Chapter 2

Logic and R

2.1 Logic

The logic of modern computer science has roots in Aristotle's efforts. Aristotle's central observation with regard to the study of logic was that arguments were *valid or not* based on their logical structure, independent of the non-logical words involved. The most well-known argument form he described is known as the syllogism and looks like:

1. All men are mortal.
2. Socrates is a man.
3. Therefore, Socrates is mortal.

You can replace "Socrates" with any other noun, and "mortal" with any other noun, and the argument remains valid. The validity of the argument is determined solely by its *logical structure*. The logical words "all", "is", are", and "therefore" are doing all the work.

In programming, we often see what are called "if then" statements. An "if then" statement may look like

If p then q.
If x > 5 then print "stop".

Sometimes, however, the p may not be a single simple statement but may consist of a combination of many simple statements, and it may be hard to discern under which conditions q should occur. It is useful to study concepts in logic to break down complex statements so that they can be understood more easily. That is part of our goal in this chapter.

2.2 Statements

An example of a statement is

"The number computed is greater than 100".

In logic, statements are declarative sentences that are either true or false but not both. We will primarily focus on the "either true or false but not both" part of the above sentence.

The word *declarative* is used in contrast to sentences that are interrogative – for example an interrogative sentence (basically meaning a question) is "Do I have to go to school?"

Examples of declarative sentences are "Today is Monday", "The sun is shining", and "3 + 4 = 7".

Each of the following sentences is either true or false but not both: 1 is true, 2 is true, 3 is true, and 4 is false.

1. All men are mortal.
2. Socrates is a man.
3. Socrates is mortal.
4. 10 > 12.

"Are you going home?" in addition to *not* being a declarative sentence, cannot be considered "true or false but not both", and therefore this is an additional reason it is not a statement.[1] In our exploration of mathematical and computer logic, we restrict ourselves to statements. In the rest of this text, we will often use letters like p, q, r, s, or t to represent *statements*. These are used in a way similar to the way we often use x, y, or z to represent numbers. Thus, we may say that p represents the statement "Socrates is a man" and q represents the statement "10 > 12".

Practice 2.2.1

1. Is the sentence "What is your name?" a statement? Why or why not?
2. Is the sentence "New York is the largest city in China." a statement? Why or why not?
3. Is "The mean of {1, 2, 3, 4, 5} is 700" a statement?

2.3 Boolean Data Type

In many programming languages, common datatypes are integers, decimal numbers, arrays, and lists. There is not a function in R called **`datatype`** which we could use to test the data type of an object, although there are various functions that serve a similar purpose. The function **`is`**`()` partially serves this purpose as does the function **`class`**`()`.

In many programming languages defining something as an integer as opposed to a decimal (or double) will have unexpected consequences. For example, in Python, if you set x = 2, Python will assume x is an integer. If you set y = 2.0 it will assume it is a decimal number. If you write x/3 you will get 1 because in Python, the output of an operation on integers is an integer, and so rounding off will occur to force the output to be an integer. On the other hand, if you write y/3, you will get 0.6666… R does not behave this way. One way to get R to round the result, of course, is to use the **`round`**`()` function.

One important data type from computer logic is the Boolean (the term *logical* is also used instead of the word *Boolean*, and R uses this term).

A Boolean variable is a variable that can only take on two possible values: true or false. In R, these special values are denoted by TRUE or FALSE. (In other programming languages, the names *True* and *False* or *true* and *false* may be used.)

[1] Don't be confused into thinking about the answer to the question "Are you going home?" – thinking "yes" means true and "no" means false. That is not what we are referring to.

Usually we think of variables as being able to take on many values. In typical mathematical equations, we expect that variables can take an infinite number of values. For example, in the mathematical equation

$$y = 2x + 5$$

the variable x can be *any* real number. In contrast, Boolean variables can only assume *one of the two values*: true or false. Thus, Booleans are quite simple and limited. Nevertheless, they are very useful.

Consider the two statements

1. p: New York is in the United States.
2. q: 1 + 1 = 1.

First, to check our understanding, is it legitimate to use the word "statement" for each of these? Yes, because each of them is either true or false, not both.

"New York is in the United States"

is true. We say its *truth value* is TRUE.[2] On the other hand,

$$\text{"}1 + 1 = 1\text{"}$$

is false. We say its *truth value* is FALSE.

We are using this phrase "truth value" as we use the word "value" when we say that for the equation

$$x + 3 = 5$$

the value of x is 2. In other words, from all the possible values in the set of acceptable values for x (the real numbers), for *this* equation, x = 2. Remember though that for *statements*, there are only two possible truth values, true and false. Then for a particular statement, the truth value is one of these two possibilities. For example, "New York is in China" has a truth value of false.

