MODULE 1:

INTRODUCTION TO R

Introduction to R:

R is a programming language and a software system for computations and graphics. According to the R FAQ, it consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files." R was originally developed in 1992 by Ross Ihaka and Robert Gentleman at the University of Auckland in New Zealand. The R language is a "dialect" of the S language2 , which was developed (mainly) by John Chambers at Bell Laboratories. This software is currently maintained by the R Core Team, which consists of more than a dozen people, and includes Ihaka, Gentleman, and Chambers.

Many other people have contributed code to R since it was first released. R is open source; the source code for R is available under the GNU General Public License, meaning that users can modify, copy, and redistribute the software or derivatives, as long as the modified source code is made available. The software is regularly updated, but changes are usually not major.

Finding and installing R

The R Core Team maintains a network of servers that contains installation files and documentation on R, called the Comprehensive R Archive Network, or CRAN. You can access it through http: //cran.r-project.org/, or a Google search for CRAN R. R is available for Windows, Mac, and Unix–like operating systems. Installation files and instructions can be downloaded from the CRAN site by selecting one of the download links at the top. Although the graphical user interfaces (GUIs) and their menus differ across systems (if present at all), the R commands do not. R has been extended by users, and thousands of add-on packages (referred to just as "packages" by R users), which are modules of R functions and possibly data, usually related to a particular purpose, are available for free online.

How to run R

R operates in two modes: interactive and batch. The one typically used is interactive mode. In this mode, you type in commands, R displays results, you type in more commands, and so on. On the other hand, batch mode does not require interaction with the user. It's useful for production jobs, such as when a program must be run periodically, say once per day, because you can automate the process.

R Sessions and Functions

R Sessions:

- An R session refers to the period of time during which R is open and active.
- When you start R, it initiates a new session, allowing you to interact with the R environment.
- During an R session, you can execute commands, load datasets, create plots, define functions, and perform various data analysis tasks.
- R sessions can be terminated by quitting R or closing the R console or IDE.

Functions in R:

- In R, a function is a block of code that performs a specific task and returns a result.
- Functions in R can be built-in functions provided by R packages or user-defined functions created by the user.
- Built-in functions cover a wide range of tasks, such as mathematical operations, data manipulation, statistical analysis, and plotting.
- User-defined functions allow users to create custom functions tailored to their specific needs.

- Functions in R follow a syntax where the function name is followed by parentheses containing optional arguments.
- To use a function in R, you typically call it by its name and provide the required arguments within the parentheses.

Examples of R Functions:

mean(): Calculates the mean of a numeric vector.

plot(): Creates a plot based on the provided data.

read.csv(): Reads data from a CSV file into a data frame.

lm(): Fits a linear regression model to the data.

print(): Prints the output to the console.

sum(): Calculates the sum of elements in a vector.

apply(): Applies a function to rows or columns of a matrix or data frame.

Defining Custom Functions:

- Users can define their own functions in R using the function() keyword followed by the function name and optional arguments.
- The body of the function contains the code to be executed, and the function may return a value using the return() statement.
- Custom functions allow users to encapsulate repetitive tasks, improve code readability, and promote code reusability.

Basic Math

The functions available in R for manipulating data are too many to be listed here. One can find all the basic mathematical functions (log, exp, log10, log2, sin, cos, tan, asin, acos, atan, abs, sqrt, . . .), special functions (gamma, digamma, beta, besselI, . . .), as well as diverse functions useful in statistics. Some of these functions are listed in the following table.

Function	Description
sum(x)	sum of the elements of x
prod(x)	product of the elements of x
max(x)	maximum of the elements of x
min(x)	minimum of the elements of x
which.max(x)	returns the index of the greatest element of x
which.min(x)	returns the index of the smallest element of x
range(x)	id. than c(min(x), max(x))
length(x)	number of elements in x
mean(x)	mean of the elements of x
median(x)	median of the elements of x
var(x) or cov(x)	variance of the elements of x (calculated on n − 1); if x is a matrix or a data frame, the variance-covariance matrix is calculated
cor(x)	correlation matrix of x if it is a matrix or a data frame (1 if x is a vector)
var(x, y) or cov(x, y)	covariance between x and y, or between the columns of x and those of y if they are matrices or data frames
cor(x, y)	linear correlation between x and y, or correlation matrix if they are matrices or data frames

These functions return a single value (thus a vector of length one), except range which returns a vector of length two, and var, cov, and cor which may return a matrix.

