3.9: Storing “True or False” Data

Time to move onto a third kind of data. A key concept in that a lot of R relies on is the idea of a ***logical value***. A logical value is an assertion about whether something is true or false. This is implemented in R in a pretty straightforward way. There are two logical values, namely `TRUE` and `FALSE`. Despite the simplicity, a logical values are very useful things. Let’s see how they work.

3.9.1 Assessing mathematical truths

In George Orwell’s classic book *1984*, one of the slogans used by the totalitarian Party was “two plus two equals five”, the idea being that the political domination of human freedom becomes complete when it is possible to subvert even the most basic of truths. It’s a terrifying thought, especially when the protagonist Winston Smith finally breaks down under torture and agrees to the proposition. “Man is infinitely malleable”, the book says. I’m pretty sure that this isn’t true of humans[36] but it’s definitely not true of R. R is not infinitely malleable. It has rather firm opinions on the topic of what is and isn’t true, at least as regards basic mathematics. If I ask it to calculate `2 + 2`, it always gives the same answer, and it’s not bloody 5:

```
2 + 2
```

```
## [1] 4
```

Of course, so far R is just doing the calculations. I haven’t asked it to explicitly assert that 2+2=4 is a true statement. If I want R to make an explicit judgement, I can use a command like this:

```
2 + 2 == 4
```

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

What I’ve done here is use the ***equality operator***, `==`, to force R to make a “true or false” judgement.[37] Okay, let’s see what R thinks of the Party slogan:

```
2+2 == 5
```

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

Booyah! Freedom and ponies for all! Or something like that. Anyway, it’s worth having a look at what happens if I try to *force* R to believe that two plus two is five by making an assignment statement like `2 + 2 = 5` or `2 + 2 <5`. When I do this, here’s what happens:

```
2 + 2 = 5
```

```
## Error in 2 + 2 = 5: target of assignment expands to non-language object
```

R doesn’t like this very much. It recognises that `2 + 2` is *not* a variable (that’s what the “non-language object” part is saying), and it won’t let you try to “reassign” it. While R is pretty flexible, and actually does let you do some quite remarkable things to redefine parts of R itself, there are just some basic, primitive truths that it refuses to give up. It won’t change the laws of addition, and it won’t change the definition of the number `2`.

That’s probably for the best.

3.9.2 Logical operations

So now we’ve seen logical operations at work, but so far we’ve only seen the simplest possible example. You probably won’t be surprised to discover that we can combine logical operations with other operations and functions in a more complicated way, like

this:

```
3*3 + 4*4 == 5*5
```

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

or this

```
sqrt( 25 ) == 5
```

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

Not only that, but as Table 3.2 illustrates, there are several other logical operators that you can use, corresponding to some basic mathematical concepts.

Table 3.2: Some logical operators. Technically I should be calling these "binary relational operators", but quite frankly I don't want to. It's my book so no-one can make me.

operation	operator	example input	answer
less than	<	2 < 3	TRUE
less than or equal to	<=	2 <= 2	TRUE
greater than	>	2 > 3	FALSE
greater than or equal to	>=	2 >= 2	TRUE
equal to	==	2 == 3	FALSE
not equal to	!=	2 != 3	TRUE

Hopefully these are all pretty self-explanatory: for example, the ***less than*** operator `<` checks to see if the number on the left is less than the number on the right. If it's less, then R returns an answer of `TRUE`:

```
99 < 100
```

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

but if the two numbers are equal, or if the one on the right is larger, then R returns an answer of `FALSE`, as the following two examples illustrate:

```
100 < 100
```

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

```
100 < 99
```

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

In contrast, the ***less than or equal to*** operator `<=` will do exactly what it says. It returns a value of `TRUE` if the number of the left hand side is less than or equal to the number on the right hand side. So if we repeat the previous two examples using `<=`,

here's what we get:

```
100 <= 100
```

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

```
100 <= 99
```

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

And at this point I hope it's pretty obvious what the ***greater than*** operator `>` and the ***greater than or equal to*** operator `>=` do! Next on the list of logical operators is the ***not equal to*** operator `!=` which – as with all the others – does what it says it does. It returns a value of `TRUE` when things on either side are not identical to each other. Therefore, since 2+2 isn't equal to 5, we get:

```
2 + 2 != 5
```

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

We're not quite done yet. There are three more logical operations that are worth knowing about, listed in Table 3.3.

