Mastering Scientific Computing with R

With this book, you will learn not just about R, but how to use R to answer conceptual, scientific, and experimental questions.

Beginning with an overview of fundamental R concepts, you’ll learn how R can be used to achieve the most commonly needed scientific data analysis tasks: testing for statistically significant differences between groups and model relationships in data. You will delve into linear algebra and matrix operations with an emphasis not on the R syntax, but on how these operations can be used to address common computational or analytical needs. This book also covers the application of matrix operations for the purpose of finding structure in high-dimensional data using the principal component, exploratory factor, and confirmatory factor analysis in addition to structural equation modeling. You will also master methods for simulation and learn about an advanced analytical method.

Who this book is written for

If you want to learn how to quantitatively answer scientific questions for practical purposes using the powerful R language and the open source R tool ecosystem, this book is ideal for you. It is ideally suited for scientists who understand scientific concepts, know a little R, and want to be able to start applying R to be able to answer empirical scientific questions. Some R exposure is helpful, but not compulsory.

What you will learn from this book

- Master data management in R
- Perform hypothesis tests using both parametric and nonparametric methods
- Understand how to perform statistical modeling using linear methods
- Model nonlinear relationships in data with kernel density methods
- Use matrix operations to improve coding productivity
- Utilize the observed data to model unobserved variables
- Deal with missing data using multiple imputations
- Simplify high-dimensional data using principal components, singular value decomposition, and factor analysis
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Programming with R'
- A synopsis of the book’s content
- More information on Mastering Scientific Computing with R

About the Authors

Paul Gerrard is a physician and healthcare researcher who is based out of Portland, Maine, where he currently serves as the medical director of the cardiopulmonary rehabilitation program at New England Rehabilitation Hospital of Portland. He studied business economics in college. After completing medical school, he did a residency in physical medicine and rehabilitation at Harvard Medical School and Spaulding Rehabilitation Hospital, where he served as chief resident and stayed on as faculty at Harvard before moving to Portland. He continues to collaborate on research projects with researchers at other academic institutions within the Boston area and around the country. He has published and presented research on a range of topics, including traumatic brain injury, burn rehabilitation, health outcomes, and the epidemiology of disabling medical conditions.

I would like to thank my beautiful wife, Deirdre, and my son, Patrick. My work on this book is dedicated to the loving memory of Fiona.
Radia M. Johnson has a doctorate degree in immunology and currently works as a research scientist at the Institute for Research in Immunology and Cancer at the Université de Montréal, where she uses genomics and bioinformatics to identify and characterize the molecular changes that contribute to cancer development. She routinely uses R and other computer programming languages to analyze large data sets from ongoing collaborative projects. Since obtaining her PhD at the University of Toronto, she has also worked as a research associate at the University of Cambridge in Hematology, where she gained experience using system biology to study blood cancer.

I would like to thank Dr. Charlie Massie for teaching me to love programming in R and Dr. Phil Kousis for all his support through the years. You are both excellent mentors and wonderful friends!
Mastering Scientific Computing with R

As an open source computing environment, R is rapidly becoming the lingua franca of the statistical computing community. R’s powerful base functions, powerful statistical tools, open source nature, and avid user community have led to R having an expansive library of powerful, cutting-edge quantitative methods not yet available to users of other high-cost statistical programs.

With this book, you will learn not just about R, but how to use R to answer conceptual, scientific, and experimental questions.

Beginning with an overview of fundamental R concepts, including data types, R program flow, and basic coding techniques, you'll learn how R can be used to achieve the most commonly needed scientific data analysis tasks, including testing for statistically significant differences between groups and model relationships in data. You will also learn parametric and nonparametric techniques for both difference testing and relationship modeling.

You will delve into linear algebra and matrix operations with an emphasis not on the R syntax, but on how these operations can be used to address common computational or analytical needs. This book also covers the application of matrix operations for the purpose of finding a structure in high-dimensional data using the principal component, exploratory factor, and confirmatory factor analysis in addition to structural equation modeling. You will also master methods for simulation, learn about an advanced analytical method, and finish by going to the next level with advanced data management focused on dealing with messy and problematic datasets that serious analysts deal with daily.

By the end of this book, you will be able to undertake publication-quality data analysis in R.

What This Book Covers

Chapter 1, Programming with R, presents an overview of how data is stored and accessed in R. Then, we will go over how to load data into R using built-in functions and useful packages for easy import from Excel worksheets. We will also cover how to use flow control statements and functions to reduce complexity and help you program more efficiently.
Chapter 2, *Statistical Methods with R*, presents an overview of how to summarize your data and get useful statistical information for downstream analysis. We will show you how to plot and get statistical information from probability distributions and how to test the fit of your sample distribution to well-defined probability distributions.

Chapter 3, *Linear Models*, covers linear models, which are probably the most commonly used statistical methods to study the relationships between variables. The *Generalized linear model* section will delve into a bit more detail than typical R books, discussing the nature of link functions and canonical link functions.

Chapter 4, *Nonlinear Methods*, reviews applications of nonlinear methods in R using both parametric and nonparametric methods for both theory-driven and exploratory analysis.

Chapter 5, *Linear Algebra*, covers algebra techniques in R. We will also learn linear algebra operations including transposition, inversion, matrix multiplication, and a number of matrix transformations.

Chapter 6, *Principal Component Analysis and the Common Factor Model*, helps you understand the application of linear algebra to covariance and correlation matrices. We will cover how to use PCA to account for total variance in a set of variables and how to use EFA to model common variance among these variables in R.

Chapter 7, *Structural Equation Modeling and Confirmatory Factor Analysis*, covers the fundamental ideas underlying structural equation modeling, which are often overlooked in other books discussing SEM in R, and then delve into how SEM is done in R.

Chapter 8, *Simulations*, explains how to perform basic sample simulations and how to use simulations to answer statistical problems. We will also learn how to use R to generate random numbers, and how to simulate random variables from several common probability distributions.

Chapter 9, *Optimization*, explores a variety of methods and techniques to optimize a variety of functions. We will also cover how to use a wide range of R packages and functions to set up, solve, and visualize different optimization problems.

Chapter 10, *Advanced Data Management*, walks you through the basic techniques for data handling and some basic memory management considerations.
Scientific computing is an informatics approach to problem solving using mathematical models and/or applying quantitative analysis techniques to interpret, visualize, and solve scientific problems. Generally speaking, scientists and data analysts are concerned with understanding certain phenomena or processes using observations from an experiment or through simulation. For example, a biologist may want to understand what changes in gene expression are required for a normal cell to become a cancerous cell, or a physicist may want to study the life cycle of galaxies through numerical simulations. In both cases, they will need to collect the data, and then manipulate and process it before it can be visualized and interpreted to answer their research question. Scientific computing is involved in all these steps.

R is an excellent open source language for scientific computing. R is broadly used in companies and academics as it has great performance value and provides a cutting-edge software environment. It was initially designed as a software tool for statistical modeling but has since then evolved into a powerful tool for data mining and analytics. In addition to its rich collection of classical numerical methods or basic actions, there are also hundreds of R packages for a wide variety of scientific computing needs such as state-of-the-art visualization methods, specialized data analysis tools, machine learning, and even packages such as Shiny to build interactive web applications. In this book, we will teach you how to use R and some of its packages to define and manipulate your data using a variety of methods for data exploration and visualization. This book will present to you state-of-the-art mathematical and statistical methods needed for scientific computing. We will also teach you how to use R to evaluate complex arithmetic expressions and statistical modeling. We will also cover how to deal with missing data and the steps needed to write your own functions tailored to your analysis requirements. By the end of this book, you will not only be comfortable using R and its many packages, but you will also be able to write your own code to solve your own scientific problems.
This first chapter will present an overview of how data is stored and accessed in R. Then, we will look at how to load your data into R using built-in functions and useful packages, in order to easily import data from Excel worksheets. We will also show you how to transform your data using the reshape2 package to make your data ready to graph by plotting functions such as those provided by the ggplot2 package. Next, you will learn how to use flow-control statements and functions to reduce complexity, and help you program more efficiently. Lastly, we will go over some of the debugging tools available in R to help you successfully run your programs in R.

