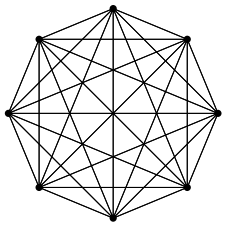
Chapter 13

Relational Operators



The relational operators $>$, $\geq$, $<$, $\leq$, $=$, and $\neq$ can be used to compare the elements of an object. These relational operators return either the logical value `TRUE` or the logical value `FALSE`. The relational operators considered here are implemented in R as

```
>          >=          <          <=          ==          !=
```

The double equals == is used for testing equality because the single equals = has already been used as an assignment operator and to identify named arguments in function calls. The relational operators are vectorized so that they can be applied element-wise to the objects under consideration. The three topics considered in this chapter are (a) relational operators applied to objects, (b) conditional execution via `ifelse`, and (c) conditional execution via `switch`.

13.1 Relational operators applied to objects

Although the illustrations used in this chapter are applied to vectors, they are easily adapted for more complicated data structures, such as matrices or arrays. To illustrate the use of relational operators, begin by defining a vector `x` that contains four numeric elements:

```
> x = c(2, -3, 0, 8)       # a vector of numeric elements
> x                        # display x
[1]  2 -3  0  8
```

Next, to check whether any of the elements of `x` assume the value `7`, type

```
> x == 7                   # which elements in x, if any, equal 7?
[1] FALSE FALSE FALSE FALSE
```

The value 7 is effectively recycled, and the equality operator == returns a vector of four `FALSE` elements because *none* of the elements of the vector `x` equal 7. In the same fashion, to check whether any elements in `x` are equal to `8`, type

```
> x == 8                   # which elements in x equal 8?
[1] FALSE FALSE FALSE  TRUE
```

The last element in `x` is `8`, so a `TRUE` is returned in that position.

The discussion concerning floating point representations from Chapter 9 also applies to relational operators. Consider the R commands

```
> sqrt(3) * sqrt(3)        # the square root of 3 squared is 3
[1] 3
> sqrt(3) * sqrt(3) == 3   # test for equality
[1] FALSE
```

What happened? The square root of 3, which is irrational, does not have an exact binary representation, so although $\sqrt{3} \cdot \sqrt{3}$ *rounds* to 3, it might not be *equal* to 3 in its binary representation. This was the case in the computation above, which can be confirmed with the R commands

```
> options(digits = 20)     # display 20 digits
> sqrt(3) * sqrt(3)        # the square root of 3 squared is approximately 3
[1] 2.9999999999999995559
```

The alternative `all.equal` function includes a tolerance argument (with a default which can be adjusted) when testing for equality.

```
> all.equal(sqrt(3) * sqrt(3), 3)
[1] TRUE
```

This returns the intuitive result. There is another alternative to using the == relational operator. The `identical` function returns `TRUE` if the internal representations of its first two arguments are identical and `FALSE` otherwise. Consider comparing 7 stored as the integer `7L` and 7 stored as a `numeric`.

```
> 7L == 7                  # compare integer 7 and numeric 7
[1] TRUE
> identical(7L, 7)         # compare integer 7 and numeric 7
[1] FALSE
```

The == relational operator does not check the internal representations of `7L` and `7`, but the `identical` function does. The key take-away here is to be very careful with the == relational operator because of the limitations associated with computer arithmetic.

The *not equal* relational operator is an exclamation point followed immediately by =, as illustrated by

```
> x                        # display x
[1]  2 -3  0  8
> x != 0                   # which elements in x are nonzero?
[1]  TRUE  TRUE FALSE  TRUE
```

The first, second, and fourth elements of `x` are nonzero, resulting in the vector of logical values displayed. To determine which elements of the vector `x` are negative, type

```
> x < 0                    # which elements in x are negative?
[1] FALSE  TRUE FALSE FALSE
```

Only the second element is negative. The logical vectors that are generated by the relational operators can be manipulated in the same manner as any other vectors. For example, using the `c` function

```
> c(x == 0, x != 0)        # vector of eight logical elements
[1] FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE
```

concatenates the two four-element vectors of logical elements together to form an eight-element vector of logical elements. The next R command locates all of the positions of the negative elements in the vector `x` and replaces their values with 17.

```
> x[x < 0] = 17            # set all negative elements in x to 17
> x                        # display x
[1]  2 17  0  8
```

In this case, the logical values are used as subscripts as they were in the previous chapter.