Variables

A variable that is visible only within a function body is said to be local to that function. In oddcount(), k and n are local variables. They disappear after the function returns:

> oddcount(c(1,2,3,7,9))

[1] 4

> n

Error: object 'n' not found

It's very important to note that the formal parameters in an R function are local variables. Suppose we make the following function call:

> z <- c(2,6,7)

> oddcount(z)

Now suppose that the code of oddcount() changes x. Then z would not change. After the call to oddcount(), z would have the same value as before. To evaluate a function call, R copies each actual argument to the corresponding local parameter variable, and changes to that variable are not visible outside the function.

Variables created outside functions are global and are available within functions as well. Here's an example:

> f <- function(x) return(x+y)

> y <- 3

> f(5)

[1] 8

Here y is a global variable. A global variable can be written to from within a function by using R's superassignment operator, <<-.

Default Arguments

R also makes frequent use of default arguments. Consider a function definition like this:

> g <- function(x,y=2,z=T) { ... }

Data types

Understanding the different types of data in R can be a bit confusing. Use of the class, mode, or even typeof functions to check data types can be helpful (or, in some cases, confusing). The following discussion will attempt to accurately describe what I think you need to know46 . The simplest data structure in R is a vector, which is a simply an ordered collection of elements. All of the data objects we have worked with so far are vectors. There are several kinds of vectors, which differ only in the type of data they contain–we'll use them to demonstrate the different types of data in R. There are four common types of data that can be held in vectors47: numeric, integer, character, and logical.

Numeric data

```
x <- 10.2

x

# [1] 10.2
```

Integer data

```
y <- 1:5

y

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

Character data

```
name <- "Johnny Appleseed"

name

# [1] "Johnny Appleseed"
```

Any time character data are entered directly into a console, you must surround individual elements with quotes. Otherwise, R will look for an object.

```
name <- Johnny
```

Either single or double quotes can be used in R49. When character data are read into R from a file, quotes are not necessary50 . Logical data contain only three values: TRUE, FALSE, or NA (NA indicates a missing value–more on this later)

```
a <- TRUE

a

# [1] TRUE
```

Note that there are no quotes around logical values (quotes would make them character data). R will return logical data for any relational expression submitted to it.

```
4 < 2

# [1] FALSE
```

or

```
b <- 4 < 2

b

# [1] FALSE
```

There are several functions that can be used to identify data type. In most cases, class or mode is the best bet. The class (i.e., the value returned by class) of an R object is meant to be “the official public view” [3, p 141] and it determines how generic functions operate on an object52. This is the same “class” that is referred to above in the discussion on generic functions.

```
class(x)

# [1] "numeric"

class(y)

# [1] "integer"
```

Technically what these output mean is that the objects are: a numeric vector, an integer vector, a character vector, and a logical vector, respectively. For more complex objects, class doesn't return info on whether data are numeric, character, etc., but describes only the structure of the object. Unfortunately, this approach can be a bit confusing. The mode and class functions return identical values for the three examples above, but this is not so for more complex objects, where mode gives a description of the way an object is stored.

```
mode(x)

# [1] "numeric"

mode(name)

# [1] "character"

mode(a)

# [1] "logical"
```

Vectors

In many programming languages, vector variables are considered different from scalars, which are single-number variables. Consider the following C code, for example:

```
int x;

int y[3];
```

This requests the compiler to allocate space for a single integer named

x and a three-element integer array (C terminology analogous to R's vector

type) named y. But in R, numbers are actually considered one-element vectors, and there is really no such thing as a scalar.

R variable types are called modes. Recall from Chapter 1 that all elements in a vector must have the same mode, which can be integer, numeric

(floating-point number), character (string), logical (Boolean), complex,

and so on. If you need your program code to check the mode of a variable x,

you can query it by the call typeof(x).

Unlike vector indices in ALGOL-family languages, such as C and Python,

vector indices in R begin at 1This requests the compiler to allocate space for a single integer named x and a three-element integer array (C terminology analogous to R's vector type) named y. But in R, numbers are actually considered one-element vectors, and there is really no such thing as a scalar. R variable types are called modes. Recall from Chapter 1 that all elements in a vector must have the

same mode, which can be integer, numeric (floating-point number), character (string), logical (Boolean), complex, and so on. If you need your program code to check the mode of a variable x, you can query it by the call typeof(x). Unlike vector indices in ALGOL-family languages, such as C and Python, vector indices in R begin at 1.

Adding and Deleting Vector Elements

Vectors are stored like arrays in C, contiguously, and thus you cannot insert or delete elements—something you may be used to if you are a Python programmer. The size of a vector is determined at its creation, so if you wish to add or delete elements, you'll need to reassign the vector. For example, let's add an element to the middle of a four-element vector:

```
> x <- c(88,5,12,13)

> x <- c(x[1:3],168,x[4]) # insert 168 before the 13

> x

[1] 88 5 12 168 13
```

Here, we created a four-element vector and assigned it to x. To insert a new number 168 between the third and fourth elements, we strung together the first three elements of x, then the 168, then the fourth element of x. This creates a new five-element vector, leaving x intact for the time being. We then assigned that new vector to x. In the result, it appears as if we had actually changed the vector stored in x, but really we created a new vector and stored that vector in x. This difference may seem subtle, but it has implications. For instance, in some cases, it may restrict the potential for fast performance in R.

Obtaining the Length of a Vector

You can obtain the length of a vector by using the length() function:

```
> x <- c(1,2,4)

> length(x)

[1] 3
```

In this example, we already know the length of x, so there really is no need to query it. But in writing general function code, you'll often need to know the lengths of vector arguments. For instance, suppose that we wish to have a function that determines the index of the first 1 value in the function's vector argument (assuming we are sure there is such a value). Here is one (not necessarily efficient) way we could write the code:

```
first1 <- function(x) {

for (i in 1:length(x)) {

if (x[i] == 1) break # break out of loop

}

return(i)

}
```

Without the length() function, we would have needed to add a second argument to first1(), say naming it n, to specify the length of x. Note that in this case, writing the loop as follows won't work:

for (n in x)

The problem with this approach is that it doesn't allow us to retrieve the index of the desired element. Thus, we need an explicit loop, which in turn requires calculating the length of x. One more point about that loop: For careful coding, you should worry that length(x) might be 0. In such a case, look what happens to the expression 1:length(x) in our for loop:

```
> x <- c()
> x
NULL
> length(x)
[1] 0
> 1:length(x)
[1] 1 0
```

Our variable i in this loop takes on the value 1, then 0, which is certainly not what we want if the vector x is empty.

Advanced Data Structures

R has a variety of data structures. Here, we will sketch some of the most frequently used structures to give you an overview of R before we dive into the details. This way, you can at least get started with some meaningful examples, even if the full story behind them must wait.

Vectors, the R Workhorse

The vector type is really the heart of R. It's hard to imagine R code, or even an interactive R session, that doesn't involve vectors. The elements of a vector must all have the same mode, or data type. You can have a vector consisting of three character strings (of mode character) or three integer elements (of mode integer), but not a vector with one integer element and two character string elements.

Scalars

Scalars, or individual numbers, do not really exist in R. As mentioned earlier, what appear to be individual numbers are actually one-element vectors. Consider the following:

```
> x <- 8
> x
[1] 8
```

Recall that the [1] here signifies that the following row of numbers begins with element 1 of a vector—in this case, x[1]. So you can see that R was indeed treating x as a vector, albeit a vector with just one element.

Character Strings

Character strings are actually single-element vectors of mode character, (rather than mode numeric):

```
> x <- c(5,12,13)
> x
[1] 5 12 13
> length(x)
[1] 3
> mode(x)
[1] "numeric"
> y <- "abc"
> y
[1] "abc"
> length(y)
[1] 1
> mode(y)
[1] "character"
> z <- c("abc","29 88")
> length(z)
[1] 2
> mode(z)
[1] "character"
```

Data Frames

On a technical level, a data frame is a list, with the components of that list being equal-length vectors. Actually, R does allow the components to be other types of objects, including other data frames. This gives us heterogeneous–data analogs of arrays in our analogy. But this use of data frames is rare in practice, and in this book, we will assume all components of a data frame are vectors. In this chapter, we'll present quite a few data frame examples, so you can become familiar with their variety of uses in R.

Creating Data Frames

To begin, let's take another look at our simple data frame example