The following is a list of the topics that we will cover in this chapter:

- Atomic vectors
- Lists
- Object attributes
- Factors
- Matrices and arrays
- Data frames
- Plots
- Flow control
- Functions
- General programming and debugging tools

Before we begin our overview of R data structures, if you haven't already installed R, you can download the most recent version from http://cran.r-project.org. R compiles and runs on Linux, Mac, and Windows so that you can download the precompiled binaries to install it on your computer. For example, go to http://cran.r-project.org, click on Download R for Linux, and then click on ubuntu to get the most up-to-date instructions to install R on Ubuntu. To install R on Windows, click on Download R for Windows, and then click on base for the download link and installation instructions. For Mac OS users, click on Download R for (Mac) OS X for the download links and installation instructions.
In addition to the most recent version of R, you may also want to download RStudio, which is an integrated development environment that provides a powerful user interface that makes learning R easier and fun. The main limitation of RStudio is that it has difficulty loading very large datasets. So if you are working with very large tables, you may want to run your analysis in R directly. That being said, RStudio is great to visualize the objects you stored in your workplace at the click of a button. You can easily search help pages and packages by clicking on the appropriate tabs. Essentially, RStudio provides all that you need to help analyze your data at your fingertips. The following screenshot is an example of the RStudio user interface running the code from this chapter:

You can download RStudio for all platforms at http://www.rstudio.com/products/rstudio/download/.

Finally, the font conventions used in this book are as follows. The code you should directly type into R is preceded by > and any lines preceded by # will be treated as comment in R.

> The user will type this into R
This is the response from R
> # If the user types this, R will treat it as a comment
Data structures in R

R objects can be grouped into two categories:

- **Homogeneous**: This is when the content is of the same type of data
- **Heterogeneous**: This is when the content contains different types of data

**Atomic vectors, Matrices, or Arrays** are data structures that are used to store homogenous data, while **Lists** and **Data frames** are typically used to store heterogeneous data. R objects can also be organized based on the number of dimensions they contain. For example, atomic vectors and lists are one-dimensional objects, whereas matrices and data frames are two-dimensional objects. Arrays, however, are objects that can have any number of dimensions. Unlike other programming languages such as Perl, R does not have scalar or zero-dimensional objects. All single numbers and strings are stored in vectors of length one.

**Atomic vectors**

Vectors are the basic data structure in R and include atomic vectors and lists. Atomic vectors are flat and can be logical, numeric (double), integer, character, complex, or raw. To create a vector, we use the `c()` function, which means combine elements into a vector:

```r
> x <- c(1, 2, 3)
```

To create an integer vector, add the number followed by L, as follows:

```r
> integer_vector <- c(1L, 2L, 12L, 29L)
> integer_vector
[1]  1  2 12 29
```

To create a logical vector, add `TRUE (T)` and `FALSE (F)`, as follows:

```r
> logical_vector <- c(T, TRUE, F, FALSE)
> logical_vector
[1] TRUE TRUE FALSE FALSE
```
To create a vector containing strings, simply add the words/phrases in double quotes:

```
> character_vector <- c("Apple", "Pear", "Red", "Green", "These are my favorite fruits and colors")
> character_vector
[1] "Apple"  "Pear"  "Red"   "Green"  "These are my favorite fruits and colors"
```

R also includes functions that allow you to create vectors containing repetitive elements with `rep()` or a sequence of numbers with `seq()`:

```
> seq(1, 12, by=3)
[1]  1  4  7 10
> seq(1, 12)  #note the default parameter for by is 1
[1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Instead of using the `seq()` function, you can also use a colon, :, to indicate that you would like numbers 1 to 12 to be stored as a vector, as shown in the following example:

```
> y <- 1:12
> y
[1]  1  2  3  4  5  6  7  8  9 10 11 12
> z <- c(1:3, y)
> z
[1]  1  2  3  1  2  3  4  5  6  7  8  9 10 11 12
```

To replicate elements of a vector, you can simply use the `rep()` function, as follows:

```
> x <- rep(3, 14)
> x
[1]  3  3  3  3  3  3  3  3  3  3  3  3  3  3
```
You can also replicate complex patterns as follows:

```r
> rep(seq(1, 4), 3)
[1] 1 2 3 4 1 2 3 4 1 2 3 4
```

Atomic vectors can only be of one type so if you mix numbers and strings, your vector will be coerced into the most flexible type. The most to the least flexible vector types are **Character**, **numeric**, **integer**, and **logical**, as shown in the following diagram:

![Vector Types Diagram](image)

This means that if you mix numbers with strings, your vector will be coerced into a character vector, which is the most flexible type of the two. In the following paragraph, there are two different examples showing this coercion in practice. The first example shows that when a character and numeric vector are combined, the class of this new object becomes a character vector because a character vector is more flexible than a numeric vector. Similarly, in the second example, we see that the class of the new object `x` is numeric because a numeric vector is more flexible than an integer vector. The two examples are as follows:

**Example 1:**

```r
> mixed_vector <- c(character_vector, numeric_vector)
> mixed_vector
[1] "Apple" "Pear" "Red" "Green" "These are my favorite fruits and colors"
[6] "1" "3.4" "5" "10"
> class(mixed_vector)
[1] "character"
```

**Example 2:**

```r
> x <- c(integer_vector, numeric_vector)
> x
[1] 1.0 2.0 12.0 29.0 1.0 3.4 5.0 10.0
> class(x)
[1] "numeric"
```
At times, you may create a group of objects and forget its name or content. R allows you to quickly retrieve this information using the `ls()` function, which returns a vector of the names of the objects specified in the current workspace or environment.

```r
> ls()
[1] "a"  "A"  "b"  "B"  "C"  "character_vector"  "influence.1"
[8] "influence.1.2"  "influence.2"  "integer_vector"  "logical_vector"
"M"  "mixed_vector"  "N"
[15] "numeric_vector"  "P"  "Q"  "second.degree.mat"  "small.network"
"social.network.mat"  "x"
[22] "y"
```

At first glance, the workspace or environment is the space where you store all the objects you create. More formally, it consists of a frame or collection of named objects, and a pointer to an enclosing environment. When we created the variable `x`, we added it to the global environment, but we could have also created a novel environment and stored it there. For example, let's create a numeric vector `y` and store it in a new environment called `environB`. To create a new environment in R, we use the `new.env()` function as follows:

```r
> environB <- new.env()
> ls(environB)
character(0)
```

As you can see, there are no objects stored in this environment yet because we haven't created any. Now let's create a numeric vector `y` and assign it to `environB` using the `assign()` function:

```r
> assign("y", c(1, 5, 9), envir=environB)
> ls(environB)
[1] "y"
```

Alternatively, we could use the `$` sign to assign a new variable to `environB` as follows:

```r
> environB$z <- "purple"
> ls(environB)
[1] "y"  "z"
```

To see what we stored in `y` and `z`, we can use the `get()` function or the `$` sign as follows:

```r
> get('y', envir=environB)
[1] 1 5 9
> get('z', envir=environB)
[1] "purple"
> environB$y
[1] 1 5 9
```
You can also retrieve additional information on the objects stored in your environment using the `str()` function. This function allows you to inspect the internal structure of the object and print a preview of its contents as follows:

```r
> str(character_vector)
  chr [1:5] "Apple" "Pear" "Red" "Green" ...
> str(integer_vector)
  int [1:4] 1 2 12 29
> str(logical_vector)
  logi [1:4] TRUE TRUE FALSE FALSE
```

To know how many elements are present in our vector, you can use the `length()` function as follows:

```r
> length(integer_vector)
[1] 4
```

Finally, to extract elements from a vector, you can use the position (or index) of the element in square brackets as follows:

```r
> character_vector[5]
[1] "These are my favorite fruits and colors"
> numeric_vector[2]
[1] 3.4
> x <- c(1, 4, 6)
> x[2]
[1] 4
```

### Operations on vectors

Basic mathematical operations can be performed on numeric and integer vectors similar to those you perform on a calculator. The arithmetic operations used are given in the following table:

<table>
<thead>
<tr>
<th>Arithmetic operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ x</td>
</tr>
<tr>
<td>- x</td>
</tr>
<tr>
<td>x + y</td>
</tr>
<tr>
<td>x - y</td>
</tr>
<tr>
<td>x * y</td>
</tr>
<tr>
<td>x / y</td>
</tr>
<tr>
<td>x ^ y</td>
</tr>
<tr>
<td>x %% y</td>
</tr>
<tr>
<td>x %/% y</td>
</tr>
</tbody>
</table>
For example, if we multiply a vector by 2, all the elements of the vector will be multiplied by 2. Let's take a look at the following example:

```r
> x <- c(1, 3, 5, 10)
> x * 2
[1] 2 6 10 20
```

You can also add vectors to each other, in which case the computation will be performed element-wise as follows:

```r
> x <- c(1, 3, 5, 10)
> y <- c(13, 15, 17, 22)
> x + y
[1] 14 18 22 32
```

If the vectors are of different lengths, the shorter vector will be extended to match the length of the longer vector by recycling its elements starting from the first element. However, you will also get a warning message from R in case you did not intend to add vectors of differing length, as follows:

```r
> x
[1] 1 3 5 10
> z <- c(1,3, 4, 6, 10)
> x + z #1 was recycled to complete the operation.
[1] 2 6 9 16 11
Warning message: In x + z : longer object length is not a multiple of shorter object length
```

In addition to this, the standard operators also have `%%`, which indicates $x \mod y$, and `%/%`, which indicates integer division as follows:

```r
> x %% 2
[1] 1 1 1 0
> x %/% 5
[1] 0 0 1 2
```

**Lists**

Unlike atomic vectors, lists can contain different types of elements including lists. To create a list, you use the `list()` function as follows:

```r
> simple_list <- list(1:4, rep(3, 5), "cat")
> str(simple_list)
List of 3
$ : int [1:4] 1 2 3 4
$ : num [1:5] 3 3 3 3 3
```
$ : chr "cat"
> other_list <- list(1:4, "I prefer pears", logical_vector, x, simple_list)
> str(other_list)
List of 5
  $ : int [1:4] 1 2 3 4
  $ : chr "I prefer pears"
  $ : logi [1:4] TRUE TRUE FALSE FALSE
  $ : num [1:3] 1 4 6
  $ :List of 3
    ..$ : int [1:4] 1 2 3 4
    ..$ : num [1:5] 3 3 3 3 3
    ..$ : chr "cat"

If you use the `c()` function to combine lists and atomic vectors, `c()` will coerce the vectors to lists of length one before proceeding. Let's go through a detailed example in R:

> new_list <- c(list(1, 2, simple_list), c(3, 4), seq(5, 6))

Now, let's take a look at the output of the list we just created by entering `new_list` in R:

> new_list
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[[3]][[1]]
[1] 1 2 3 4

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

[[3]][[3]]
[1] "cat"

[[4]]
[1] 3

[[5]]
We can further inspect the `new_list` object that we just created using the `str()` function as follows:

```r
> str(new_list)
List of 7
  $ : num 1
  $ : num 2
  $ :List of 3
    ..$ : int [1:4] 1 2 3 4
    ..$ : num [1:5] 3 3 3 3 3
    ..$ : chr "cat"
  $ : num 3
  $ : num 4
  $ : int 5
  $ : int 6
```

You can also coerce an atomic vector into a list using the `as.list()` function as follows:

```r
> x_as_list <- as.list(x)
> str(x_as_list)
List of 4
  $ : num 1
  $ : num 3
  $ : num 5
  $ : num 10
```

To access different elements in your list, you can use the index position in square brackets `[]`, as you would for a vector, or double square brackets `[[ ]]`. Let's take a look at the following example:

```r
> simple_list
[[1]]
[1] 1 2 3 4
[[2]]
[1] 3 3 3 3
```
As you will no doubt notice, by entering `simple_list[3]`, R returns a list of the single element "cat" as follows:

```r
> str(simple_list[3])
List of 1
$ : chr "cat"
```

If we use the double square brackets, R will return the object type as we initially entered it. So, in this case, it would return a character vector for `simple_list[[3]]` and an integer vector for `simple_list[[1]]` as follows:

```r
> str(simple_list[[3]])
chr "cat"
> str(simple_list[[1]])
int [1:4] 1 2 3 4
```

We can assign these elements to new objects as follows:

```r
> animal <- simple_list[[3]]
> animal
[1] "cat"
> num_vector <- simple_list[[1]]
> num_vector
[1] 1 2 3 4
```

If you would like to access an element of an object in your list, you can use double square brackets `[[ ]]` followed by single square brackets `[ ]` as follows:

```r
> simple_list[[1]][4]
[1] 4
> simple_list[1][4] #Note this format does not return the element
NULL
#Instead you would have to enter
> simple_list[1][[1]][4]
[1] 4
```
Attributes

Objects in R can have additional attributes ascribed to objects that you can store with the `attr()` function, as shown in the following code:

```r
> attr(x_as_list, "new_attribute") <- "This list contains the number of apples eaten for 3 different days"
> attr(x_as_list, "new_attribute")
[1] "This list contains the number of apples eaten for 3 different days"
> str(x_as_list)
List of 3
$ : num 1
$ : num 4
$ : num 6
- attr(*, "new_attribute")= chr "This list contains the number of apples eaten for 3 different days"
```

You can use the `structure()` function, as shown in the following code, to attach an attribute to an object you wish to return:

```r
> structure(as.integer(1:7), added_attribute = "This vector contains integers.")
[1] 1 2 3 4 5 6 7
attr(*, "added_attribute")
[1] "This vector contains integers."
```

In addition to attributes that you create with `attr()`, R also has built-in attributes ascribed to some of its functions, such as `class()`, `dim()`, and `names()`. The `class()` function tells us the class (type) of the object as follows:

```r
> class(simple_list)
[1] "list"
```

The `dim()` function returns the dimension of higher-order objects such as matrices, data frames, and multidimensional arrays. The `names()` function allows you to give names to each element of your vector as follows:

```r
> y <- c(first =1, second =2, third=4, fourth=4)
> y
first second third fourth
 1     2      4      4
```
You can use the `names()` attribute to add the names of each element to your vector as follows:

```r
# element_names <- c("first", "second", "third", "fourth")
> y <- c(1, 2, 4, 4)
> names(y) <- element_names
> y
  first  second  third  fourth
     1       2       4       4
```

You can also modify the names of vector elements using the `setNames()` function as follows:

```r
> setNames(y, c("alpha", "beta", "omega", "psi"))
alpha beta omega psi
     1       2       4       4
```

If you do not provide names for some of your vector elements, the `names()` function will return empty strings, `<NA>`, for the missing ones as follows:

```r
> y <- setNames(y, c("alpha", "beta", "psi"))
> names(y)
[1] "alpha"  "beta"  "psi"   NA
```

However, this does not mean that all vectors require names. In the event that you haven't provided any, `names()` will return `NULL` as follows:

```r
> x <- 1:12
> x <- 1:12
> names(x)
NULL
```

You can remove names using the `unname()` function or by replacing the names with `NULL`:

```r
> unname(y)
[1] 1 2 4 4
> names(y) <- NULL
> names(y)
NULL
```
Factors

When dealing with categorical data, R provides an alternative framework to store character data termed Factors. These are specialized vectors that contain predefined values referred to as Levels. For example, say you have data for "placebo" and "treatment" for four patients, you could store this information as factors instead of a character vector by using the following code:

```r
> drug_response <- c("placebo", "treatment", "placebo", "treatment")
> drug_response <- factor(drug_response)
> drug_response
[1] placebo treatment placebo treatment
Levels: placebo treatment
```

To check the integers used for each level, you can use the `as.integer()` function as follows:

```r
> as.integer(drug_response)
[1] 1 2 1 2
```

Note that you can only adjust elements in a factor with data stored as levels. Say you wanted to change the `drug_response` attribute for the fourth patient from "treatment" to "refused treatment", you will get the following warning message:

```r
Warning message:
In `[<-.factor("*tmp*", 4, value = "refused treatment")`: invalid factor level, NA generated
```

In order to correct this error, you need to first add a new level to the factor using the `factor()` function with the levels argument as follows:

```r
> drug_response <- factor(drug_response, levels = c(levels(drug_response), "refused treatment"))
> drug_response
[1] placebo treatment placebo refused treatment
Levels: placebo treatment refused treatment
```

```r
> as.integer(drug_response)
[1] 1 2 1 3
```
Multidimensional arrays

Multidimensional arrays are created by adding dimensions to the atomic vector created. In computer science, an array is defined as a data structure consisting of elements identified by at least one array index. So, atomic vectors can be seen as one-dimensional arrays. However, as mentioned earlier, arrays can have more than one dimension. These arrays are termed multidimensional arrays. In R, you can create multidimensional arrays using the `array()` function. For example, you can create a three-dimensional array using the `array()` function and specify the dimensions with the `dim` argument using a vector. Let’s create a three-dimensional array of coordinates where the maximal indices in each dimension is 2, 8, and 2 for the first, second, and third dimension, respectively:

```r
> coordinates <- array(1:16, dim=c(2, 8, 2))
> coordinates

, , 1
 [1,]  1  3  5  7  9 11 13 15
 [2,]  2  4  6  8 10 12 14 16

, , 2
 [1,]  1  3  5  7  9 11 13 15
 [2,]  2  4  6  8 10 12 14 16
```

You can also change an object into a multidimensional array using the `dim()` function as follows:

```r
> values <- seq(1, 12, by=2)
> values

[1]  1  3  5  7  9 11
> dim(values) <- c(2,3)
> values

 [,1] [,2]
 [1,]  1  5  9
 [2,]  3  7 11
> dim(values) <- c(3,2)
> values

 [,1] [,2]
 [1,]  1  7
 [2,]  3  9
 [3,]  5 11
```
To access elements of a multidimensional array, you will need to list the coordinates in square brackets [ ] as follows:

```r
> coordinates[1, , ]
   [,1] [,2]  
[1,]  1  1  
[2,]  3  3  
[3,]  5  5  
[4,]  7  7  
[5,]  9  9  
[6,] 11 11  
[7,] 13 13  
[8,] 15 15  
> coordinates[1, 2, ]
   [,1]  
[1] 3 3  
> coordinates[1, 2, 2]
   [,1]  
[1] 3  
```

**Matrices**

Matrices are a special case of two-dimensional arrays and are often created with the `matrix()` function. Instead of the `dim` argument, the `matrix()` function takes the number of rows and columns using the `ncol` and `nrow` arguments, respectively. Alternatively, you can create a matrix by combining vectors as columns and rows using `cbind()` and `rbind()`, respectively:

```r
> values_matrix <- matrix(values, ncol=3, nrow=2)
> values_matrix
   [,1] [,2] [,3]  
[1,]  1  5  9  
[2,]  3  7 11  
```

We will create a matrix using `rbind()` and `cbind()` as follows:

```r
> x <- c(1,5,9)
> y <- c(3,7,11)
> m1  <- rbind(x, y)
> m1
   [,1] [,2] [,3]  
  x  1  5  9  
  y  3  7 11  
> m2 <- cbind(x,y)
> m2
  x  y
[1,] 1 3  
[2,] 5 7  
[3,] 9 11  
```
You can access elements of a matrix using its row and column number as follows:

```r
> values_matrix[2,2]
[1] 7
```

Alternatively, matrices and arrays are also indexed as a vector, so you could also get the value at $(2, y)$ using its index as follows:

```r
> values_matrix[4]
[1] 7
> coordinates[3]
[1] 3
```

Since matrices and arrays are indexed as a vector, you can use the `length()` function to determine how many elements are present in your matrix or array. This property comes in very handy when writing `for` loops as we will see later in this chapter in the Flow control section. Let's take a look at the `length` function:

```r
> length(coordinates)
[1] 32
```

The `length()` and `names()` functions have attributes with higher-dimensional generalizations. The `length()` function generalizes to `nrow()` and `ncol()` for matrices, and `dim()` for arrays. Similarly, `names()` can be generalized to `rownames()`, `colnames()` for matrices, and `dimnames()` for multidimensional arrays.

Note that `dimnames()` takes a list of character vectors corresponding to the names of each dimension of the array.

Let's take a look at the following functions:

```r
> ncol(values_matrix)
[1] 3
> colnames(values_matrix) <- c("Column_A", "Column_B", "Column_C")
> values_matrix
     Column_A Column_B Column_C
  [1,]       1       5       9
  [2,]       3       7      11
> dim(coordinates)
[1] 2 8 2
> dimnames(coordinates) <- list(c("alpha", "beta"), c("a", "b", "c", "d", "e", "f", "g", "h"), c("X", "Y"))
> coordinates
   , , X
      a  b  c  d  e  f  g  h
   alpha 1  3  5  7  9 11 13 15
   beta  2  4  6  8 10 12 14 16
```
In addition to these properties, you can transpose a matrix using the `t()` function and an array using the `aperm()` function that is part of the `abind` package. Another interesting tool of the `abind` package is the `abind()` function that allows you to combine arrays the same way you would combine vectors into a matrix using the `cbind()` or `rbind()` functions.

You can test whether your object is an array or matrix using the `is.matrix()` and `is.array()` functions, which will return `TRUE` or `FALSE`; otherwise, you can determine the number of dimensions of your object with `dim()`. Lastly, you can convert an object into a matrix or array using the `as.matrix()` or `as.array()` function. This may come in handy when working with packages or functions that require that an object be of a particular class, that is, a matrix or an array. Be aware that even a simple vector can be stored in multiple ways, and depending on the class of the object and function they will behave differently. Quite frequently, this is a source of programming errors when people use built-in or package functions and don’t check the class of the object the function requires to execute the code.

The following is an example that shows that the `c(1, 6, 12)` vector can be stored as a matrix with a single row or column, or a one-dimensional array:

```r
> x <- c(1, 6, 12)
> str(x)
  num [1:3] 1 6 12 #numeric vector
> str(matrix(x, ncol=1))
  num [1:3, 1] 1 6 12 #matrix of a single column
> str(matrix(x, nrow=1))
  num [1, 1:3] 1 6 12 #matrix of a single row
> str(array(x, 3))
  num [1:3(1d)] 1 6 12 #a 1-dimensional array
```

**Data frames**

The most common way to store data in R is through data frames and, if used correctly, it makes data analysis much easier, especially when dealing with categorical data. Data frames are similar to matrices, except that each column can store different types of data. You can construct data frames using the `data.frame()` function or convert an R object into a data frame using the `as.data.frame()` function as follows:

```r
> students <- c("John", "Mary", "Ethan", "Dora")
> test.results <- c(76, 82, 84, 67)
```
> test.grade <- c("B", "A", "A", "C")
> thirdgrade.class.df <- data.frame(students, test.results, test.grade)
> thirdgrade.class.df
students test.results test.grade
John 76 B
Mary 82 A
Ethan 84 A
Dora 67 C
> # see page 18 for how values_matrix was generated
> values_matrix.df <- as.data.frame(values_matrix)
> values_matrix.df
Column_A Column_B Column_C
1 1 5 9
2 3 7 11

Data frames share properties with matrices and lists, which means that you can use colnames() and rownames() to add the attributes to your data frame. You can also use ncol() and nrow() to find out the number of columns and rows in your data frame as you would in a matrix. Let's take a look at an example:

> rownames(values_matrix.df) <- c("Row_1", "Row_2")
> values_matrix.df
Column_A Column_B Column_C
Row_1 1 5 9
Row_2 3 7 11

You can append a column or row to data.frame using rbind() and cbind(), the same way you would in a matrix as follows:

> student_ID <- c("012571", "056280", "096493", "032567")
> thirdgrade.class.df <- cbind(thirdgrade.class.df, student_ID)
> thirdgrade.class.df
students test.results test.grade student_ID
John 76 B 012571
Mary 82 A 056280
Ethan 84 A 096493
Dora 67 C 032567

However, you cannot create data.frame from cbind() unless one of the objects you are trying to combine is already a data frame because cbind() creates matrices by default. Let's take a look at the following function:

> thirdgrade.class <- cbind(students, test.results, test.grade, student_ID)
> thirdgrade.class
students test.results test.grade student_ID
Another thing to be aware of is that R automatically converts character vectors to factors when it creates a data frame. Therefore, you need to specify that you do not want strings to be converted to factors using the `stringsAsFactors` argument in the `data.frame()` function, as follows:

```r
> str(thirdgrade.class.df)
'data.frame': 4 obs. of 4 variables:
$ students : Factor w/ 4 levels "Dora","Ethan",..: 3 4 2 1
$ test.results: num 76 82 84 67
$ test.grade : Factor w/ 3 levels "A","B","C": 2 1 1 3
$ student_ID : Factor w/ 4 levels "012571","032567",..: 1 3 4 2
> thirdgrade.class.df <- data.frame(students, test.results, test.grade, student_ID, stringsAsFactors=FALSE)
> str(thirdgrade.class.df)
'data.frame': 4 obs. of 4 variables:
$ students : chr "John" "Mary" "Ethan" "Dora"
$ test.results: num 76 82 84 67
$ test.grade : chr "B" "A" "A" "C"
$ student_ID : chr "012571" "056280" "096493" "032567"
```

You can also use the `transform()` function to specify which columns you would like to set as character using the `as.character()` or `as.factor()` functions. This is because each row and column can be seen as an atomic vector. Let's take a look at the following functions:

```r
> modified.df <- transform(thirdgrade.class.df, test.grade = as.factor(test.grade))
> str(modified.df)
'data.frame': 4 obs. of 4 variables:
$ students : chr "John" "Mary" "Ethan" "Dora"
$ test.results: num 76 82 84 67
$ test.grade : Factor w/ 3 levels "A","B","C": 2 1 1 3
$ student_ID : chr "012571" "056280" "096493" "032567"
```

You can access elements of a data frame as you would in a matrix using the row and column position as follows:

```r
> modified.df[3, 4]
[1] "096493"
```
You can access a full column or row by leaving the row or column index empty, as follows:

```r
> modified.df[, 1]
[1] "John" "Mary" "Ethan" "Dora"
#Notice the command returns a vector
> str(modified.df[,1])
chr [1:4] "John" "Mary" "Ethan" "Dora"
> modified.df[1:2,]
students test.results test.grade student_ID
1 John 76 B 012571
2 Mary 82 A 056280
#Notice the command now returns a data frame
> str(modified.df[1:2,])
'data.frame': 2 obs. of 4 variables:
$ students : chr  "John" "Mary"
$ test.results: num  76 82
$ test.grade : Factor w/ 3 levels "A","B","C": 2 1
$ student_ID : chr  "012571" "056280"
```

Unlike matrices, you can also access a column by using its `object_name$column_name` attribute, as follows:

```r
> modified.df$test.results
[1] 76 82 84 67
```

## Loading data into R

There are several ways to load data into R. The most common way is to enter data using the `read.table()` function or one of its derivatives, `read.csv()` for the `.csv` files, or `read.delim()` for `.txt` files. You can also directly upload Excel data in the `.xls` or `.xlsx` format using the `gdata` or `XLConnect` package. Other file formats such as Minitab Portable Worksheet (.mtp) and SPSS (.spss) files can also be opened using the `foreign` package.

To download a package from within R, you can use the `install.packages()` function as follows:

```r
> install.packages(pkgname.tar.gz, repos = NULL, type = "source")
```

Next, load the package (otherwise known as a `library`) using the `library()` or `require()` function. The `require()` function is designed to use in functions because it returns `FALSE` and a warning message, instead of the error message that the `library()` returns when the package is missing. You only need to load a package once per R session.
The first thing to do before loading a file is to make sure that R is in the right working directory. You can see where R will read and save files, by default, using the getwd() function. Then, you can change it using the setwd() function. You should use the full path when setting the working directory because it is easier to avoid unwanted error messages such as Error in setwd("new_directory") : cannot change working directory.

For example, execute the following function on a Mac operating system:

```r
> getwd()
[1] "/Users/johnsonR/
> setwd("/Users/johnsonR/myDirectory")
```

To work with data in the C: drive in the myDirectory folder on a Windows version of R, you will need to set the working directory as follows:

```r
> setwd("C:/myDirectory")
```

Then, you can use the read.table() function to load your data as follows:

```r
#To specify that the file is a tab delimited text file we use the sep argument with "\t"
> myData.df <- read.table("myData.txt", header=TRUE, sep="\t")
> myData.df
   A  B  C
1 12  6  8
2  4  9  2
3  5 13  3
```

Alternatively, you could use the read.delim() function instead as follows:

```r
> read.delim("myData.txt", header=TRUE)
   A  B  C
1 12  6  8
2  4  9  2
3  5 13  3
> myData2.df <- read.csv("myData.csv", header=FALSE)
> myData2.df
  V1 V2 V3
1  A  B  C
2 12  6  8
3  4  9  2
4  5 13  3
```
Programming with R

By default, these functions return data frames with all string-containing columns converted to factors unless you set stringsAsFactors=FALSE in read.table(), read.delim(), and read.csv(). Let's take a look at an example:

```r
> str(myData2.df)
'data.frame': 4 obs. of 3 variables:
$ V1: Factor w/ 4 levels "12","4","5","A": 4 1 2 3
$ V2: Factor w/ 4 levels "13","6","9","B": 4 2 3 1
$ V3: Factor w/ 4 levels "2","3","8","C": 4 3 1 2
> myData2.df <- read.csv("myData.csv", header=FALSE, stringsAsFactors=FALSE)
> str(myData2.df)
'data.frame': 4 obs. of 3 variables:
$ V1: chr "A" "12" "4" "5"
$ V2: chr "B" "6" "9" "13"
$ V3: chr "C" "8" "2" "3"
```

To upload Excel sheets using the gdata package, you load the package into R and then use the read.xls() function as follows:

```r
> library("gdata")
> myData.df <- read.xls("myData.xlsx", sheet=1) #also uploads .xls files and returns a data frame
```

Alternatively, you could upload a complete workbook and read the worksheets separately using the XLConnect package as follows:

```r
> library("XLConnect")
> myData.workbook <- loadWorkbook("myData.xlsx")
> myData3.df <- readWorksheet(myData.workbook, sheet="Sheet1")
```

To read the .mtp and .spss files, you will first load the foreign package, and then use the read.mtp() and read.spss() functions. By default, these functions return a list of components so you will have to convert the data into a data frame afterwards. Alternatively, for .spss files, the read.spss() function has a to.data.frame argument that allows it to return a data frame instead.

```r
> myData4.df <- read.spss("myfile.spss", to.data.frame=TRUE)
```
Saving data frames

To save an object, preferably a matrix or data frame, you can write a .txt file or a file using another delimiter using the write.table() function. You can choose to include row.names and col.names by setting these arguments to TRUE. The output file will be saved to your current directory. Note that the write.table() function often saves character vectors with quotation marks in the output file. So, I also suggest that you set the quote argument to FALSE to avoid seeing quotation marks should you open the file with a text editor. Let's take a look at a few examples:

```r
> write.table(myData.df, file="savedata_file.txt", quote = FALSE, sep = "\t", row.names=TRUE, col.names=TRUE, append=FALSE)
```

By default, there is no column name for a column of row names. So your output would look like this:

```
V1 V2 V3
1 A  B  C
2 12 6  8
3 4  9  2
4 5 13 3
```

To correct this problem to view in a spreadsheet viewer such as Excel, you can write the table setting as col.names=NA and row.names=TRUE, as follows:

```r
> write.table(myData.df, file="savedata_file.txt", quote = FALSE, sep = "\t", col.names = NA, row.names = TRUE, append=FALSE)
```

```
V1 V2 V3
1 A  B  C
2 12 6  8
3 4  9  2
4 5 13 3
```

Alternatively, you could use the write.csv() function, which has col.names=NA and row.names=TRUE set as defaults:

```r
> write.csv(myData.df, file = "savedata_file.csv") #same output as above
```

If you would like to save a series of data frames in an Excel workbook, we recommend that you use the WriteXLS package, which greatly simplifies the task. Here is an example of the code you could use to save two data frames (df1 and df2) as two separate worksheets with the sheet names set as "df1_results" and "df2_results" in a file called combined_dfs_workbook.xls:

```r
> library("WriteXLS")
> dfs.tosave <- c("df1", "df2")
```
Programming with R

```r
sheets.tosave <- c("df1_results", "df2_results")
WriteXLS(dfs.tosave, ExcelFileName = "combined_dfs_workbook.xls",
         SheetNames = sheets.tosave)
```

You can also save and reload R objects for future sessions using the `dump()` and `source()` functions. For example, say you created several `list` objects containing important data for routine analysis. Saving a `list` object to a spreadsheet or `.txt` file can be difficult to reload afterwards, since most read functions return a data frame. A simpler way to proceed will be to save (or dump) the object to a file that R can reopen (source) in another session.

The following data shows how you can save that object:

```r
> dump("myData.df", "myData.R")
> # Or if you would like to save all objects in your session:
> dump(list=objects(), "all_objects.R")
```

The `myData.R` file created will contain all the commands necessary to recreate that object in a future session. At a later date, you can retrieve the data as follows:

```r
> source("mydata.R")
```

You can also use the `save()` and `load()` functions to save and retrieve your objects at a later time, as follows:

```r
> save(myData.df, file="myData.R")
> load("myData.R")
```

A good alternative to the `save()` and `load()` functions are the `saveRDS()` and `readRDS()` functions, respectively. The `saveRDS()` function doesn't save the object and its name; instead, it just saves a representation of the object. Therefore, when you retrieve the data with the `readRDS()` function, you will need to store it in an object. However, unlike the `save()` function, you can only save one object at a time with the `saveRDS()` function. For example, to save the `myData.df` object and retrieve it later, you can execute the following lines of code:

```r
# To save the object
> saveRDS(myData.df, "myData.rds")
# To load and save the object to a new object
> myData2 <- readRDS("myData.rds")
```
You can also redirect the R output to a file using the `sink(file="filename")` function as follows:

```r
> sink("data_session1.txt")
> x<-c(1,2,3)
> y <-c(4,5,6)
> #This is a comment
> x+y #Note the sum of x+y is redirected to data_session1.txt
```

To stop redirecting the output to the file and print a new output to the screen, just run the `sink()` function again without any arguments as follows:

```r
> sink()
> 3+4
[1] 7
```

When you open the `data_session1.txt` file, you will notice that only the result of the sum of `x+y` is saved to the file and not the commands or comments you entered.

The following is the output in the `data_session1.txt` file:

```
[1] 5 7 9
```

As you can see, comments and standard input aren't included in the output. Only the output is printed to the file specified in the `sink()` function.

---

**Basic plots and the ggplot2 package**

This section will review how to make basic plots using the built-in R functions and the `ggplot2` package to plot graphics.

Basic plots in R include histograms and scatterplots. To plot a histogram, we use the `hist()` function:

```r
> x <- c(5, 7, 12, 15, 35, 9, 5, 17, 24, 27, 16, 32)
> hist(x)
```
You can plot mathematical formulas with the `plot()` function as follows:

```r
> x <- seq(2, 25, by=1)
> y <- x^2 + 3
> plot(x, y)
```

The output is shown in the following plot:
You can graph a univariate mathematical function on an interval using the `curve()` function with the `from` and `to` arguments to set the left and right endpoints, respectively. The `expr` argument allows you to set a numeric vector or function that returns a numeric vector as an output, as follows:

```r
# For two figures per plot.
> par(mfrow=c(1,2))
> curve(expr=cos(x), from=0, to=8*pi)
> curve(expr=x^2, from=0, to=32)
```

In the following figure, the plot to your left shows the curve for \( \cos(x) \) and the plot to the right shows the curve for \( x^2 \). As you can see, using the `from` and `to` arguments, we can specify the \( x \) values to show in our figure.

You can also graph scatterplots using the `plot()` function. For example, we can use the `iris` dataset as part of R to plot `Sepal.Length` versus `Sepal.Width` as follows:

```r
> plot(iris$Sepal.Length, iris$Sepal.Width, main="Iris sepal length vs width measurements", xlab="Length", ylab="Width")
```
R has built-in functions that allow you to plot other types of graphics such as the `barplot()`, `dotchart()`, `pie()`, and `boxplot()` functions. The following are some examples using the `VADeaths` dataset:

```r
> VADeaths
    Rural Male Rural Female Urban Male Urban Female
50-54  11.7   8.7    15.4   8.4
55-59  18.1  11.7   24.3  13.6
60-64  26.9  20.3   37.0  19.3
65-69  41.0  30.9   54.6  35.1
70-74  66.0  54.3   71.1  50.0
> barplot(VADeaths, beside=TRUE, legend=TRUE, ylim=c(0, 100),
         ylab="Deaths per 1000 population", main="Death rate in VA")
```

#Requires that the data to plot be a vector or a matrix.
The output is shown in the following plot:

![Death rate in VA graph]

However, when working with data frames, it is often much simpler to use the `ggplot2` package to make a bar plot, since your data will not have to be converted to a vector or matrix first. However, you need to be aware that `ggplot2` often requires that your data be stored in a data frame in long format and not wide format.

The following is an example of data stored in wide format. In this example, we look at the expression level of the MYC and BRCA2 genes in two different cell lines, after these cells were treated with a vehicle-control, drug1 or drug2 for 48 hours:

```r
> geneExpdata.wide <- read.table(header=TRUE, text='
cell_line gene control drug1 drug2
CL1 MYC 20.4 15.9 1.5
CL2 MYC 26.9 18.1 6.7
CL1 BRCA2 109.5 18.1 89.8
CL2 BRCA2 121.3 24.4 120.2
')
```
The following is the data rewritten in long format:

```r
> geneExpdata.long <- read.table(header=TRUE, text='
cell_line gene variable value
1 CL1 MYC control 20.4
2 CL2 MYC control 26.9
3 CL1 BRCA2 control 109.5
4 CL2 BRCA2 control 121.3
5 CL1 MYC drug1 15.9
6 CL2 MYC drug1 18.1
7 CL1 BRCA2 drug1 18.1
8 CL2 BRCA2 drug1 24.4
9 CL1 MYC drug2 1.5
10 CL2 MYC drug2 6.7
11 CL1 BRCA2 drug2 89.8
12 CL2 BRCA2 drug2 120.2
')
```

Instead of rewriting the data frame by hand, this process can be automated using the `melt()` function, which is a part of the `reshape2` package:

```r
> library("reshape2")
> geneExpdata.long <- melt(geneExpdata.wide, id.vars=c("cell_line", "gene"), measure.vars=c("control", "drug1", "drug2"), variable.name="condition", value.name="gene_expr_value")
```

Now, we can plot the data using `ggplot2` as follows:

```r
> library("ggplot2")
> ggplot(geneExpdata.long, aes(x=gene, y=gene_expr_value)) + geom_bar(aes(fill=condition), colour="black", position=position_dodge(), stat="identity")
```
The output is shown in the following plot:

Another useful trick to know is how to add error bars to bar plots. Here, we have a summary data frame of standard deviation (sd), standard error (se), and confidence interval (ci) for the `geneExpdata.long` dataset as follows:

```r
> geneExpdata.summary <- read.table(header=TRUE, text='
gene condition N gene_expr_value        sd    se        ci
1 BRCA2   control 2          115.40  8.343860  5.90  74.96661
2 BRCA2     drug1 2           21.25  4.454773  3.15  40.02454
3 BRCA2     drug2 2          105.00 21.496046 15.20 193.13431
4   MYC   control 2           23.65  4.596194  3.25  41.29517
5   MYC     drug1 2           17.00  1.555635  1.10  13.97683
6   MYC     drug2 2            4.10  3.676955  2.60  33.03613
')
> #Note the plot is stored in the p object
> p<- ggplot(geneExpdata.summary, aes(x=gene, y= gene_expr_value, fill=condition)) + geom_bar(aes(fill=condition), colour="black", position=position_dodge(), stat="identity")
> #Define the upper and lower limits for the error bars
> limits <- aes(ymax = gene_expr_value + se, ymin= gene_expr_value - se)
> #Add error bars to plot
> p + geom_errorbar(limits, position=position_dodge(0.9), size=.3, width=.2)
```
The result is shown in the following plot:

Going back to the `VADeaths` example, we could also plot a Cleveland dot plot (dot chart) as follows:

```r
> dotchart(VADeaths, xlim=c(0, 75), xlab="Deaths per 1000", main="Death rates in VA")
```

Note that the built-in `dotchart()` function requires that the data be stored as a vector or matrix.
The result is shown in the following plot:

![Plot showing death rates in VA](image)

The following are some other graphics you can generate with built-in R functions:

You can generate pie charts with the `pie()` function as follows:

```r
> labels <- c("grp_A", "grp_B", "grp_C")
> pie_groups <- c(12, 26, 62)
> pie(pie_groups, labels, col=c("white", "black", "grey")) #Fig. 3B
```

You can generate box-and-whisker plots with the `boxplot()` function as follows:

```r
> boxplot(value ~ variable, data= geneExpdata.long, subset=gene == "MYC", ylab="expression value", main="MYC Expression by Condition", cex.lab=1.5, cex.main=1.5)
```

Note that unlike other built-in R graphing functions, the `boxplot()` function takes data frames as the input.
Using our cell line drug treatment experiment, we can graph MYC expression for all cell lines by condition. The result is shown in the following plot:

![MYC Expression by Condition](image1)

The following is another example using the iris dataset to plot Petal.Width by Species:

```r
> boxplot(Petal.Width ~ Species, data=iris, ylab="petal width", cex.lab=1.5, cex.main=1.5)
```

The result is shown in the following plot:

![Petal Width by Species](image2)
Flow control

In this section, we will review flow-control statements that you can use when programming with R to simplify repetitive tasks and make your code more legible. Programming with R involves putting together instructions that the computer will execute to fulfill a certain task. As you have noticed this far, R commands consist mainly of expressions or functions to be evaluated. Most programs are repetitive and depend on user input prior to executing a task. Flow-control statements are particularly important in this process because it allows you to tell the computer how many times an expression is to be repeated or when a statement is to be executed. In the rest of this chapter, we will go through flow-control statements and tips that you can use to write and debug your own programs.

The for() loop

The for(i in vector){commands} statement allows you to repeat the code written in brackets {} for each element (i) in your vector in parenthesis.

You can use for() loops to evaluate mathematical expressions. For example, the Fibonacci sequence is defined as a series of numbers in which each number is the sum of the two preceding numbers. We can get the first 15 numbers that make up the Fibonacci sequence starting from (1, 1), using the following code:

```
> # First we create a numeric vector with 15 elements to store the data generated.
> Fibonacci <- numeric(15)
> Fibonacci
[1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Next, we need to write down the code that will allow us to generate the Fibonacci sequence. If the first two elements of the sequence are (1, 1) and every subsequent number is the sum of the two preceding numbers, then the third element is \(1 + 1 = 2\) and the fourth element is \(1 + 2 = 3\), and so on.

