**Chapter 2. Basic Objects**

The first step of learning R programming is getting familiar with basic R objects and their behavior. In this chapter, you will learn the following topics:

* Creating and subsetting atomic vectors (for example, numeric vectors, character vectors, and logical vectors), matrices, arrays, lists, and data frames.
* Defining and working with functions

*"Everything that exists is an object. Everything that happens is a function." -- John Chambers*

For example, in statistical analysis, we often feed a set of data to a linear regression model and obtain a group of linear coefficients.

Provided that there are different types of objects in R, when we do this, what basically happens in R is that we provide a data frame object that holds the set of data, carry it to the linear model function and get a list object consisting of the properties of the regression results, and finally extract a numeric vector, which is another type of object, from the list to represent the linear coefficients.

Every task involves various different types of objects. Each object has a different goal and behavior. It's important to understand how a basic object works in order to solve real-world problems, especially with more elegant code and fewer steps. More importantly, a more concrete understanding of object behavior allows you to spend more time on working out the solution to your problem than on getting stuck by countless minor problems while producing the right code.

In the following sections, we will see a variety of basic objects in R that represent different types of data and make it easy to analyze and visualize datasets. You will have a basic understanding of how these objects work and how they interact with each other.

**Vector**

A vector is a group of primitive values of the same type. It can be a group of numbers, true/false values, texts, and values of some other type. It is one of the building blocks of all R objects.

There are several types of vectors in R. They are distinct from each other in the type of elements they store. In the following sections, we will see the most commonly used types of vectors including numeric vectors, logical vectors, and character vectors.

**Numeric vector**

A numeric vector is a vector of numeric values. A scalar number is the simplest numeric vector. An example is shown as follows:

1.5

## [1] 1.5

A numeric vector is the most frequently used data type and is the foundation of nearly all kinds of data analysis. In other popular programming languages, there are some scalar types such as integer, double, and string, and these scalar types are the building blocks of the container types such as vectors. In R, however, there is no formal definition of scalar types. A scalar number is only a special case of numeric vector, and it's special only because its length is 1.

When we create a value, it is natural to think of how to store it for future use. To store the value, we can use <- to assign the value to a symbol. In other words, we create a variable named x of the value 1.5:

x <- 1.5

Then, the value is assigned to symbol x, and we can use x to represent the value from now on:

x

## [1] 1.5

There are multiple ways to create a numeric vector. We can call numeric() to create a zero vector of a given length:

numeric (10)

## [1] 0 0 0 0 0 0 0 0 0 0

We can also use c() to combine several vectors to make one vector. The simplest case is, for example, to combine several single-element vectors to be a multi-element vector:

c(1, 2, 3, 4, 5)

## [1] 1 2 3 4 5

We can also combine a mixture of single-element vectors and multi-element vectors and obtain a vector with the same elements as we previously created:

c(1, 2, c(3, 4, 5))

## [1] 1 2 3 4 5

To create a series of consecutive integers, the : operator will easily do the trick.

1:5

## [1] 1 2 3 4 5

Precisely speaking, the preceding code produces an integer vector instead of a numeric vector. In many cases, their difference is not that important. We will cover this topic later.

A more general way to produce a numeric sequence is seq(). For example, the following code produces a numeric vector of a sequence from 1 to 10 by increment 2:

seq(1, 10, 2)

## [1] 1 3 5 7 9

Functions like seq() have many arguments. We can call such a function by supplying all the arguments, but it is not necessary in most cases. Most functions provide reasonable default values for some arguments, which makes it easier for us to call them. In this case, we only need to specify the argument that we would like to modify from its default value.

For example, we can create another numeric vector that starts from 3 with the length 10 by specifying the length.out argument:

seq(3, length.out = 10)

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

A function call like the above uses a named argument length.out so that other arguments are kept default and only this argument is modified.

There are many ways in which we can define numeric vectors, but we should always be careful when we use :, an example is shown as follows:

1 + 1:5

## [1] 2 3 4 5 6

As the result shows, 1 + 1:5 does not mean a sequence from 2 to 5, but from 2 to 6. It is because :has higher priority than +, which results in evaluating 1:5 first and adding 1 to each entry, yielding the sequence you see in the result. We will cover the priority of operators later.

**Logical vector**

In contrast to numeric vectors, a logical vector stores a group of TRUE or FALSE values. They are basically yes or no to denote the answers to a group of logical questions.

The simplest logical vectors are TRUE and FALSE themselves:

TRUE

## [1] TRUE

A more usual way to obtain a logical vector is to ask logical questions about R objects. For example, we can ask R whether 1 is greater than 2:

1 > 2

## [1] FALSE

The answer is yes, represented by TRUE. Sometimes, it is verbose to write TRUE and FALSE; so, we can use T as an abbreviation for TRUE and F for FALSE. If we want to perform multiple comparisons at the same time, we can directly use numeric vectors in the question:

c(1, 2) > 2

## [1] FALSE FALSE

R interprets this expression as the element-wise comparison between c(1, 2) and 2. In other words, it is equivalent to c(1 > 2, 2 > 2).

We can compare two multi-element numeric vectors as long as the length of the longer vector is a multiple of that of the shorter one:

c(1, 2) > c(2, 1)

## [1] FALSE TRUE

The preceding code is equivalent to c(1 > 2, 2 > 1). To demonstrate how two vectors of different lengths are compared, see the following example::

c(2, 3) > c(1, 2, -1, 3)

## [1] TRUE TRUE TRUE FALSE

This may confuse you a bit. The computing mechanism recycles the shorter vector and works like c(2 > 1, 3 > 2, 2 > -1, 3 > 3). More specifically, the shorter vector will by recycled to finish all the comparisons for each element in the longer vector.

In R, several logical binary operators are defined, such as == to denote equality, > for greater-than, >= for greater-or-equals-to, < for less-than, and <= for less-than-or-equals-to. Moreover, R provides some other additional logical operators like %in% to tell whether each element in the left-hand side vector is contained by the right-hand side vector:

1 %in% c(1, 2, 3)

## [1] TRUE

c(1, 4) %in% c(1, 2, 3)

## [1] TRUE FALSE

You may notice that all the equality operators perform recycling but %in% does not. Instead, it always works by iterating over the vector on the left and works like c(1 %in% c(1, 2, 3), 4 %in% c(1, 2, 3)) in the preceding example.

**Character vector**

A character vector is a group of strings. Here, a character does not mean literally a single letter or symbol in a language, but it means a string like this is a string. Both double quotation marks and single quotation mark, can be used to create a character vector, as follows:

"hello, world!"

## [1] "hello, world!"

'hello, world!'

## [1] "hello, world!"

We can also use the combine function c() to construct a multi-element character vector:

c("Hello", "World")

## [1] "Hello" "World"

We can use == to tell whether two vectors have equal values in corresponding positions; this applies to character vectors too:

c("Hello", "World") == c('Hello', 'World')

## [1] TRUE TRUE

The character vectors are equal because " and ' both work to create a string and do not affect its value:

c("Hello", "World") == "Hello, World"

## [1] FALSE FALSE

The previous expression yields both FALSE because neither Hello nor World  equals Hello, World. The only difference between the two quotation marks is the behavior when you create a string containing quotation marks.