Returning to the equation y = 2x + 5, in high school math, we might write

$$x \in \mathbf{R}$$

where ∈ means "is an element of" and R is the infinite set of all real numbers. Similarly, for a statement p, we could write

$$\text{"the truth value of p"} \in \{\text{true}, \text{false}\}$$

to emphasize that the truth value can only come from this very limited set of values. Note, we are not saying that p can *represent* only two possible statements. p can represent any statement, for example, "Today is Monday" or "The cost of the movie was $15". We are focusing on what we call the truth value of p. The truth value of p refers to whether p is true or p is false. It is the truth value of p that is limited to the set {true, false}. Thus, symbols for statements (like the symbol p),

[2] Using R's symbol for true.

have two separate properties. One of these properties is the particular statement they represent. The second property is the truth value of that statement. Similarly, we may say that x can have two separate properties as well: what it stands for and what its value is. For example, the mathematical variable (not logical variable) x may *stand for* the number of oranges in the refrigerator, but its *value* may be equal to 3. Our point here though is to make clear that the truth value of a statement will only be one of two possible values: TRUE or FALSE, whereas if x is the number of oranges in the refrigerator x could be any element from the set {0, 1, 2, … }.

Practice 2.3.2

1. What are the symbols in R for the values of a Boolean variable?
2. Look up another language like Python, Java, C, or C++ and find out how the values of a Boolean in that language are denoted.
3. Let p denote a statement. How many different possible statements are there that p can denote?
4. Let p denote a statement. How many different possible truth values are there for p?
5. For the equation x + y = 5, how many different values are typically possible for x?
6. For the equation x + 6 = 10, how many different values are *possible* for x (not how many make the expression true)?

```
#First, we explore the symbols in R for "true" and "false". We can
artificially define a Boolean vector like this
B <- c(FALSE,TRUE, TRUE)
class(B) #note that R uses the word "logical" instead of "Boolean".
## [1] "logical"
#FALSE and TRUE are not strings or characters even though they seem
similar to characters. But R treats these differently.
B
## [1] FALSE  TRUE  TRUE
#One of the differences is that when R reports back the values of a
character vector, it always uses quotation marks but for Boolean vectors,
as seen in B above, it does not use quotation marks. We will see more
important differences later.
C <- c("FALSE","TRUE","TRUE")
C
## [1] "FALSE" "TRUE"  "TRUE"
class(C)
## [1] "character"
#Whenever an expression is presented to R, it will evaluate it to find
its value. This is true regardless of whether the expression is a numeric
expression or a Boolean expression. For example, if the expression is
numeric, it will evaluate the expression and then return the value of the
expression. So if we type 8, of course R will respond with 8.
8
## [1] 8
#And if we write 6+1, then R will evaluate the expression and respond
with 7.
6+1
## [1] 7
#R responds similarly if we present it with a Boolean expression, (i.e.,
a statement). It will evaluate it and return the value. However, the
```

```
value of a Boolean expression is not a number, but rather either true or
false, and thus either true or false will be returned.
5<1
## [1] FALSE
5<9
## [1] TRUE
#Simpler evaluations are like this
TRUE
## [1] TRUE
FALSE
## [1] FALSE
#These two simpler examples are similar to what happens when R evaluates
8
8
## [1] 8
```

Practice 2.3.3

1. What is the class of 5 < 1? You can use the function class() to discover an object's class.
2. If p <- 5 < 1, what does p "evaluate to"? (Note that <- is the assignment operator, but < is the less than or equal to operator.)
3. If x <- 8 + 1, what does x "evaluate to"?

Now, we return to the discussion of statements. What about an expression like x < 5? Is it a statement? The answer is no. However, once we fill in x with a value then it becomes a statement. Thus, if we have the *for loop* in R

```
 for(x in 1:5){
  if(x>3){print(x)}
}
## [1] 4
## [1] 5
```

we are generating the following five separate expressions within the loop.

```
1>3
2>3
3>3
4>3
5>3
```

Each of these is a statement because each is either true or false but not both. The *truth values* for these are

```
FALSE
FALSE
FALSE
TRUE
TRUE
```

From the R printout, we see that only when the argument in the if() function is *true* will the body of the *if* clause, print(x), be run.

2.4 Compound Statements

"6 > 4 and 6 > 9"

is called a compound statement. A compound statement is formed from two simpler statements joined by what is called a logical operator (also called a logical connective). The logical operators we will use are "and", "or", "not", and "if… then".

Examples of compound statements using "and", "or", and "if … then" are as follows:

- 6 < 7 and 8 > 9.
- 6 < 7 or 8 > 9.
- If x *is divisible by 2* then x *is even*.