```
> kids <- c("Jack","Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages,stringsAsFactors=FALSE)
> d # matrix-like viewpoint
```

kids ages

1 Jack 12

2 Jill 10

The first two arguments in the call to data.frame() are clear: We wish to produce a data frame from our two vectors: kids and ages. However, that third argument, stringsAsFactors=FALSE requires more comment. If the named argument stringsAsFactors is not specified, then by default, stringsAsFactors will be TRUE. (You can also use options() to arrange the opposite default.) This means that if we create a data frame from a character vector—in this case, kids—R will convert that vector to a factor. Because our work with character data will typically be with vectors rather than factors, we'll set stringsAsFactors to FALSE.

Accessing Data Frames

Now that we have a data frame, let's explore a bit. Since d is a list, we can access it as such via component index values or component names:

> d[[1]]

[1] "Jack" "Jill"

> d$kids

[1] "Jack" "Jill"

But we can treat it in a matrix-like fashion as well. For example, we can view column 1:

> d[,1]

[1] "Jack" "Jill"

This matrix-like quality is also seen when we take d apart using str():

> str(d)

'data.frame': 2 obs. of 2 variables:

$ kids: chr "Jack" "Jill"

$ ages: num 12 10

R tells us here that d consists of two observations—our two rows—that store data on two variables—our two columns.

Consider three ways to access the first column of our data frame above: d[[1]], d[,1], and d$kids. Of these, the third would generally considered to be clearer and, more importantly, safer than the first two. This better identifies the column and makes it less likely that you will reference the wrong column. But in writing general code—say writing R packages—matrix-like notation d[,1] is needed, and it is especially handy if you are extracting subdata frames.

Extended Example: Regression Analysis of Exam Grades Continued

Recall our course examination data set in Section 1.5. There, we didn't have a header, but for this example we do, and the first few records in the file now are as follows:

"Exam 1" "Exam 2" Quiz

2.0	3.3	4.0
3.3	2.0	3.7
4.0	4.0	4.0
2.3	0.0	3.3
2.3	1.0	3.3
3.3	3.7	4.0

As you can see, each line contains the three test scores for one student. This is the classic two-dimensional file notion, like that alluded to in the preceding output of str(). Here, each line in our file contains the data for one observation in a statistical data set. The idea of a data frame is to encapsulate such data, along with variable names, into one object. Notice that we have separated the fields here by spaces. Other delimiters may be specified, notably commas for comma-separated value (CSV) files (as you'll see in Section 5.2.5). The variable names specified in the first record must be separated by the same delimiter as used for the data, which is spaces in this case. If the names themselves contain embedded spaces, as we have here, they must be quoted. We read in the file as before, but in this case we state that there is a header record:

```
examsquiz <- read.table("exams",header=TRUE)
```

Lists:

As with vectors and matrices, one common operation with lists is indexing. List indexing is similar to vector and matrix indexing but with some major differences. And like matrices, lists have an analog for the apply() function. We'll discuss these and other list topics, including ways to take lists apart, which often comes in handy.

Creating Lists

Technically, a list is a vector. Ordinary vectors—those of the type we've been using so far in this book—are termed atomic vectors, since their components cannot be broken down into smaller components. In contrast, lists are referred to as recursive vectors. For our first look at lists, let's consider an employee database. For each employee, we wish to store the name, salary, and a Boolean indicating union membership. Since we have three different modes here—character, numeric, and logical—it's a perfect place for using lists. Our entire database might then be a list of lists, or some other kind of list such as a data frame, though we won't pursue that here. We could create a list to represent our employee, Joe, this way:

```
j <- list(name="Joe", salary=55000, union=T)
```

We could print out j, either in full or by component:

```
> j
$name
[1] "Joe"
$salary
[1] 55000
```

$union

[1] TRUE

Actually, the component names—called tags in the R literature—such as salary are optional. We could alternatively do this:

> jalt <- list("Joe", 55000, T)

> jalt

[[1]]

[1] "Joe"

[[2]]

[1] 55000

[[3]]

[1] TRUE

However, it is generally considered clearer and less error-prone to use names instead of numeric indices. Names of list components can be abbreviated to whatever extent is possible without causing ambiguity:

> j$sal

[1] 55000

Since lists are vectors, they can be created via vector():

> z <- vector(mode="list")

> z[["abc"]] <- 3

> z

$abc

[1] 3

General List Operations

Now that you've seen a simple example of creating a list, let's look at how to access and work with lists.

List Indexing

You can access a list component in several different ways:

> j$salary

[1] 55000

> j[["salary"]]

[1] 55000

```
> j[[2]]