If you use " to create a string (a single-element character vector) containing " itself, you need to type "to escape " inside the string to prevent the interpreter from regarding " in the string as the close quotation mark of the string.

The following examples demonstrate the escaping of quotation marks. The code uses cat() to print the given text:

cat("Is "You" a Chinese name?")

## Is "You" a Chinese name?

If you feel that this is not easy to read, you may well use ' to create the string, which can be easier:

cat('Is "You" a Chinese name?')

## Is "You" a Chinese name?

In other words, " allows ' in the string without escaping, and ' allows " in the string without escaping.

Now we know the basic things about creating numeric vectors, logical vectors, and character vectors. In fact, we also have complex vectors and raw vectors in R. Complex vectors are vectors of complex values, such as c(1 + 2i, 2 + 3i). Raw vectors basically store raw binary data that is represented in the hexadecimal form. These two types of vectors are much less frequently used, but they share many behaviors with the three types of vectors we have covered.

In the next section, you will learn several ways to access part of a vector. By subsetting vectors, you should begin to understand how different types of vectors can be related to each other.

**Subsetting vectors**

If we want to access some specific entries or a subset of a vector, subsetting a vector means accessing some specific entries or a subset of the vector. In this section, we'll demonstrate various ways to subset a vector.

First, we create a simple numeric vector and assign it to v1:

v1 <- c(1, 2, 3, 4)

Each of the following lines gets a specific subset of v1.

For example, we can get the second element:

v1[2]

## [1] 2

We can get the second to fourth elements:

v1[2:4]

## [1] 2 3 4

We can get all elements except the third one:

v1[-3]

## [1] 1 2 4

The patterns are clear—we can put any numeric vector inside the square brackets after the vector to extract a corresponding subset:

a <- c(1, 3)v1[a]

## [1] 1 3

All the preceding examples perform subsetting by position, that is, we get a subset of a vector by specifying the positions of elements. Using negative numbers will exclude those elements. One thing to notice is that you can't use positive numbers and negative numbers together:

v1[c(1, 2, -3)]

## Error in v1[c(1, 2, -3)]: only 0's may be mixed with negative subscripts

What if we subset the vector using positions beyond the range of the vector? The following example tries to get a subset of v1 from the third element to the nonexisting sixth element:

v1[3:6]

## [1] 3 4 NA NA

As we can see, the nonexisting positions end up in missing values represented by NA. In real-world data, missing values are common. The good part is that all arithmetic calculations with NA also result in NA for consistency. On the other hand, however, it takes extra effort to deal with data because it may not be safe to assume that the data contains no missing values.

Another way to subset a vector is using logical vectors. We can supply an equal-length logical vector to determine whether each entry should be extracted:

v1[c(TRUE, FALSE, TRUE, FALSE)]

## [1] 1 3

More than subsetting, we can overwrite a specific subset of a vector like this:

v1[2] <- 0

In this case, v1 becomes the following:

v1

## [1] 1 0 3 4

We can also overwrite multiple elements at different positions at the same time:

v1[2:4] <- c(0, 1, 3)

Now, v1 becomes the following:

v1

## [1] 1 0 1 3

Like subsetting, logical selectors are also accepted for overwriting:

v1[c(TRUE, FALSE, TRUE, FALSE)] <- c(3, 2)

As you may expect, v1 becomes the following:

v1

## [1] 3 0 2 3

A useful implication of this operation is selecting entries by logical criterion. For example, the following code picks out all elements that are not greater than 2 in v1:

v1[v1 <= 2]

## [1] 0 2

A more complex selection criterion also works. The following example picks out all elements of v1 that satisfy *x2- x + 1 > 0* :

v1[v1 ^ 2 - v1 + 1 >= 0]

## [1] 3 0 2 3

To replace all entries that satisfy x <= 2 with 0, we can call the following:

v1[v1 <= 2] <- 0

As you may expect, v1 becomes the following:

v1

## [1] 3 0 0 3

If we overwrite the vector at a nonexisting entry, the vector will automatically expand with the unassigned value being NA as missing values:

v1[10] <- 8

v1

## [1] 3 0 0 3 NA NA NA NA NA 8

**Named vectors**

A named vector is not a specific type of vector parallel to a numeric or logical vector. It is a vector with names corresponding to the elements. We can give names to a vector when we create it:

x <- c(a = 1, b = 2, c = 3)

x

## a b c

## 1 2 3

Then, we can access the elements with a single-valued character vector:

x["a"]

## a

## 1

We can also get multiple elements with a character vector:

x[c("a", "c")]

## a c

## 1 3

If the character vector has duplicate elements, the selection will result in selecting duplicate elements:

x[c("a", "a", "c")]

## a a c

## 1 1 3

In addition to this, all other operations to a vector also perfectly work for named vectors.

We can get the names of a vector with names():

names(x)

## [1] "a" "b" "c"

The names of a vector are not fixed. We can change the names of a vector by assigning another character vector to its names.

names(x) <- c("x", "y", "z")

x["z"]

## z

## 3

If the names are no longer needed, we can simply remove the vector's names using NULL, a special object that represents undefined value:

names(x) <- NULL

x

## [1] 1 2 3

You may wonder what happens when the name does not exist at all. Let's experiment with the original x value:

x <- c(a = 1, b = 2, c = 3)

x["d"]

## <NA>

## NA

By intuition, accessing a nonexisting element should produce an error. However, the result is not an error but a vector of a single missing value with a missing name:

names(x["d"])

## [1] NA

If you provide a character vector in which some names exist but others do not, the resulting vector will preserve the length of the selection vector:

x[c("a", "d")]

## a <NA>

## 1 NA

**Extracting an element**

While [] creates a subset of a vector, [[]] extracts an element from a vector. A vector is like ten boxes of candy, [] gets you three boxes of candy, but [[]] opens a box and gets you a candy from it.

For simple vectors, using [] and [[]] to get one element will produce the same result. However, in some cases, they have different behaviors. For example, subsetting a named vector using one entry and extracting an element from it will produce different results:

x <- c(a =  1, b = 2, c = 3)

x["a"]

## a

## 1

x[["a"]]

## [1] 1

The metaphor of candy boxes makes it easier to understand. The x["a"] argument gives you the box of candy labeled "a", while x[["a"]] gives you the candy in the box labeled "a".

Since [[]] only extracts one element, it does not work with vectors of more than one element:

x[[c(1, 2)]]

## Error in x[[c(1, 2)]]: attempt to select more than one element

Also, it does not work with negative integers meaning excluding elements at certain positions:

x[[-1]]

## Error in x[[-1]]: attempt to select more than one element

We already know that subsetting a vector with a nonexisting position or name will produce missing values. However, [[]] simply does not work when we extract an element with a position beyond the range, nor does it work with a nonexisting name:

x[["d"]]

## Error in x[["d"]]: subscript out of bounds

For many beginners, it may be confusing to see both [[]] and [] used in the code and it is easy to misuse them. Just remember the metaphor of the candy boxes.