We next discuss each of the four connectives in detail.

2.5 Connectives

2.5.1 "and" – The Conjunction

Again p, q, and r will be used to denote statements. Any two statements can be joined by the word "and" to form a compound statement called the conjunction of the original statements. The connective "and" is called the conjunction. The conjunction of p and q is

p and q

Symbolically it is written as

p ^ q

It denotes the conjunction of the original statements and is read as "p and q".

For example, if

p: "New York is in the US"

q: "1 + 1 = 2"

then p ^ q is the compound *statement*

"New York is in the US and 1 + 1 = 2".

It is clear from these particular statements that the truth value of p is true and the truth value of q is true. What about the truth value of the statement p ^ q?

First, we should make sure that p ^ q is actually a statement. That is, can we identify it as true or false but not both? The answer is yes and of course that it is a true statement. We have combined the two statements using the conjunction operator.

This example suggests that the conjunction of two statements is itself a statement. Next we ask, under which conditions will the compound conjunctive statement be true – meaning which combination of truth values of the individual statements p and q will lead to p ^ q being true?

Given that there are two possibilities for the first element in the conjunction and two for the second element in the conjunction, there are only four possible combinations that need to be inspected. Consider the following:

i. NYC is in the US and 1 + 1 = 2
ii. NYC is in the US and 1 + 1 = 1
iii. NYC is in China and 1 + 1 = 2
iv. NYC is in China and 1 + 1 = 1.

This seems tedious and trivial, but by going through this exercise, it is possible to derive rules which help evaluating more complex statements. Which of the above compound statements are true and which are false? It is obvious using common sense that only (i) is true. The others are false. Furthermore, if we replace the statement

NYC is in the US with *Moscow is in Russia*

the true/false pattern remains the same. Only (i) would be true.

We can see that whenever the first element of the conjunction is true and the second is false, only (i) would be true. This suggests that we can build a table that shows when a compound statement that uses the conjunction connective will be true and when it will be false. Instead of using *NYC is in the US,* or *Moscow is in Russia,* or any particular statement, we just use p. We can use q for the other element of the compound statement (Table 2.1). Now we are exploring

p ^ q

Our investigation has shown that p ^ q can be tabulated as shown in Table 2.1.

In this table, T stands for true, and F stands for false, and the truth value of the compound statement is displayed in the third column. Also take note that the table shows the *truth values* for p and q, *not* the statements they represent. Notice also that in this table we have exhausted all possible combinations of the true and false values for p and q. To be clear, there is no reason we must use the order in the below table to list all these possible values. For example, we could have put F T in the first row instead of the third row. (Of course, then the third column of the first row would not be T anymore.)

Now, to put this kind of table in perspective, note that we could also make a table for the expression $x + y^2$. It might begin as shown in Table 2.2.

What is the point of bringing up this example here? In this table, we have chosen *only a few pairs of values for* x and y. If we wanted to "complete" this table, we would have to fill in every possible combination of values for x and y. This is an impossible task since there are an infinite number of possible combinations. However, for our logic table, with p and q, how many different

Table 2.1 Conjunction Truth Table

P	*q*	*p ^ q*
T	T	T
T	F	F
F	T	F
F	F	F

Table 2.2 Table of the function of x and y: $x + y^2$

x	*y*	$x + y^2$
1	1	2
1	2	5
1	3	10
2	5	27

possibilities do we have to fill in? Since there are only two possible truth values for p, true and false, and two possible truth values for q, true and false, there are only four possible combinations for the truth values of p and q together, and so we only need four rows! This is mentioned just to further drive home the point that we made earlier about the limits on the set of possible truth values of a Boolean statement:

$$\text{“the truth value of p”} \in \{\text{TRUE, FALSE}\}$$

and also to underscore the fact that the truth table for p ^ q is complete with the evaluation of exactly four rows.

Practice 2.5.4

1. To make a truth table for p ^ q ^ r, how many rows will be necessary?

2.6 “or” – Disjunction

Returning to the logical connectives, we can explore the connective “or”. It is called the disjunction. The compound statement “p or q” is symbolized in logic by p or q. To explore the nature of “or”, we can look at similar compound statements.

i. NYC is in the US or 1 + 1 = 2
ii. NYC is in the US or 1 + 1 = 1
iii. NYC is in China or 1 + 1 = 2
iv. NYC is in China or 1 + 1 = 1.

