# An Introduction to Computational Probability and Statistics with R

Bee Leng Lee

 

# Contents

 

# Part I

# An Introduction to R

# Chapter 1

# Getting Started

R is a programming language and comprehensive statistical platform for data exploration and analysis. It is free and open source, which means anyone can download and use the latest version of the software free of charge, and the source code can be studied and modified without any restriction. Yet the functionality of R rivals commercial packages. Organizations and companies such as the FDA, Facebook and Google use R on a daily basis [10]. The minimalist interface of R, however, can be daunting to beginning users, especially those who are accustomed to the point-and-click interfaces offered by commercial packages. The purpose of this chapter is to orient beginning users. An up-to-date version of R for various computing platforms can be downloaded from the Comprehensive R Archive Network (CRAN) at `http://cran.r-project.org/`. We hasten to point out that while the R graphical user interface (RGui) appears a little different on each platform, there is no substantive difference otherwise.

## 1.1 The R Console

When R starts, a window similar to that shown in Figure 1.1 is presented. By default the window, called the R *console*, displays some information about R and a *command prompt* that is represented as a greater-than symbol (">"). The command prompt invites the user to type commands into R. When the user completes a command and presses return or enter, the R *interpreter* evaluates the command and displays the result in the console or a new window

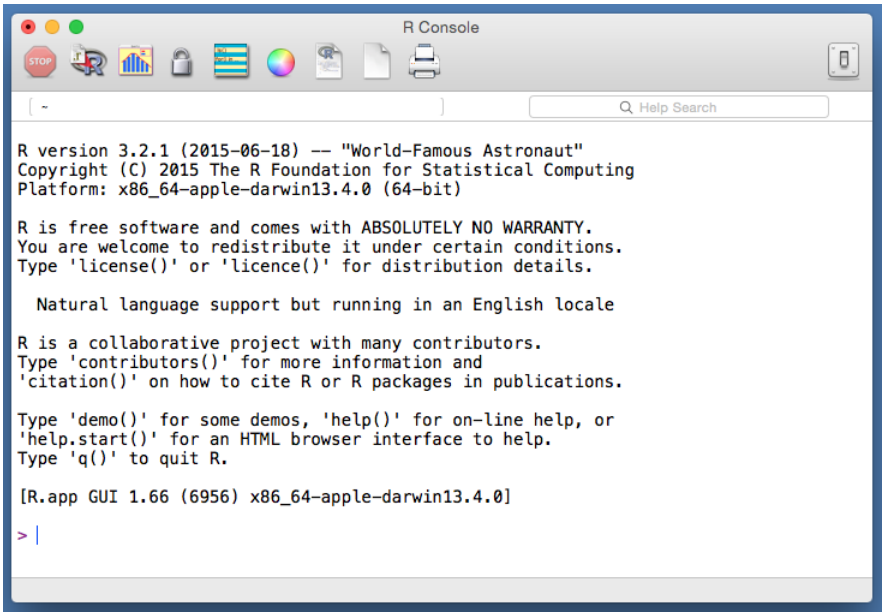


Figure 1.1: The R console on Mac OS X.

whenever appropriate.

To illustrate the command-line interface, we use R as a calculator to evaluate the sum of 1, 2 and 3:

```
> 1 + 2 + 3
[1] 6
```

The first line, excluding the command prompt, shows the command typed by the user. The `[1]` preceding the result in the second line indicates that `6` is the first element of the result—even when the result comprises one element. If a command is incomplete when return or enter is pressed, R will display the plus symbol (`+`) as a *continuation prompt* and await the user to complete the command:

```
> 1 + 2 +
+ 3
[1] 6
```

In the second line, the plus symbol was produced by R while the value `3` was subsequently typed by the user. This can be confusing. It is possible to customize the continuation prompt (and numerous other aspects of R). For example, to indicate continuation by indenting with two spaces:

 

```
> options("continue" = "   ")
> 1 + 2 +
  3
[1] 6
```

Sometimes a mistake is discovered while at the continuation prompt. To abort the command and return to the command prompt, press esc.

## 1.2 The R Editor

It is easy to mistype a command, especially when it is complex and spans multiple lines. Although R provides command-line editing, such as using ↑ and ↓ to scroll through previous commands, a better approach is to write a script or a list of commands in an editor. Besides making it easier to find and fix errors, a script can be saved for future reuse.