**Telling the class of vectors**

Sometimes we need to tell which kind of vector we are dealing with before taking an action. The class() function tells us the class of any R object:

class(c(1, 2, 3))

## [1] "numeric"

class(c(TRUE, TRUE, FALSE))

## [1] "logical"

class(c("Hello", "World"))

## [1] "character"

If we need to ensure that an object is indeed a vector of a specific class, we can use is.numeric, is.logical, is.character, and some other functions with similar names:

is.numeric(c(1, 2, 3))

## [1] TRUE

is.numeric(c(TRUE, TRUE, FALSE))

## [1] FALSE

is.numeric(c("Hello", "World"))

## [1] FALSE

**Converting vectors**

Different classes of vectors can be coerced to a specific class of vector. For example, some data are string representation of numbers, such as 1 and 20. If we leave these strings as they are, we won't be able to perform numeric calculations with them. Fortunately, these two strings can be converted to numeric vectors. This will make R regard them as numbers rather than strings so that we can do the math with them.

To demonstrate a typical conversion, we first create a character vector:

strings <- c("1", "2", "3")

class(strings)

## [1] "character"

As I mentioned, strings cannot be used to do maths directly:

strings + 10

## Error in strings + 10: non-numeric argument to binary operator

We can use as.numeric() to convert the character vector to a numeric vector:

numbers <- as.numeric(strings)

numbers

## [1] 1 2 3

class(numbers)

## [1] "numeric"

Now we can do maths with numbers:

numbers + 10

## [1] 11 12 13

Similar to is.\* functions (for example, is.numeric, is.logical, and is.character) that check the class of a given object, we can use the as.\* function family to convert a vector from its original class to another:

as.numeric(c("1", "2", "3", "a"))

## Warning: NAs introduced by coercion

## [1] 1 2 3 NA

as.logical(c(-1, 0, 1, 2))

## [1] TRUE FALSE TRUE TRUE

as.character(c(1, 2, 3))

## [1] "1" "2" "3"

as.character(c(TRUE, FALSE))

## [1] "TRUE" "FALSE"

It seems that each type of vector can be somehow converted to all other types. However, the conversion follows a set of rules.

The first line in the preceding block of code attempts to convert the character vector to a numeric vector, just as we did in the previous example. Obviously, the last element a cannot be converted to a number. The conversion is done except for the last element, so a missing value is produced instead.

As for converting a numeric vector to a logical vector, the rule is that only 0 corresponds to FALSE and all non-zero numbers will produce TRUE.

Each kind of vector can be converted to a character vector since everything has a character representation. However, if a numeric vector or a logical vector is coerced to a character vector, it cannot be directly involved in the arithmetic operations with other numeric or logical vectors unless it is converted back. That is why the following code does not work, as I have just mentioned:

c(2, 3) + as.character(c(1, 2))

## Error in c(2, 3) + as.character(c(1, 2)): non-numeric argument to binary operator

From the preceding examples, I have stressed that although R does not impose strong typing rules, it does not mean that R is smart enough to do exactly what you want it to do automatically. In most cases, it is better to ensure that the type of vectors are correct in computations; otherwise, an unexpected error will occur. In other words, only when you get the right type of data objects can you do the right math.

**Arithmetic operators for numeric vectors**

The arithmetic operations of numeric vectors are very simple. They basically follow two rules: Computing in an element-wise manner and recycling the shorter vector. The following examples demonstrate the behavior of the operators working with numeric vectors:

c(1, 2, 3, 4) + 2

## [1] 3 4 5 6

c(1, 2, 3) - c(2, 3, 4)

## [1] -1 -1 -1

c(1, 2, 3) \* c(2, 3, 4)

## [1] 2 6 12

c(1, 2, 3) / c(2, 3, 4)

## [1] 0.5000000 0.6666667 0.7500000

c(1, 2, 3) ^ 2

## [1] 1 4 9

c(1, 2, 3) ^ c(2, 3, 4)

## [1] 1 8 81

c(1, 2, 3, 14) %% 2

## [1] 1 0 1 0

Although vectors can have names, the operations do not function with corresponding names. Only the names of vectors on the left-hand side will remain and the names of those on the right-hand side will be ignored:

c(a = 1, b = 2, c = 3) + c(b = 2, c = 3, d = 4)

## a b c

## 3 5 7

c(a = 1, b = 2, 3) + c(b = 2, c = 3, d = 4)

## a b

## 3 5 7

We saw some basic behaviors of numeric vectors, logical vectors, and character vectors. They are the most commonly used data structures and are the building blocks of a wide variety of other useful objects. One of them is matrix, which is intensively used in the formulation of statistical and econometric theories, and it is very useful in representing two-dimensional data and solving linear systems. In the next chapter, we will see how we can create a matrix in R and how it is deeply rooted in vectors.

**Matrix**

A matrix is a vector represented and accessible in two dimensions. Therefore, what applies to vectors is most likely to apply to a matrix. For example, each type of vector (for example, numeric vector or logical vectors) has its matrix version, that is, there are numeric matrices, logical matrices, and so on.

**Creating a matrix**

We can call matrix() to create a matrix from a vector by setting up one of its two dimensions:

matrix(c(1, 2, 3, 2, 3, 4, 3, 4, 5), ncol = 3)

## [,1] [,2] [,3]

## [1,] 1 2 3

## [2,] 2 3 4

## [3,] 3 4 5

By specifying ncol = 3, we mean that the provided vector should be regarded as a matrix with 3 columns (and 3 rows automatically). You may feel the original vector is not as straightforward as its representation. To make the code more user-friendly, we can write the vector in multiple lines:

matrix(c(1, 2, 3,  4, 5, 6,  7, 8, 9), nrow = 3, byrow = FALSE)

## [,1] [,2] [,3]

## [1,] 1 4 7

## [2,] 2 5 8

## [3,] 3 6 9

matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = TRUE)

## [,1] [,2] [,3]

## [1,] 1 2 3

## [2,] 4 5 6

## [3,] 7 8 9

Often, we may need to create a diagonal matrix. Here, diag() is the most handy way to do this:

diag(1, nrow = 5)

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

## [1,] 1 0 0 0 0

## [2,] 0 1 0 0 0

## [3,] 0 0 1 0 0

## [4,] 0 0 0 1 0

## [5,] 0 0 0 0 1

**Naming rows and columns**

By default, creating a matrix does not automatically give names to its rows and columns. Sometimes, it is useful and straightforward to do so when different rows and columns have different meanings. We can give row names and/or column names when creating the matrix:

matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = TRUE, dimnames = list(c("r1", "r2", "r3"), c("c1", "c2", "c3")))

## c1 c2 c3

## r1 1 2 3

## r2 4 5 6

## r3 7 8 9

Alternatively, we can use row names and/or column names after the matrix is created:

m1 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)

rownames(m1) <- c("r1", "r2", "r3")

colnames(m1) <- c("c1", "c2", "c3")

Here, we encounter two new things: a list and a type of function, such as rownames(x) <-. We will discuss them later in this chapter.