Table 3.3: Some more logical operators.

operation	operator	example input	answer
not	!	!(1==1)	FALSE
or	|	(1==1) | (2==3)	TRUE
and	&	(1==1) & (2==3)	FALSE

These are the ***not*** operator `!`, the ***and*** operator `&`, and the ***or*** operator `|`. Like the other logical operators, their behaviour is more or less exactly what you'd expect given their names. For instance, if I ask you to assess the claim that "either 2+2=4 *or* 2+2=5" you'd say that it's true. Since it's an "either-or" statement, all we need is for one of the two parts to be true. That's what the `|` operator does:

```
(2+2 == 4) | (2+2 == 5)
```

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

On the other hand, if I ask you to assess the claim that "both 2+2=4 *and* 2+2=5" you'd say that it's false. Since this is an *and* statement we need both parts to be true. And that's what the `&` operator does:

```
(2+2 == 4) & (2+2 == 5)
```

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

Finally, there's the *not* operator, which is simple but annoying to describe in English. If I ask you to assess my claim that "it is not true that 2+2=5" then you would say that my claim is true; because my claim is that "2+2=5 is false". And I'm right. If we write this as an R command we get this:

```
! (2+2 == 5)
```

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

In other words, since `2+2 == 5` is a `FALSE` statement, it must be the case that `!(2+2 == 5)` is a `TRUE` one. Essentially, what we've really done is claim that "not false" is the same thing as "true". Obviously, this isn't really quite right in real life. But R lives in a much more black or white world: for R everything is either true or false. No shades of gray are allowed. We can actually see this much more explicitly, like this:

```
! FALSE
```

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

Of course, in our 2+2=5 example, we didn't really need to use "not" `!` and "equals to" `==` as two separate operators. We could have just used the "not equals to" operator `!=` like this:

```
2+2 != 5
```

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

But there are many situations where you really do need to use the `!` operator. We'll see some later on.[38]

3.9.3 Storing and using logical data

Up to this point, I've introduced *numeric data* (in Sections 3.4 and @ref(#vectors)) and *character data* (in Section 3.8). So you might not be surprised to discover that these `TRUE` and `FALSE` values that R has been producing are actually a third kind of data, called *logical data*. That is, when I asked R if `2 + 2 == 5` and it said `[1] FALSE` in reply, it was actually producing information that we can store in variables. For instance, I could create a variable called `is.the.Party.correct`, which would store R's opinion:

```
is.the.Party.correct <2 + 2 == 5
is.the.Party.correct
```

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

Alternatively, you can assign the value directly, by typing `TRUE` or `FALSE` in your command. Like this:

```
is.the.Party.correct <FALSE
is.the.Party.correct
```

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

Better yet, because it's kind of tedious to type `TRUE` or `FALSE` over and over again, R provides you with a shortcut: you can use `T` and `F` instead (but it's case sensitive: `t` and `f` won't work).[39] So this works:

```
is.the.Party.correct <F
is.the.Party.correct
```

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

but this doesn't:

```
is.the.Party.correct <f
```

```
## Error in eval(expr, envir, enclos): object 'f' not found
```

3.9.4 Vectors of logicals

The next thing to mention is that you can store vectors of logical values in exactly the same way that you can store vectors of numbers (Section 3.7) and vectors of text data (Section 3.8). Again, we can define them directly via the `c()` function, like this:

```
x <c(TRUE, TRUE, FALSE)
x
```

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

or you can produce a vector of logicals by applying a logical operator to a vector. This might not make a lot of sense to you, so let's unpack it slowly. First, let's suppose we have a vector of numbers (i.e., a "non-logical vector"). For instance, we could use the `sales.by.month` vector that we were using in Section@ref(#vectors). Suppose I wanted R to tell me, for each month of the year, whether I actually sold a book in that month. I can do that by typing this:

```
sales.by.month > 0
```

```
##  [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE
```

and again, I can store this in a vector if I want, as the example below illustrates:

```
any.sales.this.month <sales.by.month > 0
any.sales.this.month
```

```
##  [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE
```

In other words, `any.sales.this.month` is a logical vector whose elements are `TRUE` only if the corresponding element of `sales.by.month` is greater than zero. For instance, since I sold zero books in January, the first element is `FALSE` .

3.9.5 Applying logical operation to text

In a moment (Section 3.10) I'll show you why these logical operations and logical vectors are so handy, but before I do so I want to very briefly point out that you can apply them to text as well as to logical data. It's just that we need to be a bit more careful in understanding how R interprets the different operations. In this section I'll talk about how the equal to operator `==` applies to text, since this is the most important one. Obviously, the not equal to operator `!=` gives the exact opposite answers to `==` so I'm implicitly talking about that one too, but I won't give specific commands showing the use of `!=` . As for the other operators, I'll defer a more detailed discussion of this topic to Section 7.8.5.

Okay, let's see how it works. In one sense, it's very simple. For instance, I can ask R if the word `"cat"` is the same as the word `"dog"` , like this:

```
"cat" == "dog"
```

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

That's pretty obvious, and it's good to know that even R can figure that out. Similarly, R does recognise that a `"cat"` is a `"cat"` :

```
"cat" == "cat"
```

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

Again, that's exactly what we'd expect. However, what you need to keep in mind is that R is not at all tolerant when it comes to grammar and spacing. If two strings differ in any way whatsoever, R will say that they're not equal to each other, as the following examples indicate:

```
" cat" == "cat"
```

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

```
"cat" == "CAT"
```

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

```
"cat" == "c a t"
```

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