R provides an integrated editor which allows a script to be executed in part or whole from within the editor. In Figure 1.1, notice a menu bar just above the console window;[1] selecting File » New Document opens an editor window. Alternatively, one can use a keyboard shortcut, a method preferred by experienced users since it is quicker than mouse navigation. A keyboard shortcut comprises a *modifier key* and a character key. A modifier key is a special key that temporarily alters the action of other keys or mouse clicks. For example, pressing A normally produces a lowercase "a" but pressing shift + A produces an uppercase "A". To execute a keyboard shortcut, the modifier key is held down while the character key is pressed. For example, to open an editor window, press ⌘ + N on OS X or Ctrl + N on Windows.

Figure 1.2 shows an R editor window, with three lines of commands, atop the console window. To execute a single line of command, such as the second line "`4 + 5 + 6`", place the cursor anywhere on the line (which would automatically highlight the line) and press ⌘ + return on OS X or Ctrl + R on Windows. The command will appear in the console window along with any result. To execute multiple lines of commands, highlight the lines (such as by dragging the mouse across the lines) and press the aforementioned keyboard shortcut to execute commands.

There are two ways to execute all the commands in an R script. One is to highlight the entire content by pressing ⌘ + A on OS X or Ctrl + A on

[1] On OS X, the menu bar actually appears at the top of the screen.

 

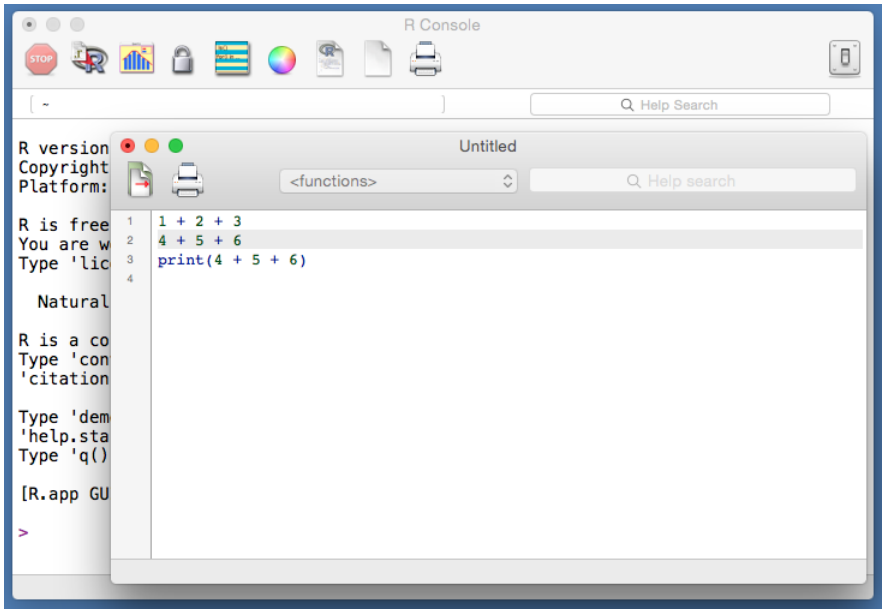


Figure 1.2: The R editor on Mac OS X.

Windows, followed by the keyboard shortcut for executing a command. The other assumes that the script has been saved.[2] From the menu bar, select File ⟩ Source File... and then locate and double-click on the name of the script in the dialog box. This process of sending commands to R for execution is referred to as "sourcing a script." Note that R responds to a sourced script a little differently; specifically, the results are not printed unless the user provides instructions to do so. The simplest way to print a result is to use the `print(`*`something`*`)` command, as shown on the third line of the script in Figure 1.2.

An R script can include *comments*, which are explanatory notes that are not meant to be evaluated by R. While a script may be written for personal use, there often comes a time when modification is necessary and, without comments, important details about how the code worked could have been forgotten by then. In R, any text that follows the symbol "`#`" is ignored, whether in a script or at the command prompt:

```
> 1 + 2 # This is ignored by R.
[1] 3
```

[2] By convention, the name of an R script ends with the suffix "`.R`".

 

```
> 1 + 2   This is not.
Error: unexpected symbol in "1 + 2   This"
```

## 1.3 The Working Directory

Each R session has associated with it a *working directory* (or folder) where files are retrieved from or saved to by default. Understanding this concept can help avoid many frustrations, such as searching every folder imaginable on the computer for a previously saved file or sourcing an outdated R script from an unintended folder. The command `getwd()` gives the working directory, which is usually `/Users/`*`username`* (not literally *`username`*) on OS X and `C:\Users\`*`username`*`\Documents` on Windows.

