4. Basics of R

R is a powerful tool for all manner of calculations, data manipulation and scientific computations. Before getting to the complex operations possible in R we must start with the basics. Like most languages R has its share of mathematical capability, variables, functions and data types.

4.1 Basic Math

Being a statistical programming language, R can certainly be used to do basic math and that is where we will start.

We begin with the “hello, world!” of basic math: 1 + 1. In the console there is a right angle bracket (>) where code should be entered. Simply test R by running > 1 + 1

[1] 2

If this returns 2, then everything is great; if not, then something is very, very wrong.

Assuming it worked, let’s look at some slightly more complicated expressions: > 1 + 2 + 3

[1] 6

> 3 * 7 * 2

[1] 42

> 4 / 2

[1] 2

> 4 / 3

[1] 1.333333

These follow the basic order of operations: Parenthesis, Exponents, Multiplication, Division, Addition and Subtraction (PEMDAS). This means operations inside parentheses take priority over other operations. Next on the priority list is exponentiation. After that, multiplication and division are performed, followed by addition and subtraction.

This is why the first two lines in the following code have the same result, while the third is different.

```
> 4 * 6 + 5

[1] 29

> (4 * 6) + 5

[1] 29

> 4 * (6 + 5)

[1] 44
```

So far we have put white space in between each operator, such as `*` and `/`. This is not necessary but is encouraged as good coding practice.

4.2 Variables

Variables are an integral part of any programming language and R offers a great deal of flexibility. Unlike statically typed languages such as C++, R does not require variable types to be declared. A variable can take on any available data type as described in Section 4.3. It can also hold any R object such as a function, the result of an analysis or a plot. A single variable can at one point hold a number, then later hold a character and then later a number again.

4.2.1 Variable Assignment

There are a number of ways to assign a value to a variable, and again, this does not depend on the type of value being assigned.

The valid assignment operators are `<-` and `=`, with the first being preferred.

For example, let's save 2 to the variable *x* and 5 to the variable *y*.

```
> x <- 2
> x

[1] 2

> y = 5
> y

[1] 5
```

The arrow operator can also point in the other direction.

```
> 3 -> z
> z

[1] 3
```

The assignment operation can be used successively to assign a value to multiple variables simultaneously.

```
> a <- b <- 7
> a

[1] 7

> b

[1] 7
```

A more laborious, though sometimes necessary, way to assign variables is to use the **assign** function.

```
> assign("j", 4)
> j

[1] 4
```

Variable names can contain any combination of alphanumeric characters along with periods (.) and underscores (_). However, they cannot *start* with a number or an underscore.

The most common form of assignment in the R community is the left arrow (`<-`), which may seem awkward to use at first but eventually becomes second nature. It even seems to make sense, as the variable is sort of pointing to its value. There is also a particularly nice benefit for people coming from languages like SQL, where a single equal sign (=) tests for equality.

It is generally considered best practice to use actual names, usually nouns, for variables instead of single letters. This provides more information to the person reading the code. This is seen throughout this book.

4.2.2 Removing Variables

For various reasons a variable may need to be removed. This is easily done using **remove** or its shortcut **rm**.

Click here to view code image

```
> j

[1] 4

> rm(j)
> # now it is gone
> j

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

This frees up memory so that R can store more objects, although it does not necessarily free up memory for the operating system. To guarantee that, use **gc**, which performs garbage collection, releasing unused memory to the operating system. R automatically does garbage collection periodically, so this function is not essential.

Variable names are case sensitive, which can trip up people coming from a language like SQL or Visual Basic.

Click here to view code image

```
> theVariable <- 17
> theVariable

[1] 17

> THEVARIABLE

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

4.3 Data Types

There are numerous data types in R that store various kinds of data. The four main types of data most likely to be used are `numeric`, `character` (string), `Date/POSIXct` (time-based) and `logical` (TRUE/FALSE).

The type of data contained in a variable is checked with the **class** function.

```
> class(x)

[1] "numeric"
```

4.3.1 Numeric Data

As expected, `R` excels at running numbers, so numeric data is the most common type in `R`. The most commonly used numeric data is `numeric`. This is similar to a `float` or `double` in other languages. It handles integers and decimals, both positive and negative, and of course, zero. A numeric value stored in a variable is automatically assumed to be `numeric`. Testing whether a variable is `numeric` is done with the function **is.numeric**.

```
> is.numeric(x)

[1] TRUE
```

Another important, if less frequently used, type is `integer`. As the name implies this is for whole numbers only, no decimals. To set an integer to a variable it is necessary to append the value with an `L`. As with checking for a `numeric`, the **is.integer** function is used.

```
> i <- 5L
> i

[1] 5

> is.integer(i)

[1] TRUE
```

Do note that, even though `i` is an `integer`, it will also pass a `numeric` check.

```
> is.numeric(i)

[1] TRUE
```

`R` nicely promotes `integer`s to `numeric` when needed. This is obvious when multiplying an `integer` by a `numeric`, but importantly it works when dividing an `integer` by another `integer`, resulting in a decimal number.

```
> class (4L)

[1] "integer"

> class(2.8)

[1] "numeric"

> 4L * 2.8

[1] 11.2

> class(4L * 2.8)

[1] "numeric"

> class(5L)

[1] "integer"

> class(2L)

[1] "integer"

> 5L / 2L
```

```
[1] 2.5
```

```
> class (5L / 2L)
```

```
[1] "numeric"
```

