Part I

Scripting and Exploratory Data Analysis

1

Introduction to R

CONTENTS

1.1 Introduction

In this chapter, we introduce the statistical programming language *R*. We learn how to express computations in the *R* language, and we take the time now, as we learn the basics, to undersand the computational paradigm of the language. Rather than simply provide the nuts and bolts and code templates, we aim to explain the essential pieces of the language and how to think about *R*'s computational model. If we begin our introduction to *R* and go beyond a utilitarian approach to the language, then we can greatly simplify our programming tasks in the future.

While *R* is a specific language that you may or may not use extensively in the future, what you learn as we explore *R* is generally applicable to many different programming languages that you might use. For example, *R* is very similar to *Matlab*, and it also shares many of the same concepts as *Perl*, *Python*, *Java*, *C*, and *FORTRAN*. Although they are different languages, they share important commonalities that are essential for communicating about computations to others and to the computer.

1.2 Getting Started

We start the *R* programming environment by launching `RStudio` or the *R* graphical user interface (GUI) or by invoking the command `R` in the shell. If you have not installed *R* on your computer, please see the accompanying Web site noted in the preface. Once *R* is running, you should see the *prompt*, > in the console. This is where we type *R* expressions. Then, when we press the Return/Enter key, *R* evaluates this expression and prints a value in the console. Try it: enter `1 + 2` at the prompt, i.e.,

```
> 1 + 2
```

Press the Return key, and you should see:

```
[1] 3
```

Then, *R* presents the prompt again and is ready for us to type another expression in the console.

The expression `1 + 2` is an example of a simple arithmetic computation that *R* can perform. We describe these and other types of computations in the next section. Don't worry right now about the `[1]`, we explain what it means in Section 1.6.

1.3 Computations and Expressions

R provides an *interactive* environment where we can give an instruction, such as add 1 and 2, then press the Enter key to have the expression immediately evaluated. The answer is printed at the console, and we can repeat this process. This sequence of actions is called a read-evaluate-print loop, or REPL for short. When we type an *expression* at the *prompt* in *R*'s console and press the Enter key, we indicate that we want *R* to perform the *computation*. That is, we ask *R* to *evaluate* our expression. Here are examples of three kinds of expressions,

```
2 + 3
sample(7)
hist(precip)
```

When the first of these three expressions is evaluated, *R* *returns* the value `5`, which is printed to the console. For the second expression, we ask *R* to provide a random ordering of the integers from 1 through 7, and *R* returns something like

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

The third expression doesn't print any value to the console. Instead, evaluating this expression yields a plot as a *side effect*. The latter two of these three expressions are function-style expressions. For all three expressions, after the computation has been performed, the loop is complete and the prompt is available for another round of computations.

1.3.1 Arithmetic Expressions and Order of Operations

With *R*, we can perform many simple arithmetic expressions, similar to what we can do with a scientific calculator. The following are examples of simple arithmetic expressions:

```
8 - 9
4 * 5
10 / 3
7 ^ 2
9 %/% 2
11 %% 7
```

In addition to the basic operations like addition, subtraction, multiplication and division (`+`, `-`, `*`, `/`), we use ^ for exponentiation, `%/%` for integer division, and `%%` for modular arithmetic.

Of course, we can combine these arithmetic operations into more complex expressions. For example,

```
10 ^ 5 - 6 / 3
```

```
[1] 99998
```

Recall that 10^5 is 100 thousand, and when we subtract 6 divided by 3 (or 2), we get the result shown here.

Order of Operations
The following expression,

```
10 ^ (5 - 6 / 3)
```

returns a value of `1000`, not `99998`. Notice that this expression is very similar to `10 ^ 5 - 6 / 3` except for the addition of parentheses. These parentheses change the order of operations, which is why the return value is so different. As expected, the order in which operations are performed follows the precedence in algebra, i.e., exponentiation, then multiplication and division, followed by addition and subtraction. These operations are carried out from left to right. However parentheses can override this order. The first expression `10 ^ 5 - 6 / 3` has no parentheses so the first computation is to raise 10 to the power 5. Next is the division of 6 by 3, and lastly 2 is subtracted from 10^5. On the other hand, the second expression places parentheses about `5 - 6 / 3` so these computations are performed first. That is, we divide 6 by 3 and subtract the result (2) from 5 to get 3. The final result is 10^3 or 1000.

1.3.2 Call Expressions

We called two functions in our discussion of expressions; these were sample() and hist(). Functions contain code (usually several expressions) that perform a specific task. We provide the functions with inputs to use in carrying out this task. For example, the abs() function takes the absolute value of the input provided,

```
abs(-0.8)
```

```
[1] 0.8
```

These inputs are called *arguments* and the output from the computation is the *return value*. When we provide a function with a particular set of values for its arguments and press the Return key, we say we are *calling* or *invoking* the function. For now, we work with *R*'s many built-in functions. Later, in Chapter 5, we write our own functions.

We saw already that when we call `sample(7)`, *R* returns a random permutation of the numbers from 1 to 7, e.g.,

```
sample(7)
```

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

The input that we provide the sample() function is the largest integer in the sequence $1, 2, \ldots$ that we want permuted. However, sample() can take more than one input. It has 4 arguments; these are

```
args(sample)
```

```
function (x, size, replace = FALSE, prob = NULL)
```

The arguments to sample() have names, *x*, *size*, *replace*, and *prob*, and 2 of them, *replace* and *prob*, have default values of `FALSE` and `NULL`, respectively. This means that they are optional, i.e., we don't need to provide the inputs for these 2 arguments. If we don't provide them in our function call, then *R* simply uses the default values. The *size* argument does not have a default value, and if it is not supplied then *R* does the sensible thing –return a permutation of all of the values in *x*. For example, if we want to sample only 3 random values from the integers from 1 to 7, then we specify *x* (as always) and we also must specify the *size* argument. We do this with

```
sample(7, 3)
```

```
[1] 7 1 5
```

There are many ways to specify the arguments to a function call. We consider these in Section 5.4, after we learn more about the syntax of the *R* language.

Expression Syntax

An expression is an instruction to the computer. The computer evaluates the expressions that we write and returns a value. In order for the software to carry out our instructions, these directions must obey the grammar of the language.

Grammar

The *R* software uses blanks, commas, parentheses, algebraic and relational operators, and naming conventions to figure out the various parts of an expression and so determine the computation to perform. Importantly, if we understand how the software reads an expression, then we can more easily identify and correct syntax errors.

In the expression below:

```
round(abs(x - x^2), digits = 4)
```

R uses the parentheses, minus and exponentiation signs, comma, and equal sign to identify the variables and functions in this expression. Starting from the inner-most expression, the instructions are to square `x`, subtract this quantity from `x`, take the absolute value of the difference, and then round the resulting value to 4 significant digits.

Readability

We can eliminate the blanks in the expression above and code it as

```
round(abs(x-x^2),digits=4)
```

R can parse this expression and carry out the instructions, which are the same as in the previous expression. However, we need to be able to read the expressions that we write too, and it's much easier for us to read code that adopts some of the conventions of written English and places a space after a comma and before and after operators and numbers.