**Subsetting a matrix**

Just as we deal with vectors, we need not only create matrices but also extract data from a matrix. This is called **matrix subsetting**.

Note that a matrix is a vector that is represented and accessible in two dimensions; we not only view a matrix in two dimensions, but also access it with a two-dimensional accessor [,], which is quite similar to the one-dimensional accessor, [] , for subsetting vectors.

To use it, we can supply two vectors for each dimension to determine a subset of a matrix. The first argument in the square bracket is the row selector, and the second is the column selector. As we tried in subsetting vectors, we can use numeric vectors, logical vectors, and character vectors in the two dimensions.

The following code demonstrates the various ways to subset the following matrix:

m1

## c1 c2 c3

## r1 1 4 7

## r2 2 5 8

## r3 3 6 9

We can extract only one element in the first row and the second column:

m1[1, 2]

## [1] 4

We can subset it with a range of positions:

m1[1:2, 2:3]

## c2 c3

## r1 4 7

## r2 5 8

If one dimension is left blank, all the values in that dimension will be selected:

m1[1,]

## c1 c2 c3

## 1 4 7

m1[,2]

## r1 r2 r3

## 4 5 6

m1[1:2,]

## c1 c2 c3

## r1 1 4 7

## r2 2 5 8

m1[, 2:3]

## c2 c3

## r1 4 7

## r2 5 8

## r3 6 9

Negative numbers exclude positions in a subsetting matrix, which is exactly the same as working with vectors:

m1[-1,]

## c1 c2 c3

## r2 2 5 8

## r3 3 6 9

m1[,-2]

## c1 c3

## r1 1 7

## r2 2 8

## r3 3 9

Note that the matrix has row names and column names, and we can use character vectors to subset it:

m1[c("r1", "r3"), c("c1", "c3")]

## c1 c3

## r1 1 7

## r3 3 9

Note again that a matrix is a vector represented and accessible in two dimensions; however, it is still a vector in its nature. This allows us to use a one-dimensional accessor for vectors to subset a matrix:

m1[1]

## [1] 1

m1[9]

## [1] 9

m1[3:7]

## [1] 3 4 5 6 7

Since a vector only contains entries of the same type, so does a matrix. Therefore, their operations are quite similar. If you type an inequality, it will return another logical matrix of equal size:

m1 > 3

## c1 c2 c3

## r1 FALSE TRUE TRUE

## r2 FALSE TRUE TRUE

## r3 FALSE TRUE TRUE

We can use an equal-sized logical matrix for subsetting as if it is a vector:

m1[m1 > 3]

## [1] 4 5 6 7 8 9

**Using matrix operators**

All arithmetic operators for vectors also work with matrices as if they were vectors. These operators perform calculations element-wise, except for matrix-only operators, such as matrix product,  %\*%:

m1 + m1

## c1 c2 c3

## r1 2 8 14

## r2 4 10 16

## r3 6 12 18

m1 - 2 \* m1

## c1 c2 c3

## r1 -1 -4 -7

## r2 -2 -5 -8

## r3 -3 -6 -9

m1 \* m1

## c1 c2 c3

## r1 1 16 49

## r2 4 25 64

## r3 9 36 81

m1 / m1

## c1 c2 c3

## r1 1 1 1

## r2 1 1 1

## r3 1 1 1

m1 ^ 2

## c1 c2 c3

## r1 1 16 49

## r2 4 25 64

## r3 9 36 81

m1 %\*% m1

## c1 c2 c3

## r1 30 66 102

## r2 36 81 126

## r3 42 96 150

We can also transpose a matrix using t():

t(m1)

## r1 r2 r3

## c1 1 2 3

## c2 4 5 6

## c3 7 8 9

Vectors and matrices are sufficient for many use cases. However, some particular problems need a data structure with even higher dimensions. In this next section, we will briefly introduce arrays and you will see how these data structures share similar behaviors.

**Array**

An array is a natural extension to a matrix in its number of dimensions. More specifically, an array is a vector that is represented and accessible in a given number of dimensions (mostly more than two dimensions).

If you are already familiar with vectors and matrices, you won't be surprised to see how arrays behave.

**Creating an array**

To create an array, we call array() by supplying a vector of data, how this data is arranged in different dimensions, and sometimes the names of the rows and columns of these dimensions.

Suppose we have some data (10 integers from 0 to 9) and we need to arrange them in three dimensions: 1 for the first dimension, 5 for the second, and 2 for the third:

a1 <- array(c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9), dim = c(1, 5, 2))

a1

## , , 1

##

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

## [1,] 0 1 2 3 4

##

## , , 2

##

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

## [1,] 5 6 7 8 9

We can clearly see how we can access these entries by looking at the notations around them.

Moreover, we can add names for these dimensions when we create the array:

a1 <- array(c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9), dim = c(1, 5, 2), dimnames = list(c("r1"), c("c1", "c2", "c3", "c4", "c5"), c("k1", "k2")))

a1

## , , k1

##

## c1 c2 c3 c4 c5

## r1 0 1 2 3 4

##

## , , k2

##

## c1 c2 c3 c4 c5

## r1 5 6 7 8 9

Alternatively, for an array that is already created, we can call dimnames(x) <- to setup the names for each dimension by supplying a list of several character vectors:

a0 <- array(c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10), dim = c(1, 5, 2))

dimnames(a0) <- list(c("r1"), c("c1", "c2", "c3", "c4", "c5"), c("k1", "k2"))

a0

## , , k1

##

## c1 c2 c3 c4 c5

## r1 0 1 2 3 4

##

## , , k2

##

## c1 c2 c3 c4 c5

## r1 5 6 7 8 9

**Subsetting an array**

The principle of subsetting an array is exactly the same as subsetting a matrix. Here, we can supply a vector for each dimension to extract a subset of an array:

a1[1,,]

## k1 k2

## c1 0 5

## c2 1 6

## c3 2 7

## c4 3 8

## c5 4 9

a1[, 2,]

## k1 k2

## 1 6

a1[,,1]

## c1 c2 c3 c4 c5

## 0 1 2 3 4

a1[1, 1, 1]

## [1] 0

a1[1, 2:4, 1:2]

## k1 k2

## c2 1 6

## c3 2 7

## c4 3 8

a1[c("r1"), c("c1", "c3"), "k1"]

## c1 c3

## 0 2

As you may notice, atomic vectors, matrices, and arrays share almost the same set of behaviors. A fundamental common feature they share is that they are all **homogeneous data types**, that is, the type of elements they store must be the same. However, there are also **heterogeneous data types** in R, that is, they can store different types of elements, which makes them much more flexible but they are less memory efficient and slower to operate.

**Lists**

A list is a generic vector that is allowed to include different types of objects, even other lists.