4.3.2 Character Data

Even though it is not explicitly mathematical, the character (string) data type is very common in statistical analysis and must be handled with care. `R` has two primary ways of handling character data: `character` and `factor`. While they may seem similar on the surface, they are treated quite differently.

```
> x <- "data"
> x
```

```
[1] "data"
```

```
> y<- factor ("data")
> y
```

```
[1] data
Levels: data
```

Notice that `x` contains the word “data” encapsulated in quotes, while `y` has the word “data” without quotes and a second line of information about the `levels` of `y`. That is explained further in Section 4.4.2 about `vectors`.

Characters are case sensitive, so “Data” is different from “data” or “DATA”.

To find the length of a `character` (or `numeric`) use the **nchar** function.

```
> nchar(x)
```

```
[1] 4
```

```
> nchar("hello")
```

```
[1] 5
```

```
> nchar (3)
```

```
[1] 1
```

```
> nchar(452)
```

```
[1] 3
```

This will not work for `factor` data.

Click here to view code image

```
> nchar(y)
```

```
Error in nchar(y): 'nchar()' requires a character vector
```

4.3.3 Dates

Dealing with dates and times can be difficult in any language, and to further complicate matters R has numerous different types of dates. The most useful are `Date` and `POSIXct`. `Date` stores just a date while `POSIXct` stores a date and time. Both objects are actually represented as the number of days (`Date`) or seconds (`POSIXct`) since January 1, 1970.

Click here to view code image

```
> date1 <- as.Date("2012-06-28")
> date1

[1] "2012-06-28"

> class(date1)

[1] "Date"

> as.numeric(date1)

[1] 15519

> date2 <- as.POSIXct("2012-06-28 17:42")
> date2

[1] "2012-06-28 17:42:00 EDT"

> class(date2)

[1] "POSIXct" "POSIXt"

> as.numeric(date2)

[1] 1340919720
```

Easier manipulation of date and time objects can be accomplished using the **lubridate** and **chron** packages.

Using functions such as **as.numeric** or **as.Date** does not merely change the formatting of an object but actually changes the underlying type.

```
> class(date1)

[1] "Date"

> class(as.numeric(date1))

[1] "numeric"
```

4.3.4 Logical

`Logical`s are a way of representing data that can be either `TRUE` or `FALSE`. Numerically, `TRUE` is the same as 1 and `FALSE` is the same as 0. So `TRUE` ∗ 5 equals 5 while `FALSE` * 5 equals 0.

```
> TRUE * 5

[1] 5

> FALSE * 5

[1] 0
```

Similar to other types, `logical`s have their own test, using the **is.logical** function.

```
> k <- TRUE
> class(k)

[1] "logical"

> is.logical(k)

[1] TRUE
```

`R` provides `T` and `F` as shortcuts for `TRUE` and `FALSE`, respectively, but it is best practice not to use them, as they are simply variables storing the values `TRUE` and `FALSE` and can be overwritten, which can cause a great deal of frustration as seen in the following example.

```
> TRUE

[1] TRUE

> T

[1] TRUE

> class(T)

[1] "logical"

> T <- 7
> T

[1] 7

> class(T)

[1] "numeric"
```

`Logical`s can result from comparing two numbers, or characters.

Click here to view code image

```
> # does 2 equal 3?
> 2 == 3

[1] FALSE

> # does 2 not equal three?
> 2 != 3

[1] TRUE

> # is two less than three?
> 2 < 3

[1] TRUE

> # is two less than or equal to three?
> 2 <= 3

[1] TRUE

> # is two greater than three?
> 2 > 3
```

```
[1] FALSE

> # is two greater than or equal to three?
> 2 >= 3

[1] FALSE

> # is "data" equal to "stats"?
> "data" == "stats"

[1] FALSE

> # is "data" less than "stats"?
> "data" < "stats"

[1] TRUE
```

4.4 Vectors

A `vector` is a collection of elements, all of the same type. For instance, c(1, 3, 2, 1, 5) is a `vector` consisting of the numbers 1, 3, 2, 1, 5, in that order. Similarly, `c(“R”, “Excel”, “SAS”, “Excel”)` is a `vector` of the `character` elements, “R”, “Excel”, “SAS”, and “Excel”. A `vector` cannot be of mixed type.

`Vector`s play a crucial, and helpful, role in `R`. More than being simple containers, `vector`s in `R` are special in that `R` is a vectorized language. That means operations are applied to each element of the `vector` automatically, without the need to loop through the `vector`. This is a powerful concept that may seem foreign to people coming from other languages, but it is one of the greatest things about `R`.

`Vector`s do not have a dimension, meaning there is no such thing as a column vector or row `vector`. These `vector`s are not like the mathematical `vector`, where there is a difference between row and column orientation.[1]

[1]. Column or row `vector`s can be represented as one-dimensional `matrices`, which are discussed in Section 5.3.

The most common way to create a `vector` is with **c**. The “c” stands for combine because multiple elements are being combined into a `vector`.