As an illustration, consider sourcing from the R console a script previously saved as "`myScript.R`". This is done using the `source("`*`filename`*`")` command:

```
> source("myScript.R")
```

If "`myScript.R`" is found in the working directory, R will execute its content; otherwise, an error message will be printed, the last two lines of which are:

```
In file(filename, "r", encoding = encoding) :
  cannot open file 'myScript.R': No such file or directory
```

To understand R's response, imagine looking for someone by the name of James Smith in a large organization with many offices in a building. If you stumble into one of the offices and ask for James Smith, chances are you will get a response similar to the above (some polite variant of "*cannot find James Smith: No such person*") or be greeted by a wrong James Smith. A better approach to look for James Smith would be to specify, in addition to his name, the division and department he works for. This information can be represented hierarchically as `/Division/Department/James Smith`.

Files on a computer are grouped into folders, which are organized in a hierarchy. The *absolute pathname* of a file describes its location in the hierarchy by tracing a path from the top-most folder, called the *root directory*, through all the intermediate folders, to the file. It begins with a "`/`" on OS X and "`C:\`" on Windows, where the letter "`C`" could be replaced by some other letter that identifies a drive or partition on the computer. This is illustrated in Figure 1.3 for OS X. The absolute pathname

 

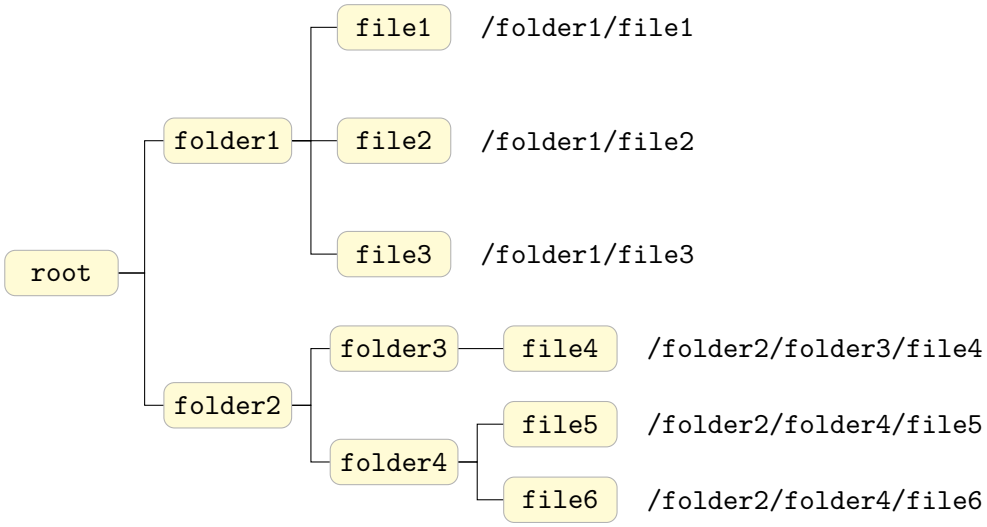


Figure 1.3: The hierarchical file structure and absolute pathnames.

"`/folder2/folder4/file6`" in the bottom right corner indicates that `file6` is a file located in a folder named `folder4`, which in turn is located in a folder named `folder2` in the root directory. The symbol "`/`" thus distinguishes folder levels, with the first "`/`" representing the root directory. If the name of a file is specified without a "`/`" on OS X or "`C:\`" on Windows, say, simply as "`myScript.R`", it is a *relative pathname* and the file is assumed to be located in the working directory. In other words, R assumes that the absolute pathname of the file is "`/Users/`*`username`*`/myScript.R`" on OS X or "`C:\Users\`*`username`*`\Documents\myScript.R`" on Windows.

It is useful to organize projects into directories and, when working on a particular project, set R's working directory to the associated directory. This can be done using the command `setwd("`*`absolutePathName`*`")`. For example, on the author's computer, the default working directory is:

```
> getwd()
[1] "/Users/blee"
```

A folder named `book` has been created in `/Users/blee` to store files related to this book. To designate this folder as the working directory:

```
> setwd("/Users/blee/book")
```

On Windows, the equivalent command would be:

 

