 56 41
```

Question: How would we extract only Chris Traeger's running times that were at most 35 minutes?

Another example:

```r
# Create a logical index
logical_index <- (running_times %% 2) == 0
logical_index
```

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

```r
# Extract the values corresponding to the TRUE values in logical_index
running_times[logical_index]
```

```
[1] 40 34 50 50 56 38
```

Question: What kind of numbers in `running_times` will output `TRUE` from the `logical_index` command `(running_times %% 2) == 0`?

If the logical index vector is shorter than the given vector, the logical index will be recycled.

```r
# Create a logical index of length 2
every_other <- c(TRUE, FALSE)
# Extract every other element from running_times
running_times[every_other]
```

```
[1] 51 57 47 50 41
```

2.2 The which() Function

Logical indices give us a way to extract elements of a vector that satisfies a certain criterion. In some scenarios, we may just want to find the positions within the vector at which the condition is satisfied.

The `which()` function inputs a logical vector and outputs a numeric vector of the *indices* (or positions) of the `TRUE` values. That is, `which()` outputs which elements of the input vector are `TRUE`.

```
# Is the running time at least 50 (minutes)?
running_times >= 50
```

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

```
# Which running times are at least 50 (minutes)?
which(running_times >= 50)
```

```
[1] 1 3 6 7 8
```

Question: What is the output of `which(is.na(c(7, NA, 4, NA, 3, Inf, NaN)))`?

We can also use the output of the `which()` function as an index for subsetting.

```
# Extract the running times that are at least 50 (minutes)
running_times[which(running_times >= 50)]
```

```
[1] 51 57 50 50 56
```

Note: In most cases, the `which()` function is unnecessary and redundant when used for subsetting. It is clearer and simpler to use the logical index itself instead of converting to a numeric index.

```
running_times[running_times >= 50]
```

```
[1] 51 57 50 50 56
```

3 Boolean Operators (Combining Logical Expressions)

So far, we have used each relational operator separately to create single logical expressions. Sometimes we want to combine two logical expressions to make a compound logical expression. The mathematical formalization for combining logical expressions is called **Boolean algebra**.

The three main **Boolean operators** are **&** (and), **|** (or), and **!** (not). All three of these operators are vectorized.

For example, suppose we have a vector of whole numbers (integers), in no particular order:

```
some_nums <- c(4, 1, 2, 6, 8, 5, 3, 7)
some_nums
```

```
[1] 4 1 2 6 8 5 3 7
```

Which entries in `some_nums` are greater than 3?

```
some_nums > 3 # Which entries are greater than 3?
```

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

Which entries in `some_nums` are less than 7?

```
some_nums < 7 # Which entries are less than 7?
```

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

3.1 The `&` (and) Operator

Suppose we want to know which entries of `some_nums` are *both* greater than 3 AND less than 7. To do this in one line in R, we can use the `&` (**and**) operator that compares two (or more) logical expressions of the same length and outputs a logical vector that is `TRUE` if both expressions are simultaneously `TRUE` and `FALSE` otherwise (both are `FALSE` or only one is `TRUE`).

```
some_nums > 3 & some_nums < 7 # Which entries are greater than 3 AND less than 7?
```

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

3.2 The `|` (or) Operator

Suppose we want to know which entries of `some_nums` are less than 3 OR greater than 7 (or both). This is an **inclusive or**, which means only one of the condtions needs to be satisfied for the statement to be true. To do this in one line in R, we can use the `|` (**or**) operator that compares two (or more) logical expressions of the same length and outputs a logical vector that is `TRUE` if *at least one* expression is `TRUE` and `FALSE` only if both are `FALSE`.

```
some_nums < 3 # Which entries are less than 3?
```

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

```
some_nums > 7 # Which entries are greater than 7?
```

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

```
some_nums < 3 | some_nums > 7 # Which entries are less than 3 OR greater than 7?
```

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

Question: Use either `&` or `|` to find which entries of `some_nums` are greater than 4 AND less than 6.

Question: Use either `&` or `|` to find which entries of `some_nums` are less than 4 OR greater than 6.

3.3 The `!` (not) Operator

The exclamation point `!` is the **not** or **logical negation** operator that outputs `TRUE` if the input is `FALSE` and `FALSE` if the input is `TRUE`. The exclamation operation is placed at the beginning of a logical statement or vector. Parentheses help clarify which expressions are being negated.