Click here to view code image

```
> x <-c (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
> x

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

4.4.1 Vector Operations

Now that we have a `vector` of the first ten numbers, we might want to multiply each element by 3. In `R` this is a simple operation using just the multiplication operator (`*`).

Click here to view code image

```
> x * 3

[1] 3 6 9 12 15 18 21 24 27 30
```

No loops are necessary. Addition, subtraction and division are just as easy. This also works for any number of operations.

Click here to view code image

```
> x + 2
```

```
[1] 3 4 5 6 7 8 9 10 11 12

> x - 3

[1] -2 -1 0 1 2 3 4 5 6 7

> x / 4

[1] 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50

> x^2

[1] 1 4 9 16 25 36 49 64 81 100

> sqrt(x)

[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
[8] 2.828427 3.000000 3.162278
```

Earlier we created a `vector` of the first ten numbers using the **c** function, which creates a `vector`. A shortcut is the `:` operator, which generates a sequence of consecutive numbers, in either direction.

Click here to view code image

```
> 1 :10

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

> 10:1

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

> -2:3

[1] -2 -1 0 1 2 3

> 5:-7

[1] 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7
```

Vector operations can be extended even further. Let’s say we have two vectors of equal length. Each of the corresponding elements can be operated on together.

Click here to view code image

```
> # create two vectors of equal length
> x <- 1:10
> y <- -5:4
> # add them
> x + y

[1] -4 -2 0 2 4 6 8 10 12 14

> # subtract them
> x - y

[1] 6 6 6 6 6 6 6 6 6 6

> # multiply them
> x * y

[1] -5 -8 -9 -8 -5 0 7 16 27 40
```

```
> # divide them--notice division by 0 results in Inf
> x / y

[1] -0.2 -0.5 -1.0 -2.0 -5.0 Inf 7.0 4.0 3.0 2.5

> # raise one to the power of the other
> x^y

[1] 1.000000e+00 6.250000e-02 3.703704e-02 6.250000e-02 2.000000e-01
[6] 1.000000e+00 7.000000e+00 6.400000e+01 7.290000e+02 1.000000e+04

> # check the length of each
> length(x)

[1] 10

> length(y)

[1] 10

> # the length of them added together should be the same
> length(x + y)

[1] 10
```

In the preceding code block, notice the hash # symbol. This is used for comments. Anything following the hash, on the same line, will be commented out and not run.

Things get a little more complicated when operating on two `vector`s of unequal length. The shorter `vector` gets recycled—that is, its elements are repeated, in order, until they have been matched up with every element of the longer `vector`. If the longer one is not a multiple of the shorter one, a warning is given.

Click here to view code image

```
> x + c(1, 2)

[1] 2 4 4 6 6 8 8 10 10 12

> x + c(1, 2, 3)

Warning in x + c(1, 2, 3): longer object length is not a
multiple of shorter object length

[1] 2 4 6 5 7 9 8 10 12 11
```

Comparisons also work on `vector`s. Here the result is a vector of the same length containing `TRUE` or `FALSE` for each element.

Click here to view code image

```
> x <= 5

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

> x > y

[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

> x < y

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

To test whether all the resulting elements are `TRUE`, use the **all** function. Similarly, the **any** function checks whether any element is `TRUE`.

```
> x <- 10:1
> y <- -4:5
> any(x < y)

[1] TRUE

> all(x < y)

[1] FALSE
```

The **nchar** function also acts on each element of a `vector`.

Click here to view code image

```
> q <- c("Hockey", "Football", "Baseball", "Curling", "Rugby",
+ "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")
> nchar(q)

[1] 6 8 8 7 5 8 10 6 7 6

> nchar(y)

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

Accessing individual elements of a `vector` is done using square brackets (`[ ]`). The first element of `x` is retrieved by typing `x[1]`, the first two elements by `x[1:2]` and nonconsecutive elements by `x[c(1, 4)]`.

```
> x[1]

[1] 10

> x[1:2]

[1] 10 9

> x[c(1, 4)]

[1] 10 7
```

This works for all types of `vector`s whether they are `numeric`, `logical`, character and so forth.

It is possible to give names to a `vector` either during creation or after the fact.

Click here to view code image

```
> # provide a name for each element of an array using a name-value pair
> c(One="a", Two="y", Last="r")

One Two Last
"a" "y" "r"

> # create a vector
> w <- 1:3
> # name the elements
> names(w) <-c("a", "b", "c")
> w

a b c
1 2 3
```

4.4.2 Factor Vectors

Factors are an important concept in R, especially when building models. Let's create a simple vector of text data that has a few repeats. We will start with the vector q we created earlier and add some elements to it.

Click here to view code image

```
> q2 <-c(q, "Hockey", "Lacrosse", "Hockey", "Water Polo",
+ "Hockey", "Lacrosse") Converting this to a factor is easy with as.factor.
```

Click here to view code image