It is useful for its flexibility. For example, the result of a linear model fit in R is basically a list object that contains rich results of a linear regression such as linear coefficients (numeric vectors), residuals (numeric vectors), QR decomposition (a list containing a matrix and other objects), and so on.

It is very handy to extract the information without calling different functions each time because these results are all packed into a list.

**Creating a list**

We can use list() to create a list, as the function name suggests. Different types of objects can be put into one list. For example, the following code creates a list that contains a single-element numeric vector, a two-entry logical vector, and a character vector of three values:

l0 <- list(1, c(TRUE, FALSE), c("a", "b", "c"))

l0

## [[1]]

## [1] 1

##

## [[2]]

## [1] TRUE FALSE

##

## [[3]]

## [1] "a" "b" "c"

We can assign names to each list entry using named arguments:

l1 <- list(x = 1, y = c(TRUE, FALSE), z = c("a", "b", "c"))

l1

## $x

## [1] 1

##

## $y

## [1] TRUE FALSE

##

## $z

## [1] "a" "b" "c"

**Extracting an element from a list**

There are various ways to access the elements of a list. The most common way is to use a dollar-sign $ to extract the value of a list element by name:

l1 <- list(x = 1, y = c(TRUE, FALSE), z = c("a", "b", "c"), m = NULL)

l1$x

## [1] 1

l1$y

## [1] TRUE FALSE

l1$z

## [1] "a" "b" "c"

l1$m

## NULL

Note that if we ask for a non-existing element m, NULL will be returned.

Alternatively, we can supply a number in double square brackets to extract the value of the nth list member. For example, we can extract the value of the second member of list l1, as follows:

l1[[2]]

## [1] TRUE FALSE

With the same notation, we can also supply a name to extract the value of the list member with that name, just like using a dollar sign:

l1[["y"]]

## [1] TRUE FALSE

It can be more flexible to use double square brackets for value extraction from a list because, sometimes, we might not know which member we need to extract before a computation:

member <- "z" # you can dynamically determine which member to extract

l1[[member]]

## [1] "a" "b" "c"

Here, we supply a runtime-evaluated, single-element character vector to the brackets. But why should we use double brackets here? Where are the single brackets?

**Subsetting a list**

In many cases, we need to extract multiple elements from a list. These multiple members also construct a list as a subset of the original list.

To subset a list, we can use single-square-bracket notation, just like what we use for vectors and matrices. We can extract some elements of a list and put them into a new list.

The notation is very much consistent with how it works for vectors. We can extract elements from a list by name using a character vector, or by position using a numeric vector, or by criterion using a logical vector:

l1["x"]

## $x

## [1] 1

l1[c("x", "y")]

## $x

## [1] 1

##

## $y

## [1] TRUE FALSE

l1[1]

## $x

## [1] 1

l1[c(1, 2)]

## $x

## [1] 1

##

## $y

## [1] TRUE FALSE

l1[c(TRUE, FALSE, TRUE)]

## $x

## [1] 1

##

## $z

## [1] "a" "b" "c"

To summarize, we can say that  [[ means extracting one element from a vector or list, and [ means subsetting a vector or list. Subsetting a vector will result in a vector. Likewise, subsetting a list will result in a list.

**Named lists**

Irrespective of whether the list members have already got names when the list is created, we can always name or rename the members of a list, by simply naming a vector:

names(l1) <- c("A","B","C")

l1

## $A

## [1] 1

##

## $B

## [1] TRUE FALSE

##

## $C

## [1] "a" "b" "c"

To remove their names, we replace the names of l1 with NULL:

names(l1) <- NULL

l1

## [[1]]

## [1] 1

##

## [[2]]

## [1] TRUE FALSE

##

## [[3]]

## [1] "a" "b" "c"

Once the names of list members are removed, we can no longer access the list members by name but by position and logical criterion.

**Setting values**

Setting the values in a list is as straightforward as working with vectors:

l1 <- list(x = 1, y = c(TRUE, FALSE), z = c("a", "b", "c"))

l1$x <- 0

If we assign a value to a nonexisting member, we will add a new member to the list with the given name or position:

l1$m <- 4

l1

## $x

## [1] 0

##

## $y

## [1] TRUE FALSE

##

## $z

## [1] "a" "b" "c"

##

## $m

## [1] 4

Also, we can set multiple values at the same time:

l1[c("y", "z")] <- list(y = "new value for y", z = c(1, 2))

l1

## $x

## [1] 0

##

## $y

## [1] "new value for y"

##

## $z

## [1] 1 2

##

## $m

## [1] 4

If we need to remove some of the members in a list, just assign the NULL value to them:

l1$x <- NULL

l1

## $y

## [1] "new value for y"

##

## $z

## [1] 1 2

##

## $m

## [1] 4

We can remove more than one member from a list altogether:

l1[c("z", "m")] <- NULL

l1

## $y

## [1] "new value for y"

**Other functions**

Many functions in R are related to lists. For example, if we are not sure whether an object is a list or not, we can call is.list() to find out:

l2 <- list(a = c(1, 2, 3), b = c("x", "y", "z", "w"))

is.list(l2)

## [1] TRUE

is.list(l2$a)

## [1] FALSE

Here, l2 is a list, and butl2$a is a numeric vector rather than a list.

We can also convert a vector to a list using as.list():

l3 <- as.list(c(a = 1, b =2, c = 3))

l3

## $a

## [1] 1

##

## $b

## [1] 2

##

## $c

## [1] 3

It is also easy to coerce a list to a vector by calling unlist that basically converts all list members and puts them to a vector of a compatible type:

l4 <- list(a = 1, b = 2, c = 3)

unlist(l4)

## a b c

## 1 2 3

If we unlist a list of numbers and texts in mixture, all members will be converted to the closest type that each one can be converted to:

l4 <- list(a = 1, b = 2, c = "hello")

unlist(l4)

## a b c

## "1" "2" "hello"

Here, l4$a and l4$b are numbers and can be converted to a character; however,  butl4$c is a character vector and cannot be converted to numeric values. Therefore, their closest type that is compatible with all elements is a character vector.

# Data frames

A data frame represents a set of data with a number of rows and columns. It looks like a matrix but its columns are not necessarily of the same type. This is consistent with the most commonly seen formats of datasets: each row, or data record, is described by multiple columns of various types.

The following table is an example that can be fully characterized by a data frame.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Gender** | **Age** | **Major** |
| Ken | Male | 24 | Finance |
| Ashley | Female | 25 | Statistics |
| Jennifer | Female | 23 | Computer Science |

## Creating a data frame

To create a data frame, we can call data.frame() and supply the data of each column by a vector of the corresponding type:

persons <- data.frame(Name = c("Ken", "Ashley", "Jennifer"),

Gender = c("Male", "Female", "Female"),

Age = c(24, 25, 23),

Major = c("Finance", "Statistics", "Computer Science"))

persons

## Name Gender Age Major

## 1 Ken Male 24 Finance

## 2 Ashley Female 25 Statistics

## 3 Jennifer Female 23 Computer Science

Note that creating a data frame is exactly the same as creating a list. This is because, in essence, a data frame is a list in which each element is a vector and represents a table column and has the same number of elements.