In English, the meaning of the word “or” is not necessarily clear. If you tell your child that she can have cake or she can have ice cream, but if she actually eats both, it may not be clear that she has disobeyed. Using our logic notation, when we have p or q, if both p and q are true, is p or q false? For the English word “or” the answer is not clear. This is not a good situation when programming. If we say “if x > 3 or y > 4, then … do something”, it should not be ambiguous what should happen if both x > 3 and y > 4 are true. Thus, we *decide* to *adopt the convention* the statement p or q will be true even if both p and q are true. We will use the symbol ‘v’ to stand for this meaning of ‘or’ and refer to this meaning of the word ‘or’ and this symbol ‘v’ as the “inclusive or”. Using the statements (i) through (iv) above as examples, we can now fill out Table 2.3 for the “inclusive or”, the disjunction p or q.

Table 2.3 Disjunction Truth Table

p	*q*	*p or q*
T	T	T
T	F	T
F	T	T
F	F	F

In R, the conjunction and disjunction look like:

```
#First, we consider a conjunction of two statements in R. The symbol for
a conjunction in R is either & or &&. They are used somewhat differently.
Here we consider only &

TRUE  & TRUE
## [1] TRUE
TRUE  & FALSE
## [1] FALSE
FALSE & TRUE
## [1] FALSE
FALSE & FALSE
## [1] FALSE
#Next consider a disjunction of two statements in R. The symbol for a
conjunction in R is either | or ||. They are used somewhat differently.
Here we consider only |.

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

Practice 2.6.5

1. Without opening R, predict what the outcome to the following will be; then open R and check the answers to the following. As expected, the parenthesis indicates the order in which the statements need to be evaluated.

```
(FALSE | FALSE) | FALSE
(FALSE | TRUE) & TRUE
(FALSE | TRUE) | FALSE
(FALSE | TRUE) | (FALSE & TRUE)
```

2.7 Negation

The next logical operator is the negation operator. The negation of p is simply "not p". Typical symbols for "not" in logic are "~" or "¬", so "not p" can be expressed in logic using either of

~p or

¬p.

How do we "translate" from logical symbol to English? If p is "NYC is in the US", then ~p *is* "NYC is not in the US". We could also express "not p" in English as "not NYC is in the US" although that is not great English. The "translation" from the logical notation to English, in general, is not necessarily straightforward. There may be numerous ways to say "not p". Perhaps in other languages, things may be more straightforward, but generally languages will have many ways to express the same concept. In English, some may prefer to think of "not p" as "NYC is not in the US", and others may prefer "not NYC is in the US".

Table 2.4 Negation Truth Table

p	~*p*
T	F
F	T

What is the truth table for ~p? Let us first note that you may be confused because this supposedly compound statement is not even made up of two smaller statements. There is a p, but there is no q. It would be fair not to call ~p a compound statement. Nevertheless, we still want to know the relationship between p and ~p. That is, we want to know if p is true, then what about "not p"? The answer is trivial, but we still will record it in a truth table (Table 2.4).

Note: We have been using the word "connective" rather than "operator" prior to discussing negation, but they are interchangeable. But when talking about negation, since there is only a p and no q, it is more natural to use the word "operator". Further, in the case of negation, we can call it a unary operator, whereas if we are talking about conjunction or disjunction, we can say binary operator. Binary in this case means there is a statement on either side of the conjunction; unary means there is a statement on just one side of the operator.

In R, this looks like