```
> q2Factor <-as.factor(q2)
> q2Factor

[1] Hockey Football Baseball Curling Rugby Lacrosse
[7] Basketball Tennis Cricket Soccer Hockey Lacrosse
[13] Hockey Water Polo Hockey Lacrosse
11 Levels: Baseball Basketball Cricket Curling Football ... Water Polo Notice that after
printing out every element of q2Factor, R also prints the levels of q2Factor. The levels
of a factor are the unique values of that factor variable. Technically, R is giving each
unique value of a factor a unique integer, tying it back to the character representation.
This can be seen with as.numeric.
```

Click here to view code image

```
> as.numeric(q2Factor)

[1] 6 5 1 4 8 7 2 10 3 9 6 7 6 11 6 7
```

In ordinary factors the order of the levels does not matter and one level is no different from another. Sometimes, however, it is important to understand the order of a factor, such as when coding education levels. Setting the ordered argument to TRUE creates an ordered factor with the order given in the levels argument.

Click here to view code image

```
> factor(x=c("High School", "College", "Masters", "Doctorate"),
+ levels=c("High School", "College", "Masters", "Doctorate"),
+ ordered=TRUE)

[1] High School College Masters Doctorate
Levels: High School < College < Masters < Doctorate Factors can drastically reduce the
size of the variable because they are storing only the unique values, but they can cause
headaches if not used properly. This will be discussed further throughout the book.
```

4.5 Calling Functions

Earlier we briefly used a few basic functions like **nchar**, **length** and **as.Date** to illustrate some concepts. Functions are very important and helpful in any language because they make code easily repeatable. Almost every step taken in R involves using functions, so it is best to learn the proper way to call them. R function calling is filled with a good deal of nuance, so we are going to focus on the gist of what is needed to know. Of course, throughout the book there will be many examples of calling functions.

Let's start with the simple **mean** function, which computes the average of a set of numbers. In its simplest form it takes a vector as an argument.

```
> mean(x)

[1] 5.5
```

More complicated functions have multiple arguments that can either be specified by the order they are entered or by using their name with an equal sign. We will see further use of this throughout the book.

R provides an easy way for users to build their own functions, which we will cover in more detail in Chapter 8.

4.6 Function Documentation

Any function provided in R has accompanying documentation, with varying quality, of course. The easiest way to access that documentation is to place a question mark in front of the function name, like this: ?mean.

To get help on binary operators like +, * or == surround them with back ticks (`).

```
> ?`+`
> ?`*`
> ?`==`
```

There are occasions when we have only a sense of the function we want to use. In that case we can look up the function by using part of the name with **apropos**.

Click here to view code image

```
> apropos("mea")

[1] ".colMeans" ".rowMeans" "colMeans"
[4] "influence.measures" "kmeans" "mean"
[7] "mean.Date" "mean.default" "mean.difftime"
[10] "mean.POSIXct" "mean.POSIXlt" "mean_cl_boot"
[13] "mean_cl_normal" "mean_sdl" "mean_se"
[16] "rowMeans" "weighted.mean"
```

4.7 Missing Data

Missing data plays a critical role in both statistics and computing, and R has two types of missing data, NA and NULL. While they are similar, they behave differently and that difference needs attention.

4.7.1 NA

Often we will have data that has missing values for any number of reasons. Statistical programs use various techniques to represent missing data such as a dash, a period or even the number 99. R uses NA. NA will often be seen as just another element of a vector. **is.na** tests each element of a vector for missingness.

Click here to view code image

```
> z <- c(1, 2, NA, 8, 3, NA, 3)
> z

[1] 1 2 NA 8 3 NA 3

> is.na(z)

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

NA is entered simply by typing the letters “N” and “A” as if they were normal text. This works for any kind of vector.

Click here to view code image

```
> zChar <- c("Hockey", NA, "Lacrosse")
> zChar

[1] "Hockey" NA "Lacrosse"

> is.na(zChar)

[1] FALSE TRUE FALSE
```

If we calculate the mean of z, the answer will be NA since **mean** returns NA if even a single element is NA.

```
> mean(z)

[1] NA
```

When the na.rm is TRUE, **mean** first removes the missing data, then calculates the mean.

```
> mean(z, na.rm=TRUE)

[1] 3.4
```

There is similar functionality with **sum**, **min**, **max**, **var**, **sd** and other functions as seen in Section 18.1.

Handling missing data is an important part of statistical analysis. There are many techniques depending on field and preference. One popular technique is multiple imputation, which is discussed in detail in Chapter 25 of Andrew Gelman and Jennifer Hill's book *Data Analysis Using Regression and Multilevel/Hierarchical Models*, and is implemented in the **mi**, **mice** and **Amelia** packages.

4.7.2 NULL

NULL is the absence of anything. It is not exactly missingness, it is nothingness. Functions can sometimes return NULL and their arguments can be NULL. An important difference between NA and NULL is that NULL is atomical and cannot exist within a vector. If used inside a vector, it simply disappears.

```
> z <- c(1, NULL,3)
> z

[1] 1 3
```

Even though it was entered into the vector z, it did not get stored in z. In fact, z is only two elements long.

The test for a NULL value is **is.null**.

```
> d <- NULL
> is.null(d)

[1] TRUE

> is.null(7)

[1] FALSE
```

Since NULL cannot be a part of a vector, **is.null** is appropriately not vectorized.