So, let's add the two first elements of the Fibonacci sequence in our Fibonacci vector as shown:

```
> Fibonacci[1:2] <- c(1,1)
```
Next, let's create a for() loop, which will add the sum of the two preceding numbers indexed at i-2 and i-1 from i=3 to i=15 (the length of the Fibonacci numeric vector we initially created):

```r
> for(i in 3:length(Fibonacci)){Fibonacci[i] <- Fibonacci[i-2] + Fibonacci[i-1]}
> Fibonacci
[1]  1  1  2  3  5  8 13 21 34 55 89 144 233 377 610
```

In this example, the vector evaluated by the for() loop is `3:length(Fibonacci)`, but we could have also expressed the vector as `c(3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)` or `seq(3, 15, by=1)`. To simplify our code, we can create a separate vector to store the sequence and then write our for() loop as follows:

```r
> Fibonacci_terms <- seq(3, 15, by=1)
> for(i in Fibonacci_terms){Fibonacci[i] <- Fibonacci[i-2] + Fibonacci[i-1]}
```

You don't always have to use a numeric or integer vector when writing for() loops. For example, you can use a character vector in a for() loop to update strings in another vector as follows:

```r
> fruits <- c("apple", "pear", "grapes")
> other_fruits <- c("banana", "lemon")
> for(i in fruits){other_fruits <-c(other_fruits, i)}  #appends fruits to other_fruits vector
> other_fruits
[1] "banana" "lemon" "apple" "pear" "grapes"
```

**The apply() function**

A good alternative to the for() loop is the apply() function, which allows you to apply a function to a matrix or array by row, column, or both. For example, let's calculate the mean of a matrix by row using the apply() function. First, let's create a matrix as follows:

```r
> m1 <-matrix(1:12, nrow=3)
> m1
[1,]  1  4  7 10
[2,]  2  5  8 11
[3,]  3  6  9 12
```

```r
> m1 <-matrix(1:12, nrow=3)
> apply(m1, 1, mean)  #apply function to matrix by row
[1] 5  6.5 7.5
```
Chapter 1

The second argument of the `apply()` function is `MARGIN`, which allows you to specify whether the function should be applied by row with 1, by column with 2, or both with `c(1,2)`. Since we want to calculate the mean by row, we will use 1 for `MARGIN`, as follows:

```r
> meanByrow <- apply(m1, 1, mean)
> meanByrow
[1] 5.5 6.5 7.5
```

The last argument of the `apply()` function is `FUN`, which refers to the function to be applied to the matrix. In our last example, we used the `mean()` function. However, you can use any function including those you wish to write yourself. For example, let's apply the `x+3` function to each value in the matrix as follows:

```r
# Notice there is no comma between function(x) and x+3 when defining the function in apply()
> m1plus3 <- apply(m1, c(1,2), function(x) x+3)
> m1plus3
[1,]   4   7  10  13
[2,]   5   8  11  14
[3,]   6   9  12  15
```

In the event that you want to specify arguments of a function, you just need to add them after the function. For example, let's say you want to apply the mean function by column to a second matrix but this time by specifying the `na.rm` argument as `TRUE` instead of the default (`FALSE`). Let's take a look at that in the following example:

```r
> z <- c( 1, 4, 5, NA, 9,8, 3, NA)
> m2 <- matrix(z, nrow=4)
> m2
     [,1] [,2]
[1,]    1    9
[2,]    4    8
[3,]    5    3
[4,]   NA   NA
# Notice you need to separate the argument from its function with a comma
> meanByColumn <- apply(m2, 2, mean, na.rm=TRUE)
> meanByColumn
[1] 3.333333 6.666667
```
The if() statement

The `if(condition){commands}` statement allows you to evaluate a condition and if it returns `TRUE`, the code in brackets will be executed. You can add an `else {commands}` statement to your `if()` statement if you would like to execute a block of code if your condition returns `FALSE`:

```r
> x <- 4
> # we indent our code to make it more legible
> if(x < 10) {
  x <- x+4
  print(x)
}
[1] 8
```

If you have several conditions to test before running an `else {}` statement, you can use an `else if(condition){commands}` statement as follows:

```r
> x <- 1
> if(x == 2) {
  x <- x+4
  print("X is equal to 2, so I added 4 to it.")
} else if (x > 2) {
  print("X is greater than 2, so I did nothing to it.")
} else {
  x <- x -4
  print("X is not greater than or equal to 2, so I subtracted 4 from it.")
}
[1] "X is not greater than or equal to 2, so I subtracted 4 from it."
```

The while() loop

The `while(condition){commands}` statement allows you to repeat a block of code until the condition in the parenthesis returns `FALSE`. If we look back at our Fibonacci sequence example, we could have written our program using a `while()` loop instead, as follows:

First, we create two objects to store the first and second number of the Fibonacci sequence:

```r
> num1 <- 1
> num2 <- 1
```
Then, we create a numeric vector to contain the first two numbers of the Fibonacci sequence:

```r
> Fibonacci <- c(num1, num2)
```

Next, we create a \texttt{count} object to store the number of elements added to the \texttt{Fibonacci} vector. We start the count at 2 since the first two numbers have already been added to the \texttt{Fibonacci} vector as follows:

```r
> count <- 2 #set count to start from 2

> while(count < 15) {

#We update the count number so that we can track the number of times the loop is repeated.
count <- count +1

#Next we make sure to store the 2nd number in a new object before it is overwritten.
oldnum2 <- num2

#Then we calculate the next number in the Fibonacci sequence.
num2 <- num1 + num2

#Then we update the Fibonacci vector with the 2nd number each time the loop is repeated.
Fibonacci <- c(Fibonacci, num2)

#Lastly, we assign the 2nd number as the new first number to use in the next iteration of the loop.
num1 <- oldnum2

}

> Fibonacci

[1]  1  1  2  3  5  8 13 21 34 55 89 144 233 377 610
```

### The repeat{} and break statement

The \texttt{repeat\{}\texttt{commands}\} statement is similar to the \texttt{while()} loop except that you do not need to set a condition to test, and your code is repeated endlessly unless you include a \texttt{break} statement. Typically, a \texttt{repeat\{}\texttt{\}} statement includes an \texttt{if(condition) break} line, but this is not required. The \texttt{break} statement causes the loop to terminate immediately.
If we go back to our Fibonacci example, we could have written the code as follows:

```r
num1 <- 1
num2 <- 1
Fibonacci <- c(num1, num2)
count <- 2
repeat {
count <- count +1
oldnum2 <- num2
num2 <- num1 + num2
Fibonacci <- c(Fibonacci, num2)
num1 <- oldnum2
if (count >= 15) { break }
}
```

Functions

Functions are bits of code that perform a particular task and print or return its output to an object. Writing functions are particularly useful to avoid rewriting code over and over in your program; instead, you can write a function and every time you would like to perform that particular task, you can call that function. In fact, all the code we used so far in our examples call built-in or third-party R package functions.

For example, we ask for the mean of x using the following code:

```r
x <- c(2, 6, 7, 12)
mean(x)
```

```
[1] 6.75
```

In the preceding code, we are actually asking R to call the `mean()` function. Each function takes arguments. If you would like to know what arguments could be passed to a particular R function, you can consult the help page. There are several ways to access the help documentation in R. First, you can use the `help()` function as follows:

```r
help(mean)
```

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments

```
Chapter 1

x An R object. Currently there are methods for numeric/logical vectors and date, date-time, and time interval objects. Complex vectors are allowed for trim = 0, only.
trim the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
na.rm a logical value indicating whether NA values should be stripped before the computation proceeds.
... further arguments passed to or from other methods.
[...]

Alternatively, you can use the ? symbol to obtain the documentation page for the mean function as follows:

> ?mean #Returns the same output as above

Alternatively, you may also want to search all the help topics as shown in the following screenshot for the mean word with the ?? symbol as follows:

> ??mean

As you can see in the preceding screenshot, R returns a table of all the search results matching the word "mean" for all the packages you have installed on your computer.
The help page is very useful because it tells you what type of object the function takes as input and a list of all the arguments it takes. The help page also informs you of the default settings used for all the arguments the function takes. By consulting the help page for the `mean()` function, you learn that the default settings are `trim=0` and `na.rm=FALSE`. With `trim` set to 0, no observations or values are removed prior to calculating the mean, and with `na.rm` set to `FALSE`, all `NA` entries are not removed before calculating the mean. Consider the following example:

```r
> x <- c(2, 6, 7, 12, NA, NA)
> mean(x)
[1] NA
```

If we specify `na.rm=TRUE`, the `NA` entries are ignored as follows:

```r
> mean(x, na.rm=TRUE)
[1] 6.75
```

So far, we have been changing default parameters by explicitly specifying which arguments to change, that is, `na.rm=TRUE`. However, R also allows you to change default parameters using the argument position only. This means we can rewrite the last command as follows:

```r
> #notice "," is used to specify unchanged missing arguments in the order they appear in the function definition on the help page
> mean(x, ,TRUE)
[1] 6.75
```

This also holds true for the functions you may write as well. Let's write a simple function called `vectorContains()` to test whether a vector contains the number 3.

To define a function in R, we write the word `function` and our list of arguments contained in parenthesis () followed by curly braces that contains the sequence of commands we want our function to execute. For example, let's write a function to check whether the value 3 is present in a vector. Here are the steps we will take to write a function to check whether a value (in this case, 3) is present in an input vector:

1. We create a function called `vectorContains` and use an argument (variable) `value.to.check` to store the value we want to check.
2. We check that the input object type is numeric using the `is.numeric()` function.
3. We ensure that there are no missing (NA) values using the `any()` and `is.na()` functions. The `any()` function will check each entry and the `is.na()` function will return `TRUE` if `NA` is present. Because we want to return `TRUE` when there is no `NA` present instead of when an `NA` is present, we use the `!` sign before the `any(is.na())` command.
Chapter 1

4. We use an \texttt{if else \{} \texttt{statement} to return an error message if the vector isn't numeric and/or contains NA values using the \texttt{stop()} function.

5. We create an object \texttt{value.found} to keep track of whether the value to be checked is found. We initially set \texttt{value.found} to \texttt{FALSE} because we assume the value is not present.

6. We check each value of our input vector using a \texttt{for()} loop. If an element (\texttt{i}) of our vector matches \texttt{value.to.check}, we set \texttt{value.found} to "yes" and break out of the \texttt{for()} loop.

7. Depending on whether \texttt{value.found} is set to "yes" or "no", we return \texttt{TRUE} or \texttt{FALSE} as follows:

\begin{verbatim}
> vectorContains <- function(v1, value.to.check=3){
  if(is.numeric(v1) && !any(is.na(v1))) {
    value.found <- "no"
    for (i in v1){
      if(i == value.to.check) {
        value.found <- "yes"
        break
      }
    }
    if(value.found == "yes") {
      return(TRUE)
    } else {
      return(FALSE)
    }
  } else {
    #When it exits the function it will print the following error message
    stop("This function takes a numeric vector without NAs as input.")
  }
}
\end{verbatim}

Now, let's test our function as follows:

\begin{verbatim}
> x <- c(2, 6, 7, 12, NA, NA)

> vectorContains(x)
Error in vectorContains(x) :
  This function takes a numeric vector without NAs as input.
> y <- c(1, 4, 6, 8, 3, 12, 15)
> vectorContains(y)
[1] TRUE
\end{verbatim}
Suppose we want to test whether a vector contains the value 6 instead of 3, we can easily change the default `value.to.check` from 3 to 6, as follows:

```r
> vectorContains(y, 6)
[1] TRUE
> vectorContains(y, value.to.check=17)
[1] FALSE
```

Hopefully, in the preceding example, you can see that the beauty of writing functions instead of individual commands because you can reuse this function to check whether a vector contains any particular value. Moreover, by saving these lines of code to a text document (for example, `vectorfunction.R`), you can reload this function in a later session using the `source()` command instead of rewriting the function, as follows:

```r
> source("/PathToFile/vectorfunction.R")
```

### General programming and debugging tools

Since this chapter is meant to review R programming, I will not go into too much detail on how to write a program step by step, but I will present some general advice on how to write a successful program.

First, it is essential that you understand the problem because R will only do what you tell it to do. So if you don't have a clear picture of the problem, it's best you sit down and work out what you want your program to do and think about what R tools and/or packages are available to help you fulfill your task. Once you've explored the R functions and packages available to you to help address your question, you should simplify your problem by writing down general steps and functions you can use to solve your problem and then translate your general ideas into a detailed implementation.

A good strategy to adopt when working on a detailed implementation for a program is to use the "top-down" design approach, which consists of writing the whole program in a couple of steps like you would an essay outline. Then, expand each step with additional key steps and keep expanding until you have a full program. To save time and make your code more legible, I would suggest breaking up each of your key steps into functions, and then run and check each function iteratively. As a general rule of thumb, if your function starts to get really long, that is, dozens of line, I would suggest thinking of ways to break down that function into a bunch of smaller functions or "subfunctions", in the same way you would break down really long paragraphs into smaller ones when writing an essay.
The beauty of programming resides in the ability to write and reuse functions in several programs. By writing generic functions that fulfill specific tasks, you can reuse that code in another program by simply executing the following code:

```r
> source("someOtherfunctions.R")
```

The trickiest part of programming is finding and solving errors (debugging). The following is a list of some generic steps you can take when trying to solve a bug:

1. Recognize that your program has a bug. This can be easy when you get an error or warning message but harder when you get an output that is not the output expected or the true answer to your problem.
2. Make the bug reproducible. It is easier to fix a bug that you know how to trigger.
3. Identify the cause of the bug. For example, this can be a variable, not updating it the way you wanted it to in a function, or a condition statement that can never return `TRUE` as written. Other common causes of error for beginners include testing for a match (equality) by writing `if(x = 12)` instead of `if(x==12)`, or the inability of your code to deal with missing data (NA values).
4. Fix the error in your code and test whether you successfully fixed it.
5. Look for similar errors elsewhere in your code.

One trick you can use to help you tease out the cause of your error message is the `traceback()` function. For example, when we tried to the `vectorContains(x)`, we got the error message "This function takes a numeric vector as input." If someone wanted to see where the error message was coming from, they could run `traceback()` and get the location as follows:

```r
> traceback()
2: stop("This function takes a numeric vector as input.") at #38
1: vectorContains(x)
```

Other useful functions include the `browser()` and `debug()` functions. The `browser()` function allows you to pause the execution of your function, and examine or change local variables, and even execute other R commands. Let's inspect the `vectorContains()` function we wrote earlier with the `browser()` function as follows:

```r
> x <- c(2, 6, 7, 12, NA, NA)
> browser()
```
We have now entered the Browser mode.

Browse[1]> x <- c(1, 2, 3)
Browse[1]> vectorContains(x)
Error in vectorContains(x) :
  This function takes a numeric vector without NAs as input.
Browse[1]> x <- c(1, 2, 3)
Browse[1]> vectorContains(x)
[1] TRUE
Browse[1]> Q #To quit browser()

Note that the variable x we changed in the browser mode was stored to our workspace. So if we enter x after we quit, the values stored in browser mode will be returned, as follows:

> x
[1] 1 2 3

When we call the debug() function, we also enter the browser mode. This allows us to execute a single line of code at a time by entering n for next, continue to run the function by entering c, or quit the function by entering Q like in browser mode. Note that each time you call the function, you will enter the browser mode unless you run the undebug() function.

The following is an example using debug to inspect our vectorContains() function:

> debug(vectorContains)
> x <- c(1, 2, 3, 9)
> vectorContains(x)
debugging in: vectorContains(x)
debug at #1: {
  if (is.numeric(v1) && !any(is.na(v1))) {
    value.found <- "no"
    for (i in v1) {
      if (i == value.to.check) {
        value.found <- "yes"
        break
      }
    }
    if (value.found == "yes") {
      return(TRUE)
    }
  } else {
    return(FALSE)
else {
    stop("This function takes a numeric vector as input.")
}
}

Browse[2]> c
exiting from: vectorContains(x)
[1] TRUE
> undebug(vectorContains)
> vectorContains(x)
[1] TRUE

Notice that debug only enters the browser mode when you call the vectorContains function.

**Summary**

In this chapter, we saw how data is stored and accessed in R. We also discussed how to write functions. You should now be able to write and access data in vectors, arrays, and data frames, and load your data into R. We also learned how to make basic plots using built-in R functions and the ggplot2 package. You should also know how to use flow-control statements in your code and write your own functions and use built-in tools to troubleshoot your code.

Now that you have a foundation in R data structures, we will move on to statistical methods in the next chapter, where you will find out how to obtain useful statistical information from your dataset and fit your data to known probability distributions.
Where to buy this book