Other than creating a data frame from raw data, we can also create it from a list by calling either data.frame directly or as.data.frame:

l1 <- list(x = c(1, 2, 3), y = c("a", "b", "c"))

data.frame(l1)

## x y

## 1 1 a

## 2 2 b

## 3 3 c

as.data.frame(l1)

## x y

## 1 1 a

## 2 2 b

## 3 3 c

We can also create a data frame from a matrix with the same method:

m1 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = FALSE)

data.frame(m1)

## X1 X2 X3

## 1 1 4 7

## 2 2 5 8

## 3 3 6 9

as.data.frame(m1)

## V1 V2 V3

## 1 1 4 7

## 2 2 5 8

## 3 3 6 9

Note that the conversion also automatically assigns column names to the new data frame. In fact, as you may verify, if the matrix already has column names or row names, they will be preserved in the conversion.

## Naming rows and columns

Since a data frame is a list but also looks like a matrix, the ways we access these two types of objects both apply to a data frame:

df1 <- data.frame(id = 1:5, x = c(0, 2, 1, -1, -3), y = c(0.5, 0.2, 0.1, 0.5, 0.9))

df1

## id x y

## 1 1 0 0.5

## 2 2 2 0.2

## 3 3 1 0.1

## 4 4 -1 0.5

## 5 5 -3 0.9

We can rename the columns and rows just like we do with a matrix:

colnames(df1) <- c("id", "level", "score")

rownames(df1) <- letters[1:5]

df1

## id level score

## a 1 0 0.5

## b 2 2 0.2

## c 3 1 0.1

## d 4 -1 0.5

## e 5 -3 0.9

## Subsetting a data frame

Since a data frame is a matrix-like list of column vectors, we can use both sets of notations to access the elements and subsets in a data frame.

### SUBSETTING A DATA FRAME AS A LIST

If we would like to regard a data frame as a list of vectors, we can use list notations to extract a value or perform subsetting.

For example, we can use $ to extract the values of one column by name, or use [[ to do so by position:

df1$id

## [1] 1 2 3 4 5

df1[[1]]

## [1] 1 2 3 4 5

List subsetting perfectly applies to a data frame and also yields a new data frame. The subsetting operator ([) allows us to use a numeric vector to extract columns by position, a character vector to extract columns by name, or a logical vector to extract columns by TRUE and FALSE selection:

df1[1]

## id

## a 1

## b 2

## c 3

## d 4

## e 5

df1[1:2]

## id level

## a 1 0

## b 2 2

## c 3 1

## d 4 -1

## e 5 -3

df1["level"]

## level

## a 0

## b 2

## c 1

## d -1

## e -3

df1[c("id", "score")]

## id score

## a 1 0.5

## b 2 0.2

## c 3 0.1

## d 4 0.5

## e 5 0.9

df1[c(TRUE, FALSE, TRUE)]

## id score

## a 1 0.5

## b 2 0.2

## c 3 0.1

## d 4 0.5

## e 5 0.9

### SUBSETTING A DATA FRAME AS A MATRIX

However, the list notation does not support row selection. In contrast, the matrix notation provides more flexibility. If we view a data frame as a matrix, the two-dimensional accessor enables us to easily access an entry of a subset, which supports both column selection and row selection.

In other words, we can use the [row, column] notation to subset a data frame by specifying the row selector and column selector, which can be numeric vectors, character vectors, and/or logical vectors.

For example, we can specify the column selector:

df1[, "level"]

## [1] 0 2 1 -1 -3

df1[, c("id", "level")]

## id level

## a 1 0

## b 2 2

## c 3 1

## d 4 -1

## e 5 -3

df1[, 1:2]

## id level

## a 1 0

## b 2 2

## c 3 1

## d 4 -1

## e 5 -3

Alternatively, we can specify the row selector:

df1[1:4,]

## id level score

## a 1 0 0.5

## b 2 2 0.2

## c 3 1 0.1

## d 4 -1 0.5

df1[c("c", "e"),]

## id level score

## c 3 1 0.1

## e 5 -3 0.9

We can even specify both selectors at the same time:

df1[1:4, "id"]

## [1] 1 2 3 4

df1[1:3, c("id", "score")]

## id score

## a 1 0.5

## b 2 0.2

## c 3 0.1

Note that the matrix notation automatically simplifies the output. That is, if only one column is selected, the result won't be a data frame but the values of that column. To always keep the result as a data frame, even if it only has a single column, we can use both notations together:

df1[1:4,]["id"]

## id

## a 1

## b 2

## c 3

## d 4

Here, the first group of brackets subsets the data frame as a matrix with the first four rows and all columns selected. The second group of brackets subsets the resultant data frame as a list with only the id column selected, which results in a data frame.

Another way is to specify drop = FALSE to avoid simplifying the results:

df1[1:4, "id", drop = FALSE]

## id

## a 1

## b 2

## c 3

## d 4

If you expect the output of a data frame subsetting to always be a data frame, you should always set drop = FALSE; otherwise, some edge cases (like a user input selecting only one column) may end up in unexpected behaviors if you assume that you will get a data frame but actually get a vector.

### FILTERING DATA

The following code filters the rows of df1 by criterionscore >= 0.5 and selects the idand level columns:

df1$score >= 0.5

## [1] TRUE FALSE FALSE TRUE TRUE

df1[df1$score >= 0.5, c("id", "level")]

## id level

## a 1 0

## d 4 -1

## e 5 -3

The following code filters the rows of df1 by a criterion that the row name must be among a, d, or e, and selects the id and score columns:

rownames(df1) %in% c("a", "d", "e")

## [1] TRUE FALSE FALSE TRUE TRUE

df1[rownames(df1) %in% c("a", "d", "e"), c("id", "score")]

## id score

## a 1 0.5

## d 4 0.5

## e 5 0.9

Both of these examples basically use matrix notation to select rows by a logical vector and select columns by a character vector.

## Setting values

Setting the values of a subset of a data frame allows both methods working with a list and a matrix.