4.8 Pipes

A new paradigm for calling functions in R is the pipe. The pipe from the **magrittr** package works by taking the value or object on the left-hand side of the pipe and inserting it into the first argument of the function that is on the right-hand side of the pipe. A simple example example would be using a pipe to feed x to the **mean** function.

```
> library(magrittr)
> x <-1:10
> mean(x)

[1] 5.5

> x %>% mean

[1] 5.5
```

The result is the same but they are written differently. Pipes are most useful when used in a pipeline to chain together a series of function calls. Given a `vector z` that contains numbers and NAs, we want to find out how many NAs are present. Traditionally, this would be done by nesting functions.

Click here to view code image

```
> z <- c(1, 2, NA, 8, 3, NA, 3)
> sum(is.na(z))

[1] 2
```

This can also be done using pipes.

```
> z %>% is.na %>% sum

[1] 2
```

Pipes read more naturally in a left-to-right fashion, making the code easier to comprehend. Using pipes is negligibly slower than nesting function calls, though as Hadley Wickham notes, pipes will not be a major bottleneck in code.

When piping an object into a function and not setting any additional arguments, no parentheses are needed. However, if additional arguments are used, then they should be named and included inside the parentheses after the function call. The first argument is not used, as the pipe already inserted the left-hand object into the first argument.

```
> z %>% mean(na.rm=TRUE)

[1] 3.4
```

Pipes are used extensively in a number of modern packages after being popularized by Hadley Wickham in the **dplyr** package, as detailed in Chapter 14.

4.9 Conclusion

Data come in many types, and R is well equipped to handle them. In addition to basic calculations, R can handle numeric, character and time-based data. One of the nicer parts of working with R, although one that requires a different way of thinking about programming, is vectorization. This allows operating on multiple elements in a `vector` simultaneously, which leads to faster and more mathematical code.

5. Advanced Data Structures

Sometimes data require more complex storage than simple `vector`s and thankfully `R` provides a host of data structures. The most common are the `data.frame`, `matrix` and `list`, followed by the `array`. Of these, the `data.frame` will be most familiar to anyone who has used a spreadsheet, the `matrix` to people familiar with matrix math and the `list` to programmers.

5.1 `data.frame`s

Perhaps one of the most useful features of `R` is the `data.frame`. It is one of the most often cited reasons for `R`'s ease of use.

On the surface a `data.frame` is just like an Excel spreadsheet in that it has columns and rows. In statistical terms, each column is a variable and each row is an observation.

In terms of how `R` organizes `data.frame`s, each column is actually a `vector`, each of which has the same length. That is very important because it lets each column hold a different type of data (see Section 4.3). This also implies that within a column each element must be of the same type, just like with `vector`s.

There are numerous ways to construct a `data.frame`, the simplest being to use the **data.frame** function. Let's create a basic `data.frame` using some of the `vector`s we have already introduced, namely `x`, `y` and `q`.

Click here to view code image

```
> x <- 10:1
> y <- -4:5
> q <- c("Hockey", "Football", "Baseball", "Curling", "Rugby",
+ "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")
> theDF <- data.frame(x, y, q)
> theDF
x y q
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
6 5 1 Lacrosse
7 4 2 Basketball
8 3 3 Tennis
9 2 4 Cricket
10 1 5 Soccer
```

This creates a `10x3 data.frame` consisting of those three `vector`s. Notice the names of `theDF` are simply the variables. We could have assigned names during the creation process, which is generally a good idea.

Click here to view code image

```
> theDF <- data.frame(First=x, Second=y, Sport=q)
> theDF
First Second Sport
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
```

```
6 5 1 Lacrosse
7 4 2 Basketball
8 3 3 Tennis
9 2 4 Cricket
10 1 5 Soccer
```

`data.frame`s are complex objects with many attributes. The most frequently checked attributes are the number of rows and columns. Of course there are functions to do this for us: **nrow** and **ncol**. And in case both are wanted at the same time there is the **dim** function.

```
> nrow(theDF)

[1] 10

> ncol(theDF)

[1] 3

> dim(theDF)

[1] 10 3
```

Checking the column names of a `data.frame` is as simple as using the **names** function. This returns a `character vector` listing the columns. Since it is a `vector` we can access individual elements of it just like any other `vector`.

Click here to view code image

```
> names(theDF)

[1] "First" "Second" "Sport"

> names(theDF)[3]

[1] "Sport"
```

We can also check and assign the row names of a `data.frame`.

Click here to view code image

```
> rownames(theDF)

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

> rownames(theDF) <- c("One", "Two", "Three", "Four", "Five", "Six",
+ "Seven", "Eight", "Nine", "Ten")
> rownames(theDF)

[1] "One" "Two" "Three" "Four" "Five" "Six" "Seven" "Eight"
[9] "Nine" "Ten"

> # set them back to the generic index
> rownames(theDF) <- NULL
> rownames(theDF)

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

Usually a `data.frame` has far too many rows to print them all to the screen, so thankfully the **head** function prints out only the first few rows.

Click here to view code image

```
> head(theDF)
```

```
First Second Sport
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
6 5 1 Lacrosse
```

```
> head(theDF, n=7)
```

```
First Second Sport
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
6 5 1 Lacrosse
7 4 2 Basketball
```

```
> tail(theDF)
```

```
First Second Sport
5 6 0 Rugby
6 5 1 Lacrosse
7 4 2 Basketball
8 3 3 Tennis
9 2 4 Cricket
10 1 5 Soccer
```

As we can with other variables, we can check the `class` of a `data.frame` using the **class** function.