```
#The symbol for 'not' is "!"
!TRUE
## [1] FALSE
!FALSE
## [1] TRUE
#Let's try a few more examples to make this clearer.
5<3
## [1] FALSE
!(5<3)
## [1] TRUE
#Now let's combine some of these operators.
#First just the OR operator
5<3 | 2<3
## [1] TRUE
#Now a combination of the OR operator and the NOT operator:
5<3 | !(2<3)
## [1] FALSE
#Next, since this last expression was FALSE,
what if we combine it with a TRUE statement , this time using the OR
operator? Note the parentheses.
(5<3 | !(2<3))  |  10<100
## [1] TRUE
```

Practice 2.7.6

1. Without opening R, predict what the outcome to the following will be; then open R and check your answers.
   1. `! (FALSE | FALSE) | FALSE`
   2. `(FALSE | TRUE) & !TRUE`
   3. `(FALSE | !TRUE) | FALSE`
   4. `! (FALSE | TRUE) | ! (FALSE & TRUE)`

2.7.1 Implication: If … then

Our last logical connective is called implication and is expressed in English as "if … then". The symbol we use in logic is the arrow, so that the statement "if p then q" is written symbolically as

$$p \rightarrow q$$

It can be translated into English as

if p then q
p implies q
p only if q

This illustrates a point made earlier; translations from logical symbols to English are not straightforward and not necessarily unique[3].

Suppose we represent the statement "Your score on your final exam is at least 95" by p and we represent the statement "You will receive an A for the final course grade" by q, then $p \rightarrow q$ can be said in English as "If your score on your final exam is at least 95, then you will receive an A for the final course grade". Notice that both p and q are statements; i.e., they can be either true or false but not both. How about $p \rightarrow q$, is it a statement? The answer is yes. However, this time, rather than first exploring this connective with examples, as we did for AND, OR, and NOT, it may be easier to start with the truth table (Table 2.5).

Some rows of the implication truth table may be confusing. The third and fourth rows may seem strange. Before we get confused with the puzzling parts, let's look at the parts that are easily ascertained. Consider row 2. We have T F, F.

What is this saying? It is saying that if p is true and q is false, what would you say about the statement $p \rightarrow q$: if p then q? Again, if we have

p: "Your score on your final exam is at least 95"
q: "You will receive an A for the final course grade"

$p \rightarrow q$ can be translated as "If your score on your final exam is at least 95, then you will receive an A for the final course grade". Now suppose it is true that you do actually get at least 95 on your final exam *but* the instructor *does not give you an A*; then if the instructor had made the statement that "If your score on your final exam is at least 95, then you will receive an A for the final course grade", would that be true? No, you would say the instructor lied! You would say that the instructor's statement was false. So, if p is true and q is false, then $p \rightarrow q$ is a false statement, as shown in Table 2.5.

Table 2.5 Implication Truth Table

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

[3] In mathematical language, such a relation from logical expressions to English is not a function since it is one-to-many.

Considering the first row of the truth table where both p and q have truth values of T, it is clearly not false. The last two rows can be puzzling. The easiest way to understand these is not to ask if they are false but to ask if they are true. In logic, if you can't say a statement is true, then it is considered false.[4] For the fourth row, the question could be phrased as "Can you say your instructor lied if you don't get at least 95 and then the instructor does not give you an A?" In this case, the truth table says the instructor has not lied (that is, the statement is true). Similar logic applies for the third row. Nevertheless, these two rows may seem confusing. It may be possible that in English, this is not necessarily correct. However, we are running into a problem we faced with the disjunction. Whether it sounds correct in English is not exactly the point. The point is that this is how we *define* the arrow symbol. The arrow symbol may not correspond exactly to our notion of the English phrase "if then", but the truth table *is* the definition of the arrow symbol in logic.

2.8 Logical Equivalence

Two statements are said to be logically equivalent if they have the same truth tables. The symbol we will use for logical equivalence is ≡. Some seemingly obvious examples are

1. p and q ≡ q and p
2. p or q ≡ q or p
3. ~ ~p ≡ p.

   You can verify the above three examples by doing a truth table for the left-hand side and another truth table for the right-hand side and confirming that they have the same truth table. You can also get a feeling for why these are true by assigning values to p and q and seeing that the three examples above seem to be "logically equivalent". For example, if p is "today is hot" and q is "today is Monday", we would probably agree that saying p and q is the same as saying q and p. At least logically, they would be the same. Literally, they are not the same. Perhaps artistically, they may have some difference as well, but we say that logically they are the same.

   A more interesting example of logical equivalence is the logical equivalence of

4. ~ p or q ≡ p → q.

   If #4 is true, then this would mean that the implication symbol is redundant in logic and that we don't need an arrow symbol. We would call it redundant in the sense that we could always replace p → q with its logical equivalent ~ p or q, and so we would never need to use the arrow symbol. Of course, in normal conversation, being able to use a phrase like "if p then q" is most convenient; it is used all the time. However, for purposes of logic and programing, it can be replaced by "not p or q". This does not give very elegant English, but logically it is fine.

[4] You may not quite agree with this. That is fine. Perhaps logic, as we define things, does not quite map our actual experience and our actual world. That is, just because something is not true, perhaps it does not absolutely mean it is false. We are defining a set of rules here. They may not be a perfect fit to our actual experience. We may therefore end up proving things that don't seem to match our actual experiences, and that would be unfortunate. However, that is the price we may have to pay for making this (somewhat simple) system which we are calling computer logic or mathematical logic. Another way to look at this is that we are only considering statements; logical objects which can only be true or false but not both.

We are suggesting that the following two compound statements are logically equivalent.

- If I live in NYC, then I live in the US.
- Either I don't live in NYC, or I live in the US.

In fact, there is not even an operator in R for the implication sign.

Let us show now that the truth table for ~ p or q is the same as the truth table for p → q.

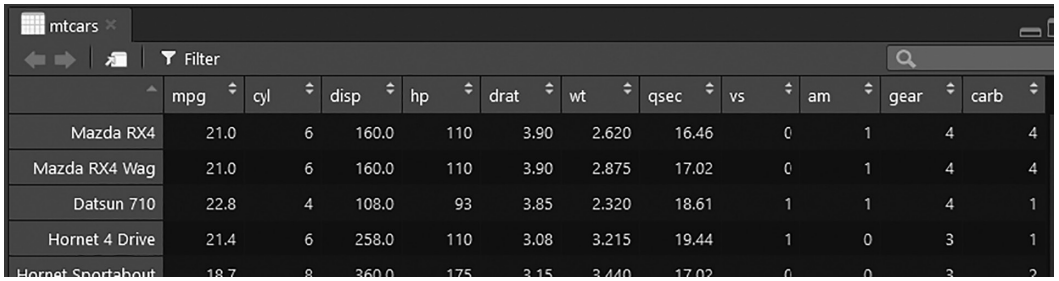
```
#Equivalence of implication and !p | q
#We already know the truth table for the implication operator. Now we
want the truth table for !p | q.
#We must use the same ordering of TRUE and FALSE that occurred in the
implication table.