### SETTING VALUES AS A LIST

We can assign new values to a list member using $ and <- together:

df1$score <- c(0.6, 0.3, 0.2, 0.4, 0.8)

df1

## id level score

## a 1 0 0.6

## b 2 2 0.3

## c 3 1 0.2

## d 4 -1 0.4

## e 5 -3 0.8

Alternatively, [ works too, and it also allows multiple changes in one expression in contrast to [[, which only allows modifying one column at a time:

df1["score"] <- c(0.8, 0.5, 0.2, 0.4, 0.8)

df1

## id level score

## a 1 0 0.8

## b 2 2 0.5

## c 3 1 0.2

## d 4 -1 0.4

## e 5 -3 0.8

df1[["score"]] <- c(0.4, 0.5, 0.2, 0.8, 0.4)

df1

## id level score

## a 1 0 0.4

## b 2 2 0.5

## c 3 1 0.2

## d 4 -1 0.8

## e 5 -3 0.4

df1[c("level", "score")] <- list(level = c(1, 2, 1, 0, 0), score = c(0.1, 0.2, 0.3, 0.4, 0.5))

df1

## id level score

## a 1 1 0.1

## b 2 2 0.2

## c 3 1 0.3

## d 4 0 0.4

## e 5 0 0.5

### SETTING VALUES AS A MATRIX

Using list notations to set values of a data frame has the same problem as subsetting–we can only access the columns. If we need to set values with more flexibility, we can use matrix notations:

df1[1:3, "level"] <- c(-1, 0, 1)

df1

## id level score

## a 1 -1 0.1

## b 2 0 0.2

## c 3 1 0.3

## d 4 0 0.4

## e 5 0 0.5

df1[1:2, c("level", "score")] <- list(level = c(0, 0), score = c(0.9, 1.0))

df1

## id level score

## a 1 0 0.9

## b 2 0 1.0

## c 3 1 0.3

## d 4 0 0.4

## e 5 0 0.5

## Factors

One thing to notice is that the default behavior of a data frame tries to use memory more efficiently. Sometimes, this behavior might silently lead to unexpected problems.

For example, when we create a data frame by supplying a character vector as a column, it will by default convert the character vector to a factor that only stores the same value once so that repetitions will not cost much memory. In fact, a factor is essentially an integer vector with a pre-specified set of possible values called levels to represent values of limited possibilities.

We can verify this by calling str() on the data frame persons we created in the beginning:

str(persons)

## 'data.frame': 3 obs. of 4 variables:

## $ Name : Factor w/ 3 levels "Ashley","Jennifer",..: 3 1 2

## $ Gender: Factor w/ 2 levels "Female","Male": 2 1 1

## $ Age : num 24 25 23

## $ Major : Factor w/ 3 levels "Computer Science",..: 2 3 1

As we can clearly find out that Name, Gender, and Major are not character vectors but factor objects. It is reasonable that Gender is represented by a factor because it may only be either Female or Male, so using two integers to represent these two values is more efficient than using a character vector to store all the values regardless of the repetition.

However, it may induce problems for other columns not limited to taking several possible values. For example, if we want to set a name in persons:

persons[1, "Name"] <- "John"

## Warning in `[<-.factor`(`\*tmp\*`, iseq, value = "John"): invalid factor

## level, NA generated

persons

## Name Gender Age Major

## 1 <NA> Male 24 Finance

## 2 Ashley Female 25 Statistics

## 3 Jennifer Female 23 Computer Science

A warning message appears. This happens because in the initial Name dictionary, there is no word called John, therefore we cannot set the name of the first person to be such a non-existing value. The same thing happens when we set any Gender to be Unknown. The reason is exactly the same: when the column is initially created from a character vector when we define a data frame, the column will by default be a factor whose value must be taken from the dictionary created from the unique values in that character vector.

This behavior is sometimes very annoying and does not really help much, especially as memory is cheap today. The simplest way to avoid this behavior is to set stringsAsFactors = FALSE when we create a data frame using data.frame():

persons <- data.frame(Name = c("Ken", "Ashley", "Jennifer"),

Gender = factor(c("Male", "Female", "Female")),

Age = c(24, 25, 23),

Major = c("Finance", "Statistics", "Computer Science"),

stringsAsFactors = FALSE)

str(persons)

## 'data.frame': 3 obs. of 4 variables:

## $ Name : chr "Ken" "Ashley" "Jennifer"

## $ Gender: Factor w/ 2 levels "Female","Male": 2 1 1

## $ Age : num 24 25 23

## $ Major : chr "Finance" "Statistics" "Computer Science"

If we really want a factor object to play its role, we can explicitly call factor() at specific columns, just like we did previously  for the Gender column.

## Useful functions for data frames

There are many useful functions for a data frame. Here we only introduce a few but the most commonly used ones.

The summary() function works with a data frame by generating a table that shows the summary statistics of each column:

summary(persons)

## Name Gender Age Major

## Length:3 Female:2 Min. :23.0 Length:3

## Class :character Male :1 1st Qu.:23.5 Class :character

## Mode :character Median :24.0 Mode :character

## Mean :24.0

## 3rd Qu.:24.5

## Max. :25.0

For a factor Gender, the summary counts the number of rows taking each value, or level. For a numeric vector, the summary shows the important quantiles of the numbers. For other types of columns, it shows the length, class, and mode of them. Another common demand is binding multiple data frames together by either row or column. For this purpose, we can use rbind() and cbind()which, as their names suggest, perform row binding and column binding respectively.

If we want to append some rows to a data frame, in this case, add a new record of a person, we can use rbind():

rbind(persons, data.frame(Name = "John", Gender = "Male", Age = 25, Major = "Statistics"))

## Name Gender Age Major

## 1 Ken Male 24 Finance

## 2 Ashley Female 25 Statistics

## 3 Jennifer Female 23 Computer Science

## 4 John Male 25 Statistics

If we want to append some columns to a data frame, in this case, add two new columns to indicate whether each person is registered and the number of projects in hand, we can use cbind():

cbind(persons, Registered = c(TRUE, TRUE, FALSE), Projects = c(3, 2, 3))

## Name Gender Age Major Registered Projects

## 1 Ken Male 24 Finance TRUE 3

## 2 Ashley Female 25 Statistics TRUE 2

## 3 Jennifer Female 23 Computer Science FALSE 3

Note that rbind() and cbind() do not modify the original data but create a new data frame with given rows or columns appended.

Another useful function is expand.grid(). This generates a data frame that includes all combinations of the values in the columns:

expand.grid(type = c("A", "B"), class = c("M", "L", "XL"))

## type class

## 1 A M

## 2 B M

## 3 A L

## 4 B L

## 5 A XL

## 6 B XL

There are many other useful functions working with data frames. We will discuss these functions in data manipulation chapters.

## Loading and writing data on disk

In practice, data is usually stored in files. R provides a number of functions to read a table from a file or write a data frame to a file. If a file stores a table, it is often well organized and follows some convention that specifies how rows and columns are arranged. In most cases, we don't have to read a file byte to byte but call functions such as read.table() or read.csv().

The most popular software-neutral data format is **CSV** (**Comma-Separated Values**). The format is basically organized in a way that values in different columns are separated by a comma and the first row is by default regarded as the header. For example, persons may be represented in the following CSV format:

Name,Gender,Age,MajorKen,Male,24,FinanceAshley,Female,25,StatisticsJennifer,Female,23,Computer Science

To read the data into the R environment, we only need to call read.csv(file) where the file is the path of the file. To ensure that the data file can be found, please place the data folder directly in your working directory, call getwd() to find out. We'll talk about this in detail in the next chapter:

read.csv("data/persons.csv")

## Name Gender Age Major

## 1 Ken Male 24 Finance

## 2 Ashley Female 25 Statistics

## 3 Jennifer Female 23 Computer Science

If we need to save a data frame to a CSV file, we may call write.csv(file) with some additional arguments:

write.csv(persons, "data/persons.csv", row.names = FALSE, quote = FALSE)

The argument row.names = FALSE avoids storing the row names which are not necessary, and the argumentquote = FALSE avoids quoting text in the output, both of which in most cases are not necessary.