1.4 Variables and Assignment

In the previous section, we saw that when we provide *R* with an expression to evaluate, *R* prints the results to the console as output. We may want to save this output for future computations. We can do this by *assigning* the result of a computation to a name, e.g.,

```
x = 10 ^ 5 - 6 / 3
```

Now, when we type this expression and hit the return key, *R* does not print 99998 to the console. Instead, the return value is assigned to a *variable* named `x`. That is, the equals sign tells *R* to assign the result of the computation to the *variable* `x`. Now `x` is a name by which we refer to the value 99998. We can check the value of `x` by typing `x` at the prompt and hitting return,

```
x
```

```
[1] 99998
```

Here, when we type `x` at the prompt, the 'computation' that we have asked *R* to perform is simply to print the value of `x` to the console. We can also change the value associated with `x`, by assigning a new value to it. For example,

```
x = 1 + 3
x
```

```
[1] 4
```

This is one of two main ways to assign a value to a variable in *R*. In addition to `=`, we can also assign a value with `<-`, i.e.,

```
x <- 1 + 2
x
```

```
[1] 3
```

Either of these two forms of assignment can be used. We suggest that you choose one and stick with it. We consistently use `=` in this book.

Variables allow us to store values without needing to recompute them. Additionally, by storing a value, we reduce redundant calculations, which can help us avoid mistakes. Variables also allow us to write general expressions. For example, the length of the hypotenuse of a right triangle with sides `a` and `b` is

```
sqrt(a^2 + b^2)
```

Here, we can use this formula over and over for different triangles, by changing the values of `a` and `b` and re-evaluating this expression.

R uses "copying" semantics in assignments. That is, when we assign the value of `x` to `y`, then `y` gets the value of `x`, but the variable `y` is not "linked" to `x`. This means that when `x` is changed, `y` does not see that change. In the code below, `x` begins with the value of 3, then *R* copies this value in the assignment statement so that `y` has the value of 3. These two variables are unrelated after that so when we assign `x` the value of 10, `y` remains unchanged. Below is the code for this simple example,

```
x
```

```
[1] 3
```

```
y = x
x = 10
x
```

```
[1] 10
```

```
y
```

```
[1] 3
```

Again, `y` continues to have the value 3 after `x`'s value has changed.

1.5 Syntax and Parsing

How does *R* know what computations to perform? It breaks down an expression into parts, called tokens. From these parts, it can figure out what to do. This is very similar to how we read and understand text. When we read, we use punctuation, such as a period, comma, semicolon, quotation marks, etc., as well as blanks and capitalization, to make sense of what's written. These conventions help us figure out what the person who wrote the text is saying. For example, the blanks, capitalization and punctuation have been stripped from some text that begins,

```
hatheads...
```

Without blanks to identify words, and punctuation to identify sentences, phrases and contractions, we don't know if this text begins as

```
Ha! The ad's
```

as in "Ha! The ad's finished", or

```
Hat! Heads
```

as in: "Hat! Heads need to keep warm in winter." The basic conventions that *R* uses for parsing code are listed in Table 1.1.

Tokens
R breaks an expression up into meaningful pieces, called *tokens*. Similar to English, *R* uses blanks and quotation marks to identify tokens. Tokens include arithmetic operations and variable names. In *R*, the expression `2*3+1` and `2 * 3 + 1` are equivalent. The atomic tokens, `*` and +, let us know that the number 2 is multiplied by 3 and then 1 is added. The blanks are not needed here, but they make it easier for us to read the expression. On the other hand, the expression `sqrt(17)` and `s qrt(17)` are definitely not the same. The blank in the second expression implies that we want to call the qrt() function, not sqrt(). We adopt the convention of placing blanks in expressions to help readability, which we discuss more in Section 5.8.

Naming Conventions
Naming conventions for variables also help *R* parse expressions. Variable names cannot start with a digit or underscore, can contain numbers, upper and lower case letters, and some punctuation. The `.` and _ are allowed, but most others are not. Also, upper and lower case letters are not the same so `X` and `x` refer to different variables.

New Lines
R uses a 'new line' to parse expressions. We produce a "new line" when we hit the Return key. In all of the expressions we have written so far, we end the expression when we press the Enter key. Then, *R* carries out the calculation and returns the value. However, a new line does not always indicate the end of an expression. We can split an expression over multiple lines in the console. For example,

```
> 10 ^ 5 -
+ 6 / 3


[1] 99998
```

TABLE 1.1: Parsing *R* Expressions

Convention	**Example**
White space	`abs(-2)` and `abs( - 2 )` are equivalent expressions, but the expression: `a bs(-2)` is different.
Atomic tokens	In addition to arithmetic tokens like + and `^`, there is also `#` which stands for comment, `(` and `)`, boolean algebra tokens, such as `&` and `|`, and relational operators, such as > and `!=`.
Quotation marks	These may be, `"Hi"` or `'Bye'`, but they must match, e.g., `"My'` is not valid.
Naming conventions	For example, `x22` is a valid name but `2x` is not because it does not begin with a letter or period.
New line	When working at the console or writing an *R* script, if we start a new line, then this indicates the end of the expression, unless *R* detects that the expression is incomplete.

Notice the + in the second line above. This symbol appears rather than the typical > prompt to indicate that the expression on the first line is not complete and *R* is waiting for us to write the rest of the expression on this continuation line. We can even spread this expression over four lines with

```
> 10 ^
+ 5 -
+ 6 /
+ 3
```

```
[1] 99998
```

Each line ends with an arithmetic operator so *R* continues the expression on the next line and looks for the second number. This is not a very clearly written expression! However, we may at times have long expressions and breaking them up across lines helps with the readability of the code. Of course, if we try to break up our expression after, say, the 6, then we do not get the expected results:

```
> 10 ^ 5 - 6
```

```
[1] 99994
```

Instead of providing us with a continuation line for us to type `/ 3`, *R* evaluates `10 ^ 5 - 6`. This happens because `10 ^ 5 - 6` is a valid *R* expression so *R* evaluates it and returns the value, 99994. Since we have provided *R* with a valid expression, *R* can't discern that we have not finished writing our expression.

Note that we typically do not include the prompts (> and +) in our code display. We do here only to make clear how *R* parses these expressions.

Compound Expressions

We have seen several simple call expressions, e.g., `sample(3, 7)` and `sqrt(16)`. A compound call expression is like a compound function in algebra, e.g., $f(g(x))$. Recall from algebra that the return value from evaluating $g(x)$ is passed to f as input to that function. For an example in *R*, say we want the integer part of the square root of `x`. Then, we can pass the return value of `sqrt(x)` to the floor() function; that is,

```
floor(sqrt(x))
```

```
[1] 3
```

(Recall that `x` has the value `10`). We can make compound expressions with functions that use more than one argument, e.g, `round(sqrt(10), digits = 2)` returns `3.16`.