#This is what the inputs to that table looked like.
#   p      q
# TRUE   TRUE
# TRUE   FALSE
# FALSE  TRUE
# FALSE  FALSE

#Next negate each p and then conjunct this with q to arrive at !p | q.
#We hope that the pattern will be the same as that we saw in the
implication table:
# TRUE
# FALSE
# TRUE
# TRUE

# !p    |  q
FALSE  | TRUE
## [1] TRUE
FALSE  | FALSE
## [1] FALSE
TRUE   | TRUE
## [1] TRUE
TRUE   | FALSE
## [1] TRUE
#Since the truth tables are the same, we have established their logical
equivalence, and p → q can be replaced with !p | q.
```

Practice 2.8.7

1. Show the following logical equivalences in R
   p ^ q ≡ q ^ p
   p or q ≡ q or p
   ~ ~p ≡ p

2.9 Implementing Logic in the Context of a Dataframe

Logical operators occur in R programming when, for example, we do subsetting and we want to select rows that satisfy the conjunction of two conditions. We will see this shortly.



	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

Figure 2.1 View of mtcars.

The primary data type in R is the vector. Scalars, in fact, are considered to be vectors of length 1.

Because some of the topics we cover in this text are originally derived from statistics, there will be occasions when we refer to *a row of a dataframe* as an *individual* or an *observational unit*. This language will be clarified in Chapter 4. For example, the preinstalled dataset in RStudio called "mtcars" has automobile data for various models of cars like the Mazda RX4, the Datsun 710, and so on. The data was extracted from the 1974 *MotorTrend* US magazine (Figure 2.1).

```
data("mtcars")
View(mtcars)
```

Each row corresponds to a model and shows various characteristics about that model such as the mpg, number of cylinders, the displacement of the engine, the horsepower, and so on. Each row is referred to as an *individual* even though this data is not about people. The word "individual" originates from the fact that many datasets are about people and each row would be a particular person and contain characteristics about that person. The word "individual" is still used in cases where the dataset does not represent people, and so we refer to each row of the mtcars dataset as an individual. Another term used sometimes instead of individual in statistics is *observational unit*.

We have studied Boolean statements like 5 > 2. We can also generate a Boolean expression within our dataset.

```
data("mtcars")
is.data.frame(mtcars) #check to see if it is a dataframe
## [1] TRUE
colnames(mtcars) #check to see what the column names are for this
dataframe
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"
"gear"
## [11] "carb"
df<-mtcars
#We know that the dataframe name followed by a $ operator followed by a
column name will give a vector.
head(df$mpg)
## [1] 21.0 21.0 22.8 21.4 18.7 18.1
is(df$mpg)
## [1] "numeric" "vector" #is() reveals that it is a vector and that the
elements of the vector are numeric.
is.vector(df$mpg)
## [1] TRUE
```

```
#There are various other ways to find the type of object in R. The
subject can be confusing because R has a heritage that includes the
S language and the S-Plus language. Different functions come from
different parts of the history of R. For comparison, you may want to look
into the following.

# class(df$mpg)
# str(df$mpg)
# mode(df$mpg)
# type(df$mpg)
# typeof(df$mpg)
# storage.mode(df$mpg)

#To isolate the mpg of a particular car, we can do
df$mpg[1]
## [1] 21
#or
df$mpg[2]
## [1] 21.
#Now to form a statement - a sentence that evaluates to TRUE or FAlSE, we
can employ one of the numeric operators we have used above like > or <,
or we can use some of the others that we have yet to use in this chapter,
for example, <=, >=, or ==. We form a comparison like
df$mpg[1]>25
## [1] FALSE
#We see that this is a statement; i.e., it evaluates to a Boolean value.