Although all of the examples in this chapter use relational operators to compare numeric values, they can also be applied to strings. To compare the strings `"C"` and `"R"`, for example, type

```
> "C" < "R"                # is "C" less than "R"?
[1] TRUE
```

The internal representations of these two strings are compared, and the value for `"C"` is less than the value for `"R"`, so `TRUE` is returned. Longer strings are compared in a lexicographic manner (a lexicographic ordering is a generalization of alphabetical ordering to include numbers and symbols), for example,

```
> "C3PO" > "R2D2"          # is "C3PO" greater than "R2D2"?
[1] FALSE
```

Relational operators can also be applied to two objects in a component-wise fashion. Begin by assigning the values 1, 4, 2, and 5 to the four-element numeric vector `y`.

```
> y = c(1, 4, 2, 5)        # another vector of numeric elements
> y                        # display y
[1] 1 4 2 5
```

The vectors `x` (which currently contains the elements 2, 17, 0, 8) and `y` can be compared in an element-wise fashion with

```
> x > y                    # which elements of x are greater than y
[1]  TRUE  TRUE FALSE  TRUE
```

In this case, the vectors `x` and `y` were of the same length. If this were not the case, then R would recycle the values in the shorter vector in the usual fashion. In a minor extension of this example, you can display the elements of `x` in which the corresponding element of `x` is greater than the associated element of `y` with

```
> x[x > y]                 # elements of x in which x is greater than y
[1]  2 17  8
```

If you would like to take those elements associated with `x > y` and multiply each of the associated elements of the `x` vector by 4, type

```
> x[x > y] = x[x > y] * 4  # multiply elements of x in which x > y by four
> x                        # display x
[1]  8 68  0 32
```

The `which` function is useful for identifying the index values that satisfy a particular logical relationship. The `which` function accepts a vector of logical elements as an argument and returns the *indexes* of `TRUE` values in the vector, not the values of the associated elements themselves. The R command

```
> which(x < 10)             # indexes of elements in x that are less than 10
[1] 1 3
```

indicates that the first and third elements of `x` (`x` currently contains the numeric values 8, 68, 0, 32) are the elements that are less than 10. Notice that the `x < 10` portion of this R command creates a vector of four logical elements, and the `which` function then returns the index or indexes of the `TRUE` elements in the vector. As a second example, `which` determines the indexes of those values of `x` that are less than the sample mean:

```
> which(x < mean(x))       # indexes of elements in x that are less than xbar
[1] 1 3
```

The first and third elements of `x` are less than the sample mean.

Finally, the following R commands assign the four-element vectors `name`, `age`, and `va` to elements that are character strings, numeric values, and logical values, respectively, then finds the names of all people under the age of 21 who live in Virginia.

```
> name = c("Joe", "Ian", "Liz", "Ali")  # vector of names
> age  = c(19, 21, 18, 20)              # vector of ages
> va   = c(TRUE, FALSE, TRUE, FALSE)    # vector of Va. residency statuses
> name[age < 21 & va]                   # names of young Virginians
[1] "Joe" "Liz"
```

This example combines operators on logical elements from last chapter and relational operators from this chapter with subscripting to produce efficient, vectorized calculations. In the last command in this example, R had to choose between the relational operator < and the vectorized *and* operator & in terms of precedence. R chose to execute the < before the &. The table below gives the precedence rules for the operators that we have encountered thus far, with the operator with highest precedence ranked first. Exceptions to these precedences can be accomplished by adding parentheses. A more complete list of the operator preferences is available in an R session by typing

```
> help(Syntax)              # list of operator precedences
```

operator	description
[ ]	subscripting
^	exponentiation operator
-	unary minus
:	vector creation
%/%, %%	integer-divide and modulo operators
*, /	multiplication and division operators
+, -	addition and subtraction operators
>, >=, <, <=, ==, !=	relational operators
!	*not* operator
&, &&, |, ||	*and* and *or* operators
=, <-	assignment operators

Now that you are familiar with the use of relational operations, the versatile `ifelse` function can be introduced.