There are a number of built-in functions and several packages related to reading and writing data in different formats. We will cover this topic in later chapters.

**Functions**

A function is an object you can call. Basically, it is a machine with internal logic that takes a group of inputs (parameters or arguments) and returns a value as output.

In the previous sections, we encountered some built-in functions of R. For example, is.numeric()takes an argument that can be any R object and returns a logical value that indicates whether the object is a numeric vector. Similarly, is.function() can tell whether a given R object is a function object.

In fact, in R environment, everything we use is an object, everything we do is a function, and, maybe to your surprise, all functions are still objects. Even <- and + are both functions that take two arguments. Although they are called binary operators, they are essentially functions.

When we do casual, interactive data analysis, at times, we won't have to write any function on our own since the built-in functions and those provided by thousands of packages are usually enough.

However, if you need to repeat your logic or a process in data manipulation or analysis, those functions may not fully serve your purpose because they are not designed to meet the specific needs of a task or the format of a particular dataset. Then, you need to create your own functions targeting a specific set of demands.

**Creating a function**

It is easy to create a function in R. Suppose we define a function called add that simply adds two numbers x and y, respectively:

add <- function(x, y) { x + y}

The syntax function (x, y) specifies the arguments of the function. In other words, the function takes two arguments named x and y. The { x + y } is the function body that contains a series of expressions expressed in terms of x, y and other symbols available. The value of the last expression determines the value returned by the function unless return() is called inside the function. Finally, the function is assigned to add so that we can call this function using add later on.

Creating such a simple function, or any more complicated functions, does not impose any difference on evaluating a vector. The function in R just acts like another object. To see what object add refers to, just type add at the console:

add

## function(x, y) {

## x + y

## }

**Calling a function**

Once the function is defined, we can call the function just as we do in math. The calling requires the same syntax: name (arg1, arg2, ...). Take a look at the following:

add(2, 3)

## [1] 5

The call is quite transparent. When we evaluate such a call, R will find out if there is a function named add in the environment. Then, it will figure out that add refers to the function we just created and creates a local environment in which x takes 2 and y takes 3. The expression in the function body is then evaluated given the values of the arguments. Finally, the function returns the value of that expression, 5.

**Dynamic typing**

Functions in R can be very flexible since it is not strongly typed. In other words, the type of inputs are not fixed prior to the calling. Even if the function is originally designed to work for scalar numbers, it is automatically generalized to also work with all vectors as long as + works with them. For example, we can run the following code without any change in the function:

add(c(2, 3), 4)

## [1] 6 7

The preceding example does not really demonstrate the flexibility of dynamic typing because scalar is also a vector in R. A more qualified example is:

add(as.Date("2014-06-01"), 1)

## [1] "2014-06-02"

The function put the two arguments into the expression without any type checking. as.Date()creates a Date object, which has a date representation. Without changing any code of add, it works with Date perfectly. The function fails only when + is not well-defined for the two arguments:

add(list(a = 1), list(a = 2))

## Error in x + y: non-numeric argument to binary operator

**Generalizing a function**

Functions are a well-defined abstraction of a particular set of logic or process intended for solving some particular problem. Developers often want a function to be general enough to adapt to a wide range of use cases so that we can easily use it to solve similar problems without writing too many specialized functions for each problem.

To make a function more widely applicable is called **generalization**. It is very handy to generalize a function in a weakly-typed programming language like R, but it can be error-prone if it is incorrectly implemented.

To make add() more general so that it can handle various primitive algebraic operations, we can define another function called calc. This new function accepts three arguments where x and y are the two vectors, and type accepts a character vector which is the kind of algebraic operation the user wants to perform.

The following code implements such a function using **flow control**, which we will cover soon, but it should be easy to understand at first look. In this code, the choice of expression to be evaluated depends on the value of type:

calc <- function(x, y, type) {

  if (type == "add") {

    x + y

  } else if (type == "minus") {

    x - y

} else if (type == "multiply") {

x \* y

} else if (type == "divide") {

x / y

} else {

stop("Unknown type of operation")

}

}

Once the function is defined, we can call it by supplying appropriate arguments:

calc(2, 3, "minus")

## [1] -1

The function automatically works with numeric vectors:

calc(c(2, 5), c(3, 6), "divide")

## [1] 0.6666667 0.8333333

The function is also generalized to work with non-numeric vectors for which + is well-defined:

calc(as.Date("2014-06-01"), 3, "add")

## [1] "2014-06-04"

Consider supplying some invalid arguments:

calc(1, 2, "what")

## Error in calc(1, 2, "what"): Unknown type of operation

In this case, no conditions are satisfied, so the expression in the last else block will be evaluated. The stop() call yields an error message and terminates the whole evaluation immediately.

The functions seem to work fine and consider all possible situations with invalid arguments. However, it is not true:

calc(1, 2, c("add", "minue"))

## Warning in if (type == "add") {: the condition has length > 1 and only the

## first element will be used

## [1] 3

Here, we didn't consider the case where type is given as a multi-element vector. The problem is: when such a vector is compared with another vector, it will also result in a multi-element logical vector, it will also result in a mult-element logical vector which makes an ambiguous condition for if. Consider what it means by if(c(TRUE, FALSE))?

To avoid such ambiguity explicitly, we need to refine the function so that the error will be more informative and transparent. To proceed, we just need to check whether the vector has the length 1:

calc <- function(x, y, type) {

if (length(type) > 1L) stop("Only a single type is accepted")

if (type == "add") {

x + y

} else if (type == "minus") {

x - y

} else if (type == "multiply") {

x \* y

} else if (type == "divide") {

x / y

} else {

stop("Unknown type of operation")

}

}

Then, we retry the trouble-making call and see how the exception is handled by pre-checking of arguments:

calc(1, 2, c("add", "minue"))

## Error in calc(1, 2, c("add", "minue")): Only a single type is accepted

**Default value for function arguments**

Some functions are very flexible because they accept a wide range of input and meet a variety of demands. In many cases, more flexibility means an increasing number of arguments.

If we have to specify tens of arguments each time using a very flexible function, it would certainly be a mess to look at the code. In this case, reasonable default values for arguments will largely simplify the code to call a function.

To set the default value of an argument, use arg = value. This will make the argument optional. The following example creates a function with an optional argument:

increase <- function(x, y = 1) {

x + y

}

The new function increase() allows us to call it with only x. In this case, y automatically takes 1unless it is explictly specified.

increase(1)

## [1] 2

increase(c(1, 2, 3))

## [1] 2 3 4

Many R functions have multiple arguments and some of them are given default values. Sometimes, it is tricky to determine the default values of arguments because it heavily relies on the intention of most users.