#Next, we do something similar but with vectors. We know that R is very
comfortable with vectors and that most things that can be done with a
single number can also be done with vectors. So instead of running
something like 10<2 where we have a number on each side of the less than
sign, we can have vectors on each side of the less than sign.

a <- 1:5
b <- 5:1

#We can use cat() and the new line symbol \n, to make a nice display of
vectors a and b where they are lined up vertically.

cat(" a: ",a,"\n", "b: ",b,"\n")
##  a:  1 2 3 4 5
##  b:  5 4 3 2 1
#Show the comparison of a and b using the less than sign.
a<b
## [1]  TRUE  TRUE FALSE FALSE FALSE
#We see the result is not just one Boolean,  but rather, since we are
comparing two vectors instead of two scalars, a vector of Booleans! We
will refer to it as a Boolean vector (or logical vector). Recall that
when R compares two vectors, it compares them element by element. Thus,
in the example of a<b, it will compare 1 and 5, 2 and 4, 3 and 3, 4 and
2, and 5 and 1.

#Something very similar can be done with the vector df$mpg. We can
compare it to another vector using the less than operator.
length(df$mpg)
```

```
## [1] 32
#We will construct a vector of the same length as df$mpg with each entry
being the number 25.
x <- rep(25,length(df$mpg))
x
##  [1] 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25
## [24] 25 25 25 25 25 25 25 25
#Now generate a Boolean vector by comparing df$mpg and x using greater
than operator.
df$mpg>x
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
## [23] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
#Of course, we do not actually need to create x because of R's so-called
recycling property. Note that for Python programmers this is very similar
to what is called broadcasting. Using recycling, to get the same
comparison, we can just write
df$mpg>25.
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
## [23] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
#Index Vectors Inside a Vector
#An index vector is a Boolean vector that is placed inside the subsetting
function []. Recall the vector "a" from above.
a
## [1] 1 2 3 4 5
#Put a Boolean vector of the same length as a inside the subsetting
function like this
u <- c(TRUE, FALSE,FALSE,FALSE,TRUE)
a[u]
## [1] 1 5
#We would call "u" the indexing vector. This will extract from "u" only
the elements of "a" that have an index of TRUE.
#We saw that df$mpg>25 is a Boolean vector, and we saw that it was the
same length as df$mpg. Therefore we can do the same thing we did with "a"
and "u", with df$mpg and df$mpg>25.
df$mpg[df$mpg>25]
## [1] 32.4 30.4 33.9 27.3 26.0 30.4
#Only the values where the Boolean vector df$mpg>25 is true appear above.
```

2.10 NA in Truth Table

When running comparisons using the arithmetic operators within the context of working with a dataframe, it is likely that some of the entries in the dataframe will be missing (those values are naturally referred to as missing values). If a compound statement occurs in such a scenario, this will generate a logical comparison between NA and either a TRUE or FALSE value. Although this is not a topic of logic, it will be worthwhile taking note of how R deals with this situation. It may be a useful exercise to ask, "How would you, if you were the designer of the R language, choose to handle the following comparisons?" There is no definitive correct answer for this.

TRUE & NA
FALSE & NA
TRUE | NA
FALSE | NA
NA | NA
NA & NA

You may choose to say that wherever there is an NA within a compound statement, the result should evaluate to NA. This is not exactly how R handles this scenario. Have a look at the printout here to see the way the R designers chose to handle NAs.

```
TRUE & NA
## [1] NA
FALSE & NA
## [1] FALSE
TRUE | NA
## [1] TRUE
FALSE | NA
## [1] NA
NA | NA
## [1] NA
NA & NA
## [1] NA
```

2.11 Conclusion

You might feel that what we have done in this chapter on logic is tedious. However, we have developed some nice machinery and skills that will boost our ability to understand programs and to think about code.

We have developed some concepts and language that will help us speak in more fluent and sophisticated terms in out next tasks in programming algorithms. As an analogy, imagine trying to learn algebra *before* knowing arithmetic. It is clearly better to learn arithmetic first and then algebra. In this chapter, we have learned the "arithmetic" of *statements* rather than the arithmetic of numbers. To continue the analogy, in elementary school, we learn about the arithmetic operations of plus, minus, multiplication, and division. Here we have developed *operations* for statements – and, or, not, if … then – and learned how to use these operations.

What we have studied in this chapter is just one part of what is often called Discrete Mathematics, and it is highly recommended that you study this topic as a course by itself. However, for this text, this chapter and the next on sets will be enough discrete math for our purposes.

2.11.1 Extraneous Notes

There are several functions in R which are useful for orienting yourself when you are writing code or trying to follow code that is already written. It is a good practice to use these periodically to refresh your memory of what type of objects you are working with. We will briefly discuss the following useful functions.