```
> class(theDF)
```

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

Since each column of the `data.frame` is an individual `vector`, it can be accessed individually and each has its own `class`. Like many other aspects of R, there are multiple ways to access an individual column. There is the $ operator and also the square brackets. Running `theDF$Sport` will give the third column in `theDF`. That allows us to specify one particular column by name.

Click here to view code image

```
> theDF$Sport
```

```
[1] Hockey Football Baseball Curling Rugby Lacrosse
[7] Basketball Tennis Cricket Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis Similar to vectors,
data.frames allow us to access individual elements by their position using square
brackets, but instead of having one position, two are specified. The first is the row
number and the second is the column number. So to get the third row from the second column
we use theDF[3, 2].
```

```
> theDF[3, 2]
```

```
[1] -2
```

To specify more than one row or column, use a `vector` of indices.

Click here to view code image

```
> # row 3, columns 2 through 3
> theDF[3, 2:3]
```

```
Second Sport
3 -2 Baseball

> # rows 3 and 5, column 2
> # since only one column was selected it was returned as a vector
> # hence the column names will not be printed
> theDF[c(3, 5), 2]

[1] -2 0

> # rows 3 and 5, columns 2 through 3
> theDF[c(3, 5), 2:3]

Second Sport
3 -2 Baseball
5 0 Rugby
```

To access an entire row, specify that row while not specifying any column. Likewise, to access an entire column, specify that column while not specifying any row.

Click here to view code image

```
> # all of column 3
> # since it is only one column a vector is returned
> theDF[, 3]

[1] Hockey Football Baseball Curling Rugby Lacrosse
[7] Basketball Tennis Cricket Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis

> # all of columns 2 through 3
> theDF[, 2:3]

Second Sport
1 -4 Hockey
2 -3 Football
3 -2 Baseball
4 -1 Curling
5 0 Rugby
6 1 Lacrosse
7 2 Basketball
8 3 Tennis
9 4 Cricket
10 5 Soccer

> # all of row 2
> theDF[2, ]

First Second Sport
2 9 -3 Football

> # all of rows 2 through 4
> theDF[2:4, ]

First Second Sport
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
```

To access multiple columns by name, make the column argument a character vector of the names.

Click here to view code image

```
> theDF[, c("First", "Sport")]
```

```
First Sport
1 10 Hockey
2 9 Football
3 8 Baseball
4 7 Curling
5 6 Rugby
6 5 Lacrosse
7 4 Basketball
8 3 Tennis
9 2 Cricket
10 1 Soccer
```

Yet another way to access a specific column is to use its column name (or its number) either as second argument to the square brackets or as the only argument to either single or double square brackets.

Click here to view code image

```
> # just the "Sport" column
> # since it is one column it returns as a (factor) vector
> theDF[, "Sport"]

[1] Hockey Football Baseball Curling Rugby Lacrosse
[7] Basketball Tennis Cricket Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis

> class(theDF[, "Sport"])

[1] "factor"

> # just the "Sport" column
> # this returns a one column data.frame
> theDF["Sport"]

Sport
1 Hockey
2 Football
3 Baseball
4 Curling
5 Rugby
6 Lacrosse
7 Basketball
8 Tennis
9 Cricket
10 Soccer

> class(theDF["Sport"])

[1] "data.frame"

> # just the "Sport" column
> # this also returns a (factor) vector
> theDF[["Sport"]]

[1] Hockey Football Baseball Curling Rugby Lacrosse
[7] Basketball Tennis Cricket Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis

> class(theDF[["Sport"]])

[1] "factor"
```

All of these methods have differing outputs. Some return a `vector`; some return a single-column data.frame. To ensure a single-column data.frame while using single square brackets, there is a third argument: drop=FALSE. This also works when specifying a single column by number.

Click here to view code image

```
> theDF[, "Sport", drop=FALSE]

Sport
1 Hockey
2 Football
3 Baseball
4 Curling
5 Rugby
6 Lacrosse
7 Basketball
8 Tennis
9 Cricket
10 Soccer

> class(theDF[, "Sport", drop=FALSE])

[1] "data.frame"

> theDF[, 3, drop=FALSE]

Sport
1 Hockey
2 Football
3 Baseball
4 Curling
5 Rugby
6 Lacrosse
7 Basketball
8 Tennis
9 Cricket
10 Soccer

> class(theDF[, 3, drop=FALSE])

[1] "data.frame"
```

In Section 4.4.2 we see that factors are stored specially. To see how they would be represented in data.frame, form use **model.matrix** to create a set of indicator (or dummy) variables. That is one column for each level of a factor, with a 1 if a row contains that level or a 0 otherwise.

Click here to view code image