Ill-formed Expressions
An ill-formed expression is one that *R* cannot properly parse. For example,

```
floor(sqrt(x)]
```

```
Error: unexpected ']' in "floor(sqrt(x)]"
```

Here, we have no return value because *R* can't parse our expression. Do you see the mistake? The error message indicates that the right square bracket is unexpected. What does *R* expect? A right parentheses. If we understand how *R* parses expressions, then that can help us figure out our mistakes and easily correct them. See, if you can spot the errors in the following expressions:

```
round(sqrt(10)), digits = 2)
```

```
Error: unexpected ',' in "round(sqrt(10)),"
```

```
round(sqrt(10, digits = 2))
```

```
Error in sqrt(10, digits = 2) :
  2 arguments passed to 'sqrt' which requires 1
```

Can you understand what the error message tells us about these ill-formed expressions?

Types of Expressions
In the examples below, `x` contains the value `10`.

Arithmetic: `2 + x^2`
Arithmetic expressions, are composed with the typical operators, e.g., `+`, `-`, `*`, and `/`, for addition, subtraction, multiplication, and division. The order of evaluation follows the rules of precedence in algebra and parentheses can override. This particular expression evaluates to `102`.

Relational: `x > 2`
Relational expressions use the operators such as `<`, `<=`, and `==` for less than, less than or equal to, and equal to, respectively. These expressions evaluate to `TRUE` or `FALSE`. This particular expression returns `TRUE`.

Boolean: `x > 20 | x == 0`
Boolean expressions operate on logical values with operators such as `&` and `!` for and and or. This particular expression evaluates to `FALSE`.

Assignment: `y = 2 + x^2`
The return value from a computation can be assigned to a variable. This variable can then be used in other expressions. Here `y` is `102`.

Call: `sqrt(x)`
Expressions can invoke, or call, functions, e.g., this expression computes the square root of `x`. Functions can have multiple arguments; some arguments may be required, meaning that we must provide a value for them when we call the function. Other arguments may be optional; a default value is provided by the function, and we can

override this default in the call. For example, the round() function has an argument *x* with no default value and an argument *digits*, which has a default of 0. We can assign `sqrt(x)` to `z` and round the value in `z` to 1 significant digit with `round(z, digits = 1)` and the return value is `3.2`. See Section 5.4 for more details on how to specify parameter values in a function call.

Compound: `round(sqrt(abs(x - y + 5)), 1)`
Functions can be nested, i.e., we can compose functions as in algebra. Here we have nested an arithmetic expression within 3 function calls. The first computation is `x - y + 5`, which returns `-87`. Then, this is passed into abs(), which returns `87`, and `87` is passed to sqrt() which returns about `9.327`, and lastly, this result is the input to the round() function. The second argument to round() is `1` so `9.3` is returned.

1.6 Data Types

In *R*, *vectors* are the primitive objects. A vector is simply an ordered collection of values grouped together into a single container. Some primitive types of vectors are numeric, logical, and character. A very important characteristic of these vectors is that they can only store values of the same type. In other words, a vector contains values that are homogeneous primitive elements. A *numeric* vector contains real numbers, a *logical* vector stores values that are either `TRUE` or `FALSE`, and *character* vectors store strings.

Example 1-1 Vectors of Measurements on a Family
We have created some artificial data on a 14-member family to help us explore many of the concepts in this chapter. These data are available in an RDA file for you to load into your *R* session and follow along by typing in the commands shown. We begin by loading the RDA file with

```
load("family.rda")
```

The names of the family members are in the vector called `fnames`, and we can see its contents with

```
fnames
```

```
[1] "Tom"    "Maya"   "Joe"    "Robert" "Sue"    "Liz"    "Jon"
[8] "Sally"  "Tim"    "Tom"    "Ann"    "Dan"    "Art"    "Zoe"
```

The numbers `[1]` and `[8]` are there to help us keep track of the order of the elements in the vector. We see that `"Tom"` is 1st, `"Maya"` 2nd, ..., `"Sally"` 8th, ..., and `Zoe` 14th. We can confirm that `fnames` is indeed a character vector by calling the class() function and passing the vector `fnames` as input, i.e.,

```
class(fnames)
```

```
[1] "character"
```

There are several other vectors with information on the family, including: `fweight`, weight in pounds; `fbmi`, body mass index (BMI); `foverWt`, whether or not BMI is above 25; and `fsex`, which is `f` for female and `m` for male. These vectors provide examples of several data types. The variable, `fbmi` is a numeric vector,

```
fbmi
```

```
 [1] 25.16239 21.50106 24.45884 24.48414 18.06089 28.94981 28.18797
 [8] 20.67783 26.66430 30.04911 26.05364 22.64384 24.26126 22.91060
```

and `foverWt` is a logical vector,

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

■

Another type of number in *R* is the *integer*. The integer vector is similar to a numeric vector, except that the values must all be integers. The variable `fage`, which contains the person's age in years, is an example,

```
class(fage)
```

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

Factor
The *factor* is a somewhat special data type for use with qualitative measurements. The values are internally stored as integers, but each integer corresponds to a *level*, which is held as a character string. The values of `fsex`, which is a factor vector, are:

```
 [1] m f m m f f m f m m f m m f
Levels: m f
```

Notice that the values are not printed with quotation marks, as with character values. We confirm the data type of fsex() with a call to class(),

```
class(fsex)
```

```
[1] "factor"
```

Also, the levels() function provides the levels or labels associated with the vector.

Special Values
R provides a few special values, including `NULL`, `NA`, `NaN`, and `Inf`. These stand for null or empty, not available, not a number, and infinity, respectively. `NULL` denotes an empty vector. The value `NA` can be an element of a vector of any type. It is different from the character string `"NA"`. We can check for the presence of `NA` values in a vector with the function is.na() and for an empty vector with is.null().

The special values 'not a number' and infinity come about from computations. In particular, they can occur when we divide by 0. Here are three examples that return infinity, negative infinity, and not a number, respectively: `12 / 0`, `-100 / 0`, and `0 / 0`.

1.6.1 Finding Information on Vectors

R has many utility functions that provide information about vectors (and other objects that we will soon learn about). We mentioned already the two functions is.na() and is.null(), which provide us with information about the presence of `NA`s in a vector or whether or not a vector is empty, respectively. We may also be interested in the vector's type. We can determine this with the class() function, or we can ask specifically if a vector is of a certain type with functions, such as is.factor(), is.logical(), etc. We can find the number of elements in a vector with length(); the first or last few values with head() and tail(), respectively; and the names of elements with names().

Example 1-2 Finding Details on the Family

Let's use some of these functions to find out more about the family. The length of `fnames`, which contains the names of the family members, is

```
length(fnames)
```

```
[1] 14
```

We can confirm that all of the vectors with family information have length 14 with further calls to length(), e.g., `length(fbmi)`.

We may also want summary information about, say, the weights of the family members. We may want to know the smallest and largest values, the average, or the median weight. We can find these with, respectively, min(), max(), mean(), and median(). For example the smallest weight in the family is

```
min(fweight)
```

```
[1] 98
```

Some of these functions return a single value, such as the minimum weight just calculated. Others return a value for each element of the input vector. One example is the names() function, e.g.,

```
names(fheight)
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
```

The names() function returns a character vector the same length as the input vector and each element of the return value is the name of the corresponding element of the input vector. For example, `"c"` is the name of the third element in `fheight`. Not all vectors have named elements, and when this is the case, the return value is an empty vector, e.g.,

```
names(fweight)
```

```
NULL
```

Often it can be helpful to examine the first few or last few values in a vector to confirm that the data are what we expect. The head() and tail() functions, respectively, return these initial or final values. By default we are given the first (last) 6 elements, i.e., the return value is a vector of length 6, e.g,.

```
head(fweight)
```

```
[1] 175 125 185 156  98 190
```

However, we can request more or fewer elements by specifying a value for the *n* argument, e.g., to view the last two elements in `fweight` we use `tail(fweight, n = 2)` and the return value is `150 125`.