```
mode(), str(), class(), typeof(), storage.mode(), is()
```

It is written that mode() is a mutually exclusive classification of objects according to their basic structure. The basic (also referred to as atomic) modes are *numeric*, *complex*, *character*, and *logical* (Boolean). More complex objects have modes such as *list*, *function*, *array*, and so on. To say that mode is a mutually exclusive classification simply means that if an object has one type of mode, it can't have another. For example, if something has a numeric mode, it can't have a mode of character. This is not necessarily true for the properties we cover next.

"class" is a property assigned to an object that determines how generic functions operate with it. Examples of different possible classes are: "integer", "numeric", "list", etc. What is a generic function? We explain it next. We will use the function summary() as an example here although we will not have many occasions to use this function in our coding. But this will help us to understand the concept of generic functions and therefore give insight into the meaning of the class() function.

When we apply a function to an object, for example, when we apply the summary() function to the object x as in summary(x), what R does is look for the "version" of the summary() function that will be appropriate for the class of object that x is. The version of summary() for a numeric vector would of course be different than the version for a dataframe, and therefore the code to produce a summary would need to be different. For example, suppose x is a vector but z is a dataframe. We may write summary(x), and we may also write summary(z). But obviously, the calculation and the code used to do the calculation for summary(x) and summary(z) must be different. In R, it is said that summary() is the generic function. However, since the calculations for summary(x) and summary(z) are different, R checks the class of the argument and calls different functions depending on the class of that argument. In R, there is another function called summary.dataframe() which has the specific code used to generate the summary of a dataframe. There are also functions like summary.Date(), summary.matrix(), summary.factor(), and so on. (You can see them if you type "summary." into the console in RStudio.) Each of these will consist of different code since they are meant to be applied to objects of different classes. However, as a user, we do not have to type summary.dataframe() when we want to apply summary() to the dataframe z. We only need to type summary(z). We can use summary.dataframe() if we want but we do not need to. There is also a function called summary.default(), and in fact, this is the function that is used when we write summary(x), but what this function does is to search for the form which is appropriate to the class of the argument x. Generally the function that is used is the name of the generic function followed by a dot and then followed by the class of the object. Thus, if the class of p is factor and we want its summary, the specific function that would be used is summary.factor(). Another interesting point here is that when you type the name of a function into R without using the following parentheses, that is, if you just type summary instead of summary(), we see the code for that function. If this is done with summary.dataframe, we find the code for summary.dataframe(). If this is done with summary.default, we see the code for summary.default(). However, what about if you type summary only? What should R report back? It will not show the code of summary.dataframe(), nor will it show the code for summary.default(). All we see is a reference to the useMethod() function actually. This means that all that summary does is find the right function to use based on the class of the object. Also, if we run ?summary(x), we see in the R documentation, that summary is a generic function.

Regarding the class() function, R uses the class of an object to decide which version of function to use.[5] So if you want to know which function will be used on an object X when you type

[5] The discussion above is for S3 functions. R relies on the S programming language, and as the S programming language progressed, there were different versions like S3, S4, and S5.

summary(X), you can check the class of X with class(X). This will tell you the class, and you can therefore determine which version of summary() will be used.

If an object has no specific class assigned to it, such as a simple numeric vector, it's class is usually the same as its mode, by convention. However, the class of an object can be changed very simply and for that matter arbitrarily. For example, suppose x is a list, say x has two elements a character vector "a" "b" and a numeric vector 1:5. That is, suppose x = list(c("a", "b"), 1:5). If we write class(x), we will see that the class is *list*. However, we can change the class of x arbitrarily to "gorilla" simply by writing class(x) = "gorilla". Now if we write class(x), we will see that the class of x is "gorilla". Thus, the class could be defined by the user. (Typically, though, we would not have a need to change an object's class.)

Whereas class can be defined by the user, *typeof* cannot. For example, define a list as follows.

```
> x<-list(c("a","b"), c(1,2))
> class(x)
[1] "list"
> # However, we can change the class.
> class(x)<-"gorilla"
> class(x)
[1] "gorilla"

> typeof(x)
[1] "list"
# This won't work however.
> typeof(x)<-"newclass"
Error in typeof(x) <- "newclass" : could not find function "typeof<-"
```

The reason that typeof cannot be changed is that typeof determines the way R stores the data on the computer. This is thus a *physical* characteristic (because it determines the way the data is physically stored on the computer).

Tip: It is a good idea to update RStudio to the latest version. On the other hand, updating R itself is not always appropriate. Some packages are written only for earlier versions of R, and so if you wish to use those packages, you would need the appropriate version of R. Additionally, as mentioned at the beginning of this text, the code in this text is based on a particular version of R. You can, however, have various versions of R installed on your computer and switch between them in RStudio (in the options menu).