```
> newFactor <-factor(c("Pennsylvania", "New York", "New Jersey",
+ "New York", "Tennessee", "Massachusetts",
+ "Pennsylvania", "New York"))
> model.matrix(~ newFactor - 1)
newFactorMassachusetts newFactorNew Jersey newFactorNew York
1 0 0 0
2 0 0 1
3 0 1 0
4 0 0 1
5 0 0 0
6 1 0 0
7 0 0 0
8 0 0 1

newFactorPennsylvania newFactorTennessee
1 1 0
```

```
2 0 0
3 0 0
4 0 0
5 0 1
6 0 0
7 1 0
8 0 0
attr(,"assign")
[1] 1 1 1 1 1
attr(,"contrasts")
attr(,"contrasts")$newFactor
[1] "contr.treatment"
```

We learn more about formulas (the argument to **model.matrix**) in Sections 11.2 and 14.3.2 and Chapters 18 and 19.

5.2 Lists

Often a container is needed to hold arbitrary objects of either the same type or varying types. R accomplishes this through `list`s. They store any number of items of any type. A `list` can contain all `numeric`s or `character`s or a mix of the two or `data.frame`s or, recursively, other `list`s.

`List`s are created with the **list** function where each argument to the function becomes an element of the `list`.

Click here to view code image

```
> # creates a three element list
> list(1, 2, 3)

[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

> # creates a single element list
> # the only element is a vector that has three elements
> list(c(1, 2, 3))

[[1]]
[1] 1 2 3

> # creates a two element list
> # the first is a three element vector
> # the second element is a five element vector
> (list3 <- list(c(1, 2, 3), 3:7))

[[1]]
[1] 1 2 3

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

> # two element list
> # first element is a data.frame
> # second element is a 10 element vector
> list(theDF, 1:10)
```

```
[[1]]
First Second Sport
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
6 5 1 Lacrosse
7 4 2 Basketball
8 3 3 Tennis
9 2 4 Cricket
10 1 5 Soccer

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

> # three element list
> # first is a data.frame
> # second is a vector
> # third is list3 which holds two vectors
> list5 <- list(theDF, 1:10, list3)
> list5

[[1]]
First Second Sport
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
6 5 1 Lacrosse
7 4 2 Basketball
8 3 3 Tennis
9 2 4 Cricket
10 1 5 Soccer

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

[[3]]
[[3]][[1]]
[1] 1 2 3

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

Notice in the previous block of code (where `list3` was created) that enclosing an expression in parenthesis displays the results after execution.

Like `data.frame`s, `list`s can have names. Each element has a unique name that can be either viewed or assigned using **names**.

Click here to view code image

```
> names(list5)

NULL

> names(list5) <-c("data.frame", "vector", "list")
> names(list5)

[1] "data.frame" "vector" "list"

> list5
```

```
$data.frame
First Second Sport
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
6 5 1 Lacrosse
7 4 2 Basketball
8 3 3 Tennis
9 2 4 Cricket
10 1 5 Soccer

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

$list
$list[[1]]
[1] 1 2 3

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

Names can also be assigned to `list` elements during creation using name-value pairs.

Click here to view code image

```
> list6 <- list(TheDataFrame=theDF, TheVector=1:10, TheList=list3)
> names(list6)

[1] "TheDataFrame" "TheVector" "TheList"

> list6

$TheDataFrame
First Second Sport
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
6 5 1 Lacrosse
7 4 2 Basketball
8 3 3 Tennis
9 2 4 Cricket
10 1 5 Soccer

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

$TheList
$TheList[[1]]
[1] 1 2 3

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

Creating an empty `list` of a certain size is, perhaps confusingly, done with **vector**.

Click here to view code image

```
> (emptyList <- vector(mode="list", length=4))

[[1]]
```

```
NULL

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
```

To access an individual element of a `list`, use double square brackets, specifying either the element number or name. Note that this allows access to only one element at a time.

Click here to view code image

```
> list5[[1]]

First Second Sport
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
6 5 1 Lacrosse
7 4 2 Basketball
8 3 3 Tennis
9 2 4 Cricket
10 1 5 Soccer

> list5[["data.frame"]]

First Second Sport
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
6 5 1 Lacrosse
7 4 2 Basketball
8 3 3 Tennis
9 2 4 Cricket
10 1 5 Soccer
```

Once an element is accessed it can be treated as if that actual element is being used, allowing nested indexing of elements.

Click here to view code image

```
> list5[[1]]$Sport

[1] Hockey Football Baseball Curling Rugby Lacrosse
[7] Basketball Tennis Cricket Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis

> list5[[1]][, "Second"]

[1] -4 -3 -2 -1 0 1 2 3 4 5

> list5[[1]][, "Second", drop=FALSE]

Second
1 -4
2 -3
```

```
3 -2
4 -1
5 0
6 1
7 2
8 3
9 4
10 5
```

It is possible to append elements to a `list` simply by using an index (either numeric or named) that does not exist.