In addition to finding the type of a vector with the class() function, e.g.,

```
class(fsex)
```

```
[1] "factor"
```

we can ask whether or not a vector is a particular type. For example,

```
is.factor(fsex)
```

```
[1] TRUE
```

Similarly, `is.integer(fbmi)` returns `FALSE`, but `is.numeric(fbmi)` returns `TRUE`. The `fage` variable is an integer vector, what do you think the call `is.numeric(fage)` returns? Try it and see. Since an integer is a special case of a number, *R* returns `TRUE` in this case.

■

Data Types

Primitive Types

In *R*, *vectors* are the basic or primitive objects. A vector is an *ordered* container of *homogeneous* values. In other words, a vector contains values that are the same type and these values have an ordering. Statisticians analyze measurements of some quantity on a group, and the vector is a convenient structure for this purpose.

The primitive types include:

- `numeric`– real numbers,
- `logical`–`TRUE` and `FALSE` only,
- `character`– strings.

Factor Type

Qualitative measurements are an important kind of data, e.g., sex, marital status, and education level, and *R* has a `factor` data type for this purpose, i.e.,

- `factor`– values are stored as integers, and each integer corresponds to a *level*, which is a string. The levels in a factor can be ordered, e.g., education level.

Typically we want to perform different types of calculations with `factor` data, e.g., for a summary, we want tallies of the number of observations at each level, not a mean or median. Many *R* functions, e.g., summary(), perform different operations on a vector depending on whether it is `numeric` or `factor`.

1.7 Vectorized Operations

The philosophy in *R* is that operations work on an entire vector. This makes sense given that vectors are the basic data types. A simple example is with subtraction. We can subtract one vector from another element-wise using the `-` operator. For example, if we want to know the difference between the actual weight and desired weight for our family members, then we can simply subtract `fdesiredWt` from `fweight`. That is,

```
fweight - fdesiredWt
```

```
[1]   0  10  10   6 -12  40  25   6  10  20  16   4  10   0
```

Here, the 1st element of `fdesiredWt` is subtracted from the 1st element of `fweight` to get 0, the 2nd element of `fdesiredWt` is subtracted from the 2nd element of `fweight` to get 10, and so on.

The notion of vectorized operations is very powerful and convenient. It allows us to express computations at a high-level, indicating what we mean rather than hiding it in a loop.

Example 1-3 Computing BMI

Although we are provided with the BMI of each family member in the `fbmi` variable, we can compute this quantity ourselves from the height and weight of each individual. The formula for BMI is

$$\frac{weight\ in\ kg}{(height\ in\ m)^2}.$$

If we use this formula, we need to convert our measurements from pounds into kilograms and from inches into meters. Since there are 2.2 pounds to a kilogram, we can change the units for the values in `fweight` with

```
fweight / 2.2
```

```
 [1] 79.55 56.82 84.09 70.91 44.55 86.36 84.09 56.36 79.55
[10] 97.73 75.45 63.64 68.18 56.82
```

Similarly, we can convert inches to meters with

```
fheight * 0.0254
```

```
    a     b     c     d     e     f     g     h     i     j
1.778 1.626 1.854 1.702 1.549 1.727 1.727 1.651 1.727 1.803
    k     l     m     n
1.702 1.676 1.676 1.575
```

We see that this return vector appears somewhat differently than the return from dividing `fweight` by 2.2. This is because the elements in `fheight` are named so the return value has these names too.

We can combine these calculations into a single calculation of BMI with

```
(fweight / 2.2) / (fheight * 0.0254)^2
```

```
    a     b     c     d     e     f     g     h     i     j
25.16 21.50 24.46 24.48 18.56 28.95 28.19 20.68 26.66 30.05
    k     l     m     n
26.05 22.64 24.26 22.91
```

Again, the names of the elements in `fheight` have been carried over to the return value. Also, we can examine `fweight` and `fheight` to check that, e.g., the 3rd weight in `fweight` and the 3rd height in `fheight` are properly combined to create the 3rd BMI in the return value.

For greater clarity, we can create intermediate variables for the converted height and weight with

```
WtKg = fweight / 2.2
HtM = fheight * 0.0254
bmi = WtKg / HtM^2
```

We have assigned our computation of BMI to the variable `bmi`.

■

1.7.1 Aggregator Functions

Some vectorized functions work on all of the elements of a vector, but return only a single value for the result. We have seen examples of these already, others include mean(), min(), max(), sum(), prod(), and median(). For example, the average BMI for the family is `mean¬ (bmi)`, which evaluates to about `24.61`.

1.7.2 Relational Operations

In addition to the arithmetic operators, such as `+` and `*`, *R* also has relational operators for comparing values. These operators are greater than (`>`), less than (`<`), greater than or equal to (`>=`), less than or equal to (`<=`), not equal to (`!=`), and equal to (`==`). The relational operators are vectorized, meaning that if you give them a vector of length n, they operate on all n elements.

Example 1-4 Compute Who is Over Weight

We can use relational operators to determine which of the family members are over weight. Our definition of over weight is a BMI that exceeds 25. Again, although the variable `foverWt` already contains this information, we can compute it ourselves by comparing `bmi` (`fbmi`) to 25. We do this with

```
overWt = bmi > 25
overWt
```

```
    a     b     c     d     e     f     g     h     i     j 
 TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE 
    k     l     m     n 
 TRUE FALSE FALSE FALSE 
```

We can compare our vector `overWt` to the one supplied for the family (i.e., `foverWt`) to see if they match. To do this, we can use another relational operator, namely the 'equal to' operator (`==`).

```
overWt == foverWt
```

```
   a    b    c    d    e    f    g    h    i    j 
TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE 
   k    l    m    n 
TRUE TRUE TRUE TRUE 
```

Note that this computation checks that the 1st element of `overWt` equals the 1st element of `foverWt`, the 2nd element of `overWt` equals the 2nd element of `foverWt`, and so on. All of these values are `TRUE` so we have consistent results, i.e., the element of these two vectors have the same values. Rather than examine each of the 14 values to see if they are all `TRUE`, we can pass the return vector from our comparison to the all() function.

```
all(overWt == foverWt)
```

```
[1] TRUE
```

The all() function returns `TRUE` if all of the elements in the input vector are `TRUE`.

Another helpful function for determining if two objects are the same is the identical() function. We call it with

```
identical(overWt, foverWt)
```

```
[1] FALSE
```

This is somewhat surprising, given that we just compared the values of each of the elements and found them all to be the same. Can you figure out what is the problem? Notice that our new variable `overWt` is a vector with named elements, but `foverWt` is not. The identical() function checks more than the values of the elements. These two vectors are not identical because one has named elements and the other does not.