[1] 55000
```

We can refer to list components by their numerical indices, treating the list as a vector. However, note that in this case, we use double brackets instead of single ones. So, there are three ways to access an individual component c of a list lst and return it in the data type of c:

lst$c

- lst[["c"]]
- lst[[i]], where i is the index of c within lst

Each of these is useful in different contexts, as you will see in subsequent examples. But note the qualifying phrase, "return it in the data type of c." An alternative to the second and third techniques listed is to use single brackets rather than double brackets:

lst["c"]

lst[i], where i is the index of c within lst

Both single-bracket and double-bracket indexing access list elements in vector-index fashion. But there is an important difference from ordinary (atomic) vector indexing. If single brackets [ ] are used, the result is another list—a sublist of the original. For instance, continuing the preceding example, we have this:

```
> j[1:2]

$name

[1] "Joe"

$salary

[1] 55000

> j2 <- j[2]

> j2

$salary

[1] 55000

> class(j2)

[1] "list"

> str(j2)

List of 1

$ salary: num 55000
```

The subsetting operation returned another list consisting of the first two components of the original list j. Note that the word returned makes sense here, since index brackets are functions. This is similar to other cases you've seen for operators that do not at first appear to be functions, such as +. By

contrast, you can use double brackets [[ ]] for referencing only a single component, with the result having the type of that component.

```
> j[[1:2]]
Error in j[[1:2]] : subscript out of bounds
> j2a <- j[[2]]
> j2a
[1] 55000
> class(j2a)
[1] "numeric"
```

Adding and Deleting List Elements

The operations of adding and deleting list elements arise in a surprising number of contexts. This is especially true for data structures in which lists form the foundation, such as data frames and R classes. New components can be added after a list is created.

```
> z <- list(a="abc",b=12)
> z
$a
[1] "abc"
$b
[1] 12
> z$c <- "sailing" # add a c component
> # did c really get added?
> z
$a
[1] "abc"
$b
[1] 12
$c
[1] "sailing"
```

Matrices and Arrays:

Matrices are special cases of a more general R type of object: arrays. Arrays can be multidimensional. For example, a three-dimensional array would consist of rows, columns, and layers, not just rows and columns as in the matrix case. Most of this chapter will concern matrices, but we will briefly discuss higher-dimensional arrays in the final section. Much of R's power comes from the various operations

you can perform on matrices. We'll cover these operations in this chapter, especially those analogous to vector subsetting and vectorization.

Creating Matrices

Matrix row and column subscripts begin with 1. For example, the upper-left corner of the matrix a is denoted a[1,1]. The internal storage of a matrix is in column-major order, meaning that first all of column 1 is stored, then all of column 2, and so on.

One way to create a matrix is by using the matrix() function:

```
> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)
> y
[,1] [,2]
[1,] 1 3
[2,] 2 4
```

Here, we concatenate what we intend as the first column, the numbers 1 and 2, with what we intend as the second column, 3 and 4. So, our data is (1,2,3,4). Next, we specify the number of rows and columns. The fact that R uses column-major order then determines where these four numbers are put within the matrix. Since we specified the matrix entries in the preceding example, and there were four of them, we did not need to specify both ncol and nrow; just nrow or ncol would have been enough. Having four elements in all, in two rows, implies two columns:

```
> y <- matrix(c(1,2,3,4),nrow=2)
> y
[,1] [,2]
[1,] 1 3
[2,] 2 4
```

Note that when we then print out y, R shows us its notation for rows and columns. For instance, [,2] means the entirety of column 2, as can be seen in this check:

```
> y[,2]
[1] 3 4
```

Another way to build y is to specify elements individually:

```
> y <- matrix(nrow=2,ncol=2)
> y[1,1] <- 1
> y[2,1] <- 2
> y[1,2] <- 3
> y[2,2] <- 4
> y
```

[,1] [,2]

[1,] 1 3

[2,] 2 4

General Matrix Operations

Now that we've covered the basics of creating a matrix, we'll look at some common operations performed with matrices. These include performing linear algebra operations, matrix indexing, and matrix filtering.

Performing Linear Algebra Operations on Matrices

You can perform various linear algebra operations on matrices, such as matrix multiplication, matrix scalar multiplication, and matrix addition. Using y from the preceding example, here is how to perform those three operations:

>y%*% y # mathematical matrix multiplication

[,1] [,2]

[1,] 7 15

[2,]10 22

> 3*y # mathematical multiplication of matrix by scalar

[,1] [,2]

[1,] 3 9

[2,] 6 12

> y+y # mathematical matrix addition

[,1] [,2]

[1,] 2 6

[2,] 4 8

Matrix Indexing

The same operations we discussed for vectors in Section 2.4.2 apply to matrices as well. Here's an example:

> z

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

[1,] 1 1 1

[2,] 2 1 0

[3,] 3 0 1

[4,] 4 0 0

> z[,2:3]

[,1] [,2]

[1,] 1 1

[2,] 1 0

[3,] 0 1

[4,] 0 0

Here, we requested the submatrix of z consisting of all elements with column numbers 2 and 3 and any row number. This extracts the second and third columns. Here's an example of extracting rows instead of columns:

> y

[,1] [,2]

[1,]11 12

[2,]21 22

[3,]31 32

> y[2:3,]

[,1] [,2]

[1,]21 22

[2,]31 32

> y[2:3,2]

[1] 22 32

Higher-Dimensional Arrays

In a statistical context, a typical matrix in R has rows corresponding to observations, say on various people, and columns corresponding to variables, such as weight and blood pressure. The matrix is then a two-dimensional data structure. But suppose we also have data taken at different times, one data point per person per variable per time. Time then becomes the third dimension, in addition to rows and columns. In R, such data sets are called arrays. As a simple example, consider students and test scores. Say each test consists of two parts, so we record two scores for a student for each test. Now suppose that we have two tests, and to keep the example small, assume we have only three students. Here's the data for the first test:

> firsttest

[,1] [,2]

[1,] 46 30

[2,] 21 25

[3,] 50 50

Student 1 had scores of 46 and 30 on the first test, student 2 scored 21 and 25, and so on. Here are the scores for the same students on the second test:

```
> secondtest
[,1] [,2]
[1,] 46 43
[2,] 41 35
[3,] 50 50
```

Now let's put both tests into one data structure, which we'll name tests. We'll arrange it to have two "layers"—one layer per test—with three rows and two columns within each layer. We'll store firsttest in the first layer and second test in the second. In layer 1, there will be three rows for the three students' scores on the first test, with two columns per row for the two portions of a test. We use R's array function to create the data structure:

```
> tests <- array(data=c(firsttest,secondtest),dim=c(3,2,2))
```

In the argument dim=c(3,2,2), we are specifying two layers (this is the second 2), each consisting of three rows and two columns. This then becomes an attribute of the data structure:

```
> attributes(tests)
$dim
[1] 3 2 2
```

Each element of tests now has three subscripts, rather than two as in the matrix case. The first subscript corresponds to the first element in the $dim vector, the second subscript corresponds to the second element in the vector, and so on. For instance, the score on the second portion of test 1 for student 3 is retrieved as follows:

```
> tests[3,2,1]
[1] 48
```

R's print function for arrays displays the data layer by layer:

```
> tests
,,1
[,1] [,2]
[1,] 46 30
[2,] 21 25
[3,] 50 48
,,2
[,1] [,2]
[1,] 46 43
```

[2,] 41 35

[3,] 50 49

Classes

R is an object-oriented language. Objects are instances of classes. Classes are a bit more abstract than the data types you've met so far. Here, we'll look briefly at the concept using R's S3 classes. (The name stems from their use in the old S language, version 3, which was the inspiration for R.) Most of R is based on these classes, and they are exceedingly simple. Their instances are simply R lists but with an extra attribute: the class name. For example, we noted earlier that the (nongraphical) output of the hist() histogram function is a list with various components, such as break and count components. There was also an attribute, which specified the class of the list, namely histogram.

> print(hn)

$breaks

[1] 400 500 600 700 800 900 1000 1100 1200 1300 1400

$counts

[1] 1 0 5 20 25 19 12 11 6 1

...

...

attr(,"class")

[1] "histogram"

At this point, you might be wondering, "If S3 class objects are just lists, why do we need them?" The answer is that the classes are used by generic functions. A generic function stands for a family of functions, all serving a similar purpose but each appropriate to a specific class. A commonly used generic function is summary(). An R user who wants to use a statistical function, like hist(), but is unsure of how to deal with its output (which can be voluminous), can simply call summary() on the output, which is not just a list but an instance of an S3 class. The summary() function, in turn, is actually a family of summary-making functions, each handling objects of a particular class. When you call summary() on some output, R searches for a summary function appropriate to the class at hand and uses it to give a friendlier representation of the list. Thus, calling summary() on the output of hist() produces a summary tailored to that function, and calling summary() on the output of the lm() regression function produces a summary appropriate for that function.

The plot() function is another generic function. You can use plot() on just about any R object. R will find an appropriate plotting function based on the object's class. Classes are used to organize objects. Together with generic functions, they allow flexible code to be developed for handling a variety of different but related tasks.