Click here to view code image

```
> # see how long it currently is
> length(list5)

[1] 3

> # add a fourth element, unnamed
> list5[[4]] <- 2
> length(list5)

[1] 4

> # add a fifth element, name
> list5[["NewElement"]] <- 3:6
> length(list5)

[1] 5

> names(list5)

[1] "data.frame" "vector" "list" "" "NewElement"

> list5

$data.frame
First Second Sport
1 10 -4 Hockey
2 9 -3 Football
3 8 -2 Baseball
4 7 -1 Curling
5 6 0 Rugby
6 5 1 Lacrosse
7 4 2 Basketball
8 3 3 Tennis
9 2 4 Cricket
10 1 5 Soccer

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

$list
$list[[1]]
[1] 1 2 3

$list[[2]]
[1] 3 4 5 6 7

[[4]]
[1] 2

$NewElement
```

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

Occasionally appending to a `list`—or `vector` or `data.frame` for that matter—is fine, but doing so repeatedly is computationally expensive. So it is best to create a `list` as long as its final desired size and then fill it in using the appropriate indices.

5.3 Matrices

A very common mathematical structure that is essential to statistics is a `matrix`. This is similar to a `data.frame` in that it is rectangular with rows and columns except that every single element, regardless of column, must be the same type, most commonly all `numeric`s. They also act similarly to `vector`s with element-by-element addition, multiplication, subtraction, division and equality. The **nrow**, **ncol** and **dim** functions work just like they do for `data.frame`s.

Click here to view code image

```
> # create a 5x2 matrix
> A <- matrix(1:10, nrow=5)
> # create another 5x2 matrix
> B <- matrix(21:30, nrow=5)
> # create another 5x2 matrix
> C <- matrix(21:40, nrow=2)
> A

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

> B

[,1] [,2]
[1,] 21 26
[2,] 22 27
[3,] 23 28
[4,] 24 29
[5,] 25 30

> C

[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] 21 23 25 27 29 31 33 35 37 39
[2,] 22 24 26 28 30 32 34 36 38 40

> nrow(A)

[1] 5

> ncol(A)

[1] 2

> dim(A)

[1] 5 2

> # add them
> A + B
```

```
[,1] [,2]
[1,] 22 32
[2,] 24 34
[3,] 26 36
[4,] 28 38
[5,] 30 40

> # multiply them
> A * B

[,1] [,2]
[1,] 21 156
[2,] 44 189
[3,] 69 224
[4,] 96 261
[5,] 125 300

> # see if the elements are equal
> A == B

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

`Matrix` multiplication is a commonly used operation in mathematics, requiring the number of columns of the left-hand `matrix` to be the same as the number of rows of the right-hand `matrix`. Both `A` and `B` are 5*X*2 so we will transpose `B` so it can be used on the right-hand side.

Click here to view code image

```
> A %*% t(B)

[,1] [,2] [,3] [,4] [,5]
[1,] 177 184 191 198 205
[2,] 224 233 242 251 260
[3,] 271 282 293 304 315
[4,] 318 331 344 357 370
[5,] 365 380 395 410 425
```

Another similarity with `data.frame`s is that `matrices` can also have row and column names.

Click here to view code image

```
> colnames(A)

NULL

> rownames(A)

NULL

> colnames(A) <- c("Left", "Right")
> rownames(A) <- c("1st", "2nd", "3rd", "4th", "5th")
>
> colnames(B)

NULL

> rownames(B)

NULL
```

```
> colnames(B) <- c("First", "Second")
> rownames(B) <- c("One", "Two", "Three", "Four", "Five")
>
> colnames(C)

NULL

> rownames(C)

NULL

> colnames(C) <- LETTERS[1:10]
> rownames(C) <- c("Top", "Bottom") There are two special vectors, letters and LETTERS,
that contain the lower case and upper case letters, respectively.
```

Notice the effect when transposing a `matrix` and multiplying `matrices`. Transposing naturally flips the row and column names. `Matrix` multiplication keeps the row names from the left `matrix` and the column names from the right `matrix`.

Click here to view code image

```
> t(A)

1st 2nd 3rd 4th 5th
Left 1 2 3 4 5
Right 6 7 8 9 10

> A %*% C
A B C D E F G H I J
1st 153 167 181 195 209 223 237 251 265 279
2nd 196 214 232 250 268 286 304 322 340 358
3rd 239 261 283 305 327 349 371 393 415 437
4th 282 308 334 360 386 412 438 464 490 516
5th 325 355 385 415 445 475 505 535 565 595
```

5.4 Arrays

An `array` is essentially a multidimensional `vector`. It must all be of the same type, and individual elements are accessed in a similar fashion using square brackets. The first element is the row index, the second is the column index and the remaining elements are for outer dimensions.

Click here to view code image

```
> theArray <- array(1:12, dim=c(2, 3, 2))
> theArray

, , 1

[,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6

, , 2

[,1] [,2] [,3]
[1,] 7 9 11
[2,] 8 10 12

> theArray[1, , ]

[,1] [,2]
[1,] 1 7
```

```
[2,] 3 9
[3,] 5 11

> theArray[1, , 1]

[1] 1 3 5

> theArray[, , 1]

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

The main difference between an `array` and a `matrix` is that `matrices` are restricted to two dimensions, while `array`s can have an arbitrary number.

5.5 Conclusion

Data come in many types and structures, which can pose a problem for some analysis environments, but `R` handles them with aplomb. The most common data structure is the one-dimensional `vector`, which forms the basis of everything in `R`. The most powerful structure is the `data.frame`—something special in `R` that most other languages do not have—which handles mixed data types in a spreadsheet-like format. `List`s are useful for storing collections of items, like a hash in Perl.