■

In addition to the relational operators, many functions in *R* are vectorized. The nchar() function is one example. If we give nchar() the character vector of family member names, then we get

```
nchar(fnames)
```

```
[1] 3 4 3 6 3 3 3 5 3 3 3 3 3 3
```

Here we have a vector of length 14 that contains mostly 3s because most family members' names have only 3 letters in them.

1.7.3 Boolean Algebra

Boolean operators take logical vectors as inputs and perform Boolean algebra. The three common operations are "not", "or", and "and". The "not" operator, which is `!` in *R*, is a unary operator because it has only one input. It turns `TRUE` into `FALSE` and vice versa. For example, `!foverWt` returns

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

This vector has `TRUE` for individuals who are not over weight and `FALSE` for those who are.

The "and" operator, `&`, compares the elements of two vectors and returns a logical vector where `TRUE` indicates the corresponding elements in the input vectors are both `TRUE`, and `FALSE` indicates otherwise. The "or" operator, `|`, compares the elements of two vectors and returns a logical vector where `TRUE` denotes that either one or the other or both of the corresponding elements in the input vectors are `TRUE`. Of course, these operations can be combined into compound statements. We provide an example.

Example 1-5 Identifying Certain Family Members

Let's suppose we are interested in identifying female family members who are either over weight or under 45 years old. Then, we can create a logical vector that indicates these characteristics with the following expression:

```
fsex == "f" & (fage < 45 | foverWt)
```

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

We can check the values of `fsex`, `fage` and `foverWt` to confirm that our algebra is correct.

■

In addition to all(), another function that operate on logical vectors and can be quite useful is any(). As demonstrated in Q.1-4 (page 17), the all() function returns `TRUE` if all of the elements of the logical vector are `TRUE`. The any() function returns `TRUE` if any of the elements are `TRUE`. For example, since some of the elements in `foverWt` are `TRUE` and some are `FALSE`, we find `any(foverWt)` returns `TRUE`, and `all(foverWt)` is `FALSE`.

1.7.4 Coercion

At times, we may want to change the type of a vector, e.g., from logical to numeric so that the vector contains 1s and 0s rather than `TRUE`s and `FALSE`s. We can do this with the collection of "as." functions, e.g., as.numeric() attempts to convert the input vector into a numeric vector. We try it with the logical vector `foverWt`

```
as.numeric(foverWt)
```

```
 [1] 1 0 0 0 0 1 1 0 1 1 1 0 0 0
```

As expected, a value of `TRUE` is converted to 1 and `FALSE` to 0. When we try to convert `fnames`, which is a character vector of the family member names, we get:

```
as.numeric(fnames)
```

```
 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA
Warning message:
NAs introduced by coercion
```

In this case, the conversion results in a vector of `NA` values. Notice also that a warning message is issued to let us know that we have missing values in our result. *R* can convert some character values into numbers, e.g.,

```
as.numeric(" -17.01")
```

```
[1] -17.01
```

However, when a string is not made up of digits (and a possible period and negative sign), then the conversion results in `NA`.

The reverse coercion, i.e., converting a number into a character string, works as expected, with a few subtleties as shown in the next example.

Example 1-6 How Many Digits are in `fbmi`?

We can convert the numeric vector, `fbmi`, into a character vector with

```
as.character(fbmi)
```

```
 [1] "25.1623879871136" "21.5010639538325" ...
```

These first two elements look a bit different from what we have seen before. When we print `fbmi` at the console, we see

```
head(fbmi)
```

```
[1] 25.16 21.50 24.46 24.48 18.51 28.95
```

It appears that the first two values of `fbmi` are 25.16 and 21.50, not 25.1623879871136 and 21.5010639538325.

This is simply a matter of controlling the number of digits printed to the console. If we want more digits printed for `fbmi`, then we can explicitly specify this with the print() function, e.g.,

```
print(fbmi, digits = 22)
```

```
[1] 25.16238798711363244820 21.50106395383245327935 ...
```

Now we have more digits than in the character strings! We discuss the topic of how numbers are represented in *R* in Chapter 10. For now, we simply recognize that what prints to the console may be different from the actual values in the vector.

We also mention that if we want to change the default number of digits printed for all subsequent computations in our *R* session, then we can do this through the options() function, e.g.,

```
options(digits = 12)
fbmi
```

```
[1] 25.1623879871 21.5010639538 ...
```

This example shows us that we should be careful when reading and comparing numeric values because what is printed at the console is not necessarily the actual value in a vector.
■

Converting Factors

Factors are a special data type and coercion of a factor has special behaviors. Recall that a factor consists of integer values where each integer represents a level, and a level is associated with a character string. When we coerce a factor into a number, we get the integer value of the level. However, when we coerce a factor into a character string, then the return value is the label for the level. We demonstrate with `fsex`:

```
as.numeric(fsex)
```

```
 [1] 1 2 1 1 2 2 1 2 1 1 2 1 1 2
```

```
as.character(fsex)
```

```
 [1] "m" "f" "m" "m" "f" "f" "m" "f" "m" "m" "f" "m" "m" "f
```

When we performed the relational operation, `fsex == "f"` in Q.1-5 (page 18), *R* implicitly converted `fsex` to a character vector before performing the comparison.

Implicit Coercion

Implicit coercion occurs when we operate on a vector in a way that is not intended for its type. For example, if we add `1` to a logical vector, then the logical values are converted to 0s and 1s implicitly, and `1` is added to each element.

```
1 + foverWt
```

```
[1] 2 1 1 1 1 2 2 1 2 2 2 1 1 1
```

As another example, when we use the logical operator `>` to compare the values in `fweight` to `"150"`, *R* determines that it can convert `"150"` to a numeric value; that is, `fweight > '150'` produces

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

In general, with the exception of adding 0 or 1 to a logical vector to convert it to numeric, it's best to avoid implicit coercion. It can produce nonsensical and unexpected results, if we don't understand well enough how coercion works. For example, can you figure out why the following comparison yields all `FALSE`s?

```
fweight > "abc"
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE ...
```

To help you, examine the built-in vector `letters` and see if you can make sense of the comparison, `letters > "c"`.

1.8 Data Frames

We have used information about a 14-member family to demonstrate several features of the *R* language. This information appears in several vectors, each with 14 elements. It's natural for us to picture this information arranged in a table, where each column of the table corresponds to a variable, like name, age, height and weight, and each row corresponds to a particular individual, such as the 77-year old Tom in the first row and the 27 year-old Sue in the 5th row. *R* provides a data structure, called a *data frame*, for collecting vectors into one object, which we can imagine as a table. More specifically, a data frame is an ordered collection of vectors, where the vectors must all be the same length but can be different types.

Example 1-7 A Data Frame of Measurements on a Family

The workspace that we loaded into our *R* session in Q.1-1 (page 12) contains a data frame for our 14-member family. This data frame is called `family`. We can use head() to examine the first few rows of `family` with

```
head(family)
```

```
  firstName sex age height weight   bmi overWt
1       Tom   m  77     70    175 25.16   TRUE
2      Maya   f  33     64    125 21.50  FALSE
3       Joe   m  79     73    185 24.46  FALSE
4    Robert   m  47     67    156 24.48  FALSE
5       Sue   f  27     61     98 18.51  FALSE
6       Liz   f  33     68    190 28.95   TRUE
```

The names of the variables in the data frame are not exactly the same as the names of the individual vectors, but we can see that these data are the same as in the separate vectors, e.g., `fbmi` and the column labeled `bmi` have the same values in the same order. These vectors are heterogenous in type. That is, `firstName` is a character vector, `sex` a factor, `age` is integer, `bmi` is numeric, and `overWt` is logical. The ordering of the vectors in the data frame is `firstName`, `sex`, `age`, and so on.

The types of some of the vectors are not immediately apparent from examining the head of the data frame. For example `firstName` and `sex` look similar in type. We can try to confirm this with a call to the class() function:

```
class(family)
```

```
[1] "data.frame"
```

We did not get what we expected, i.e., the data types of all the variables in `family`. Calling class() with `family` returns the type of the `family` object, not the vectors that it contains. We figure out shortly how to get the data types of the vectors in `family`.

We can find out additional information about the data frame with some of the same functions that we used to find out information about vectors. For example, length() returns the number of vectors in the data frame, names() gives us the names of the vectors. Additionally, dim() provides the number of rows and columns in the data frame.

■

`Dollar-sign` Notation

To access the vectors within a data frame we can use the $-notation. For example, we find the class of `firstName` and `sex` in `family` with

```
class(family$firstName)
```

```
[1] "character"
```

```
class(family$sex)
```

```
[1] "factor"
```

Now we have the answer that alluded us in Q.1-7 (page 21).

Example 1-8 Exploring the `family` Data

We can use the $-notation to pass a vector in the `family` data frame as input to any of the functions we have introduced that accept vectors as inputs. Of course, our family data is artificial so we can not make too much of what we find in our exploration, but it gives an idea as to what is possible.

We can find the average height and weight of the family members with, respectively,

```
mean(family$height)
```

```
[1] 67.86
```

```
mean(family$weight)
```

```
[1] 157.8
```

Alternatively, the summary() function accepts a data frame as input and provides summary statistics for each variable. We call summary() with

```
summary(family)
```

```
  firstName            sex          age            height     
 Length:14          m:8    Min.   :24.0   Min.   :61.0  
 Class :character   f:6    1st Qu.:33.0   1st Qu.:65.2  
 Mode  :character          Median :47.5   Median :67.0  
                           Mean   :48.1   Mean   :66.9  
                           3rd Qu.:58.0   3rd Qu.:68.0  
                           Max.   :79.0   Max.   :73.0  
     weight           bmi          overWt       
 Min.   : 98   Min.   :18.5   Mode :logical  
 1st Qu.:129   1st Qu.:22.7   FALSE:8        
 Median :161   Median :24.5   TRUE :6        
 Mean   :158   Mean   :24.6   NA's :0        
 3rd Qu.:182   3rd Qu.:26.5                  
 Max.   :215   Max.   :30.0                  
```

Notice that summary() does not provide the same summary statistics for all the vectors in `family`. The factor, `sex`, is summarized with counts of the number of elements in each level, and the same type of summary is provided for the logical `overWt`. The character vector `firstName` is summarized only by its length. The summary of integer and numeric variables include mean and median, minimum and maximum, and upper and lower quartiles.

Lastly, we can also make a scatter plot of height and weight with

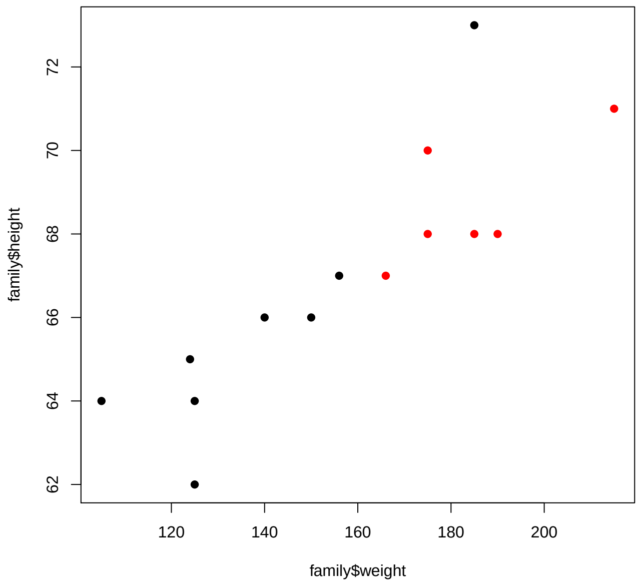

Figure 1.1: Family Heights and Weights. *This scatter plot shows the heights and weights of the individuals in the artificial family that we created for code demonstration. The red points correspond to the over weight family members.*

```
plot(y = family$height, x = family$weight,
     pch = 19, col = 1 + family$overWt)
```

Notice that we coerced `overWt` into a numeric vector with values 1 and 2 in order to use these values to specify colors for the points (the 2s are the red points). The resulting plot appears in Figure 1.1.

■

Data Frames

Data from a study or an experiment often consist of different types of measurements on a set of subjects of experimental units. For example, the World Bank provides summary statistics on countries around the world. Here the unit is the country and the measurements include gross domestic product, life expectancy, and literacy rate. These data are naturally organized into a table format. In *R*, the data frame is a data structure designed for the purpose of working with these kinds of data.

Rows

Each row in a data frame corresponds to a subject in the study, unit in an experiment, etc. The various measurements on a subject are in one row of the data frame.

Columns

The columns in a data frame correspond to variables. These can be different types. For example, a health study may contain height (numeric), sex (factor), education level (orded factor), and a unique identifier (character) for all subjects.

Working with Data Frames
The data frame facilitates many kinds of statistical analyses. For example, we can easily examine the relationship between income and sex using the formula `income ~ sex` in a call to the `plot()` function. And this same formula can be used to compare the average income levels for the two sexes via the `lm()` function. In both cases, the function performs a computation on the columns in a data frame. Depending on the data types, a different plot is created or a different model is fitted.

1.9 Subsets

A lot of what we do in statistics and exploratory data analysis is to examine subgroups of a sample or population. We determine characteristics about that subset and compare them to other groups or to the overall group. We might look at the age of over weight individuals and compare these values to the ages of all members of the group. Or, we may want to compare the BMI for men and women. These are all examples of how we look at different parts of our data using categorical or continuous variables to "zoom in" on a subgroup.

Being able to compute subgroups easily within our data is a very powerful and flexible feature of *R*, but takes some getting used to. There are essentially 5 different ways to compute a subgroup in *R*. They all use the `[` operator, and the only differences are what you specify as the value to use to identify the particular subset of interest.

1.9.1 Subsetting with Logical Vectors

One of the most intuitive approaches to subsetting uses a logical vector that is the same length as the vector from which we want to compute a subgroup. The `TRUE` values in this logical vector appear in the positions that correspond to the elements in the original vector that we want in our subset, and the `FALSE` values indicate those elements to be dropped. Figure 1.2 provides a diagram of the concept behind subsetting with a logical vector.

Example 1-9 Finding the Ages of the Over Weight Family Members

To find the ages of the over weight family members, we use the `foverWt` vector to identify the elements in `fage` that we want.

```
fage[foverWt]
```

```
[1] 77 33 67 59 27 55
```

If we are only after the average age of these individuals then we can pass the return vector of the 6 ages to the mean function with

```
mean(fage[foverWt])
```

```
[1] 53
```

On the other hand, if we want to perform additional calculations on this subgroup then we can assign the subset to a variable, e.g.,

```
ageOfOverWt = fage[foverWt]
```

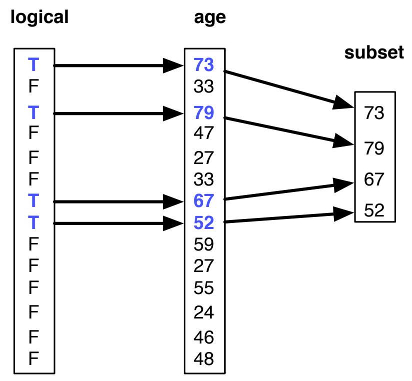

Figure 1.2: Subsetting with Logicals. *This diagram illustrates the concept of subsetting by logicals. The logical vector on the left has the same length as the vector* `age` *from which we want to compute a subgroup. The* `TRUE` *values in the logical vector identify the elements in* `age` *that we want to keep. The logical vector identifies the 1st, 3rd, 7th, and 8th elements in* `age` *for the subset. The resulting subset is on the righthand side of the diagram. Its length matches the number of* `TRUE` *values in the logical. (Note that for brevity we use shorthand of* `T` *for* `TRUE` *and* `F` *for* `FALSE`*.)*

Then we can continue to work with the subsetted ages that are now in `ageOfOverWt`. Reciprocally, to obtain a subset of those who are not over weight, we use the boolean not operator to turn the `TRUE` values into `FALSE` and vice versa. We do this with `fage[ !foverWt ]`.

■

Example 1-10 Finding the Heights of the Females in the Family

As another example, to examine the heights of the females in the family, we can create a logical vector that indicates which family members are female and use it to subset `fheight`. We use the relational operator `==` to create a vector of `TRUE`s for females and `FALSE` for males, with

```
fsex == "f"
 [1] FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE
[10] FALSE  TRUE FALSE FALSE  TRUE
```

Then we use this logical vector to index into `fheight` with

```
fheight[fsex == "f"]
[1] 64 61 68 65 67 62
```

■

The subset() function performs subsetting by logicals without the use of the `[` operator. It takes two inputs: the object to be subsetted and the logical expression that indicates the elements to keep. In other words, these two examples of computing a subset of the ages of over weight people in Q.1-9 (page 24) and the heights of the females in Q.1-10 (page 25) can be re-expressed using subset() as follows:

```
subset(fage, foverWt)
subset(fheight, fsex == "f")
```

This function can be very handy, but it does not support the other forms of subsetting, which we describe next, nor subsetting other data structures, such as lists.

1.9.2 Subsetting by Position

If we know that we want the BMI of the 10th person in the family, then we can use this position to specify the subset, i.e.,

```
fbmi[10]
```

```
[1] 30.05
```

More generally, we can provide a vector of positions to use in subsetting. These positions identify the elements in the original vector to appear in the subset. The order of the positions determines the order of the values in the return vector. If a position appears multiple times then the corresponding element appears multiple times in the subset. Figure 1.3 provides two conceptual diagrams of subsetting by position.

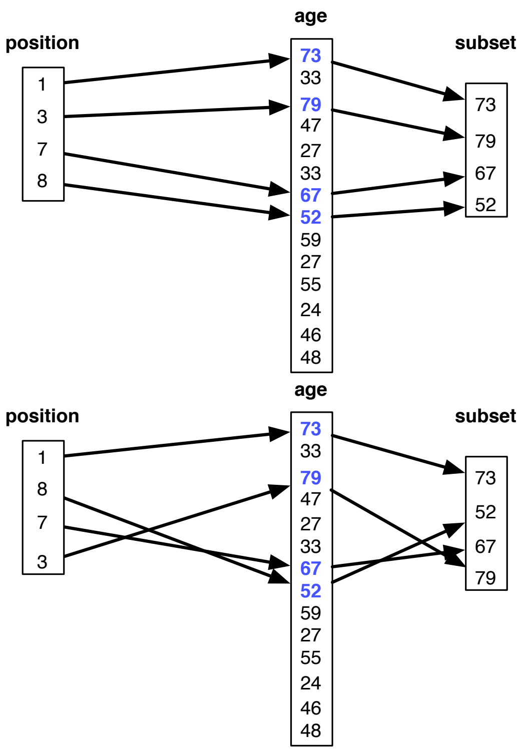

Figure 1.3: Subsetting by Position. *These two diagrams show how subsetting by position works. The vector of integers on the lefthand side of each diagram identifies the positions of the values in the `age` vector that we want in the subset. As in Figure 1.2, we have identified the 1st, 3rd, 7th, and 8th elements for our subset. Notice that the position vector in the bottom diagram has these same values except they appear in a different order and one value appears more than once. The order determines the ordering of the subset, e.g., the value 8 is the 2nd and 5th elements in the positions vector so the 8th element of `age` is placed in the 2nd and 5th elements of the subset.*

Example 1-11 Finding a Subset of BMI by Position

Suppose we want the 1st and 14th BMI values in our subset. We can create a vector with two elements, 1 and 14, with the expression `c(1, 14)`, where `c()` stands for concatenate (more on this function in Section 1.11.1). Then we compute our subset with

```
fbmi[ c(1, 14) ]
```

```
[1] 25.16 22.91
```

If we swap the order of 1 and 14 in our vector of positions, then the 14th BMI value in the original vector is the first value in the subset and the 1st BMI value in `fbmi` is the 2nd element of the subset. Furthermore, if we include 1 twice in the vector of positions, then the BMI for the first person appears twice in the subset. That is,

```
fbmi[ c(14, 1, 1) ]
```

```
[1] 22.91 25.16 25.16
```

This may seem a bit unusual, but it can be very useful, particularly for coloring points in plots (see Section 3.5).

■

What if we give a position that makes no sense, e.g., that is larger than the length of the starting vector? For example, we know there are only 14 members in our family so let's ask for the 20th element of one of the vectors. When we do this, we find, e.g.,

```
fage[20]
```

```
[1] NA
```

The result is a missing value, `NA`. This makes sense in many contexts. It is something we should be aware of so that we can understand how `NA`s might be introduced into our computations.

There are two other values that might be considered meaningless. What if we ask for the 0-th element of a vector? For example,

```
fage[0]
```

```
integer(0)
```

```
fage[c(0, 2)]
```

```
[1] 33
```

Essentially, *R* ignores a request for the 0-th element and doesn't include a value in the result for that element. This means that the result may not have as many elements as we asked for. That is, in our second expression above, we requested 2 elements of `fage` and were given only 1. Can you figure out what is the return value from `fage[ c(20, 0, 2) ]`? Try to reason it out from the previous examples. It makes sense that the return value has 2 elements, not 3, and these are `NA 33`, in that order.

Now, what if we ask for a negative index? That is subsetting by exclusion.

1.9.3 Subsetting by Exclusion

If the positions that we supply are negative numbers, then *R* drops those particular elements of the vector to create the subset (see Figure 1.4). For example, we drop the 1st through 3rd elements of `fage` to compute a subset from the 4th through 14th elements with

```
fage[ c(-1, -2, -3)]
```

```
 [1] 47 27 33 67 52 59 27 55 24 46 48
```

We cannot mix positive and negative indices in a single subsetting call. In other words, we cannot include some elements and omit others in one action. The following request is not valid:

```
fage[c(-1, -3, 5, 6, 7)]
```

It might seem reasonable to drop the 1st and 3rd elements and include the 5th, 6th and 7th. However, if we give such a command, we get an error,

```
fage[c(-1, -3, 5, 6, 7)]
```

```
Error in fage[c(-1, -3, 5, 6, 7)] :
  only 0's may be mixed with negative subscripts
```

This mix of positive and negative positions does not make sense because we are saying that we want to only include elements in positions 5, 6, and 7, but also we want to exclude those in positions 1 and 3. What about the 2nd, 4th, 8th, ..., 14th elements? Are they part of the inclusion or the exclusion? For this reason, *R* accepts either all nonnegative or nonpositive positions.

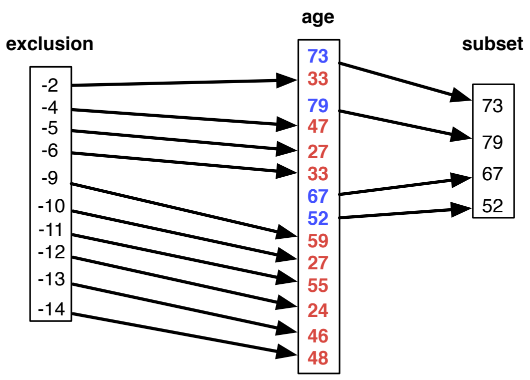

Figure 1.4: Subsetting by Exclusion. *This diagram demonstrates the idea of subsetting by exclusion. The vector of negative integers on the left identifies the positions of the values in* `age` *that we want excluded from the subset. According to these negative indices, all but the 1st, 3rd, 7th, and 8th elements are excluded from* `age` *to create the subset. This yields the same subset as in Figure 1.2 and in the top diagram in Figure 1.3.*

1.9.4 Subsetting by Name

We have seen that vector elements can have names. If we subset a vector where the elements are named, then we can refer to the elements we want in the subset using these names (see Figure 1.5). Recall that `fheight` has named elements, where the names are the letters `"a"` through `"n"`. We can compute a subset of the elements named `"c"` and `"a"`, in that order, with

```
fheight[ c("c", "a")]
```

```
 c  a
73 70
```

As with subsetting by position, if we ask for a non-existent element, then we get an `NA` in the result, e.g.,

```
fheight[ c("c", "z", "a")]
```

```
   c <NA>    a
  73   NA   70
```

Notice that in the resulting subset both the 2nd element and its name are `NA`.

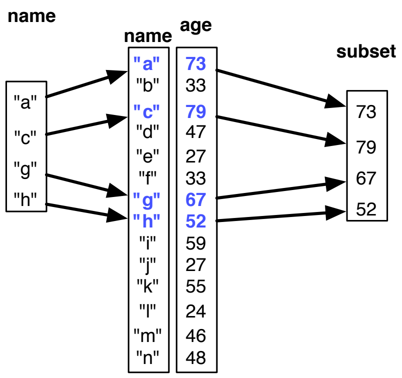

Figure 1.5: Subsetting by Name. *This diagram illustrates the idea of subsetting by name. The `age` vector in this diagram has names associated with the elements. The character vector on the lefthand side of the diagram contains the strings, `"a"`, `"c"`, `"g"`, and `"h"`. These identify the elements in `age` that we want in the subset. Note that the resulting vector is also a named vector, but we do not include these names for simplicity.*

We cannot use this style of subsetting to exclude elements. Think about what `fheight[ -c("c", "z", "a")]` means when *R* interprets the command. While we can understand that we mean to drop the elements named `"c"`, `"z"` and `"a"`, *R* first evaluates `-c("c", "z", "a")`. This is meaningless as the negative of a string doesn't make sense and can't be coerced into values that do make sense. The error *R* gives us comes from this part of the computation.

```
fheight[ -c("c", "z", "a")]
```

```
Error in -c("c", "z", "a") : invalid argument to unary operator
```

What's the unary operator? It is the `-` operator.

1.9.5 Subsetting All

The last kind of subsetting occurs when we pass no value for the indexing vector, e.g., `fage[ ]`. The result is the `fage` vector itself, i.e, all of the elements in their original order. This is not the same as passing in a vector with length 0, e.g., `fbmi[ integer(0) ]`. That returns a subset with the same length as the indexing vector, which is an empty vector,

```
fbmi[ integer(0) ]
```

```
numeric(0)
```

Why is this kind of subsetting useful? There are several reasons. Something that we have not mentioned about subsetting yet is that not only can we access sub-vectors using these 5 techniques, but we can also modify the contents of these sub-vectors in the original vector. We show how to do this next in Section 1.9.6. Another time when "subsetting all" is useful is when we work with data frames and multi-dimensional vectors, such as matrices. In these cases, we must provide both the row and column indices so subsetting all rows (or columns) can be quite useful. We provide examples of this in Section 1.9.7.

1.9.6 Assigning Values to Subsets

We can use the 5 subsetting techniques to assign new values to a subgroup of the elements in the original vector. In this case, we place the subsetting specification on the lefthand side of the equal sign and the new values on the righthand side. For example,

```
fweight[ 10 ] = NA
```

This assignment sets the weight of the 10th element in `fweight` to `NA`.

Example 1-12 Re-assigning Age Values

Suppose we made a mistake and erroneously switched the ages of the 1st and 10th individuals, i.e., the two Toms in the family have each other's age. Since we know that the 1st Tom is listed as 77 years old and the 2nd Tom as 27 we can swap these ages with the following assignment:

```
fage[ c(1, 10) ] = c(27, 77)
```

Alternatively, we don't need to type the ages. We can simply get the ages via subsetting and then assign them, i.e.,

```
fage[ c(1, 10) ] = fage[ c(10, 1) ]
fage
```

```
 [1] 27 33 79 47 27 33 67 52 59 77 55 24 46 48
```

An even more general approach that uses the names of the individuals and does not require us to know their positions in `fage` is to use subsetting with logicals on the lefthand side. We do this with

```
fage[fnames == "Tom"] = rev(fage[fnames == "Tom"])
```

Let's examine this assignment statement more closely. Breaking it down, we see the expression `fnames == "Tom"`, which returns a logical vector where the 1st and 10th elements are `TRUE` and the rest are `FALSE`. Then `fage[ fnames == "tom" ]` computes a subset of `fage` from its 1st and 10th values. Now the equal sign indicates that we are replacing