```
!FALSE # Not FALSE
```

```
[1] TRUE
```

```
!(some_nums < 3) # Which entries are NOT less than 3?
```

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

```
!(some_nums < 3 | some_nums > 7) # Which entries are NOT (less than 3 OR greater than 7)?
```

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

```
!(some_nums < 3) | some_nums > 7 # Which entries are (NOT less than 3) OR greater than 7?
```

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

```
!(some_nums > 3 & some_nums < 7) # Which entries are NOT (greater than 3 AND less than 7)?
```

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

3.4 The `&&` and `||` Operators

The `&&` and `||` operators are similar to their respective `&` and `|` counterparts but with two key differences.

- The `&&` and `||` operators are *not* vectorized. The operators expect length 1 logical vector (i.e., single logical value) inputs. If vectors of length greater than 1 are used, they will throw a warning and only compare the first element of each vector.

  Note: In earlier versions of R (prior to R Version 4.2.0), the `&&` and `||` operators did not throw warnings when using vectors of length greater than 1.

- The `&&` and `||` operators use **short-circuit evaluation**: They will evaluate expressions from left to right and *only* evaluate the right expression if necessary. For example, if `A` is `TRUE`, then `A || B` will be `TRUE` regardless of the value of `B`, so `B` will not need to be evaluated for `A || B` to output `TRUE`. Short-circuit evaluation is particularly beneficial when `B` is complicated, slow to evaluate, or evaluating `B` will throw an error or a warning when `A` is `TRUE`.

```
c(TRUE, FALSE, TRUE) && c(FALSE, TRUE, TRUE)
```

```
Warning in c(TRUE, FALSE, TRUE) && c(FALSE, TRUE, TRUE): 'length(x) = 3 > 1' in
coercion to 'logical(1)'

Warning in c(TRUE, FALSE, TRUE) && c(FALSE, TRUE, TRUE): 'length(x) = 3 > 1' in
coercion to 'logical(1)'
```

```
[1] FALSE
```

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

```
Warning in c(TRUE, FALSE, TRUE) || c(FALSE, TRUE, TRUE): 'length(x) = 3 > 1' in
coercion to 'logical(1)'
```

```
[1] TRUE
```

```
x <- -5
x < 0 | is.na(sqrt(x) > 2)
```

```
Warning in sqrt(x): NaNs produced
```

```
[1] TRUE
```

```
x < 0 || is.na(sqrt(x) > 2)
```

```
[1] TRUE
```

These non-vectorized operators are generally preferred/recommended for flow control statements, such as in `if()` statements or `while()` loops, where single logical values are expected.

Question: Why does the first command above throw two warnings but the second command only throws one?

3.5 Boolean Algebra and Set Theory

There is an important relationship between Boolean algebra and set theory. In statistics, we often represent events or collections of outcomes from random scenarios using set theory notation. The Boolean operators in logic correspond to the set theoretic operations we use to combine sets (or events).

Suppose we interpret logical statements A and B as sets. The set "A and B" consists of elements which are both in A and in B. In set theory, A and B is called the **intersection** of A and B, denoted by $A \cap B$. Similarly, "A or B" is the set of elements that are in A or in B (or both), which is also called the **union** of A and B, denoted by $A \cup B$. The set "not A" is called the **complement** of A, denoted by A^c.

Since logical statements can only have two values (true and false), we can put all of the results from Boolean operations in a table, often called a **truth table**, given below.

Boolean	A	B	not A	not B	A and B	A or B
R	`A`	`B`	`!A`	`!B`	`A & B`	`A | B`
	`TRUE`	`TRUE`	`FALSE`	`FALSE`	`TRUE`	`TRUE`
	`TRUE`	`FALSE`	`FALSE`	`TRUE`	`FALSE`	`TRUE`
	`FALSE`	`TRUE`	`TRUE`	`FALSE`	`FALSE`	`TRUE`
	`FALSE`	`FALSE`	`TRUE`	`TRUE`	`FALSE`	`FALSE`

Table 1: Truth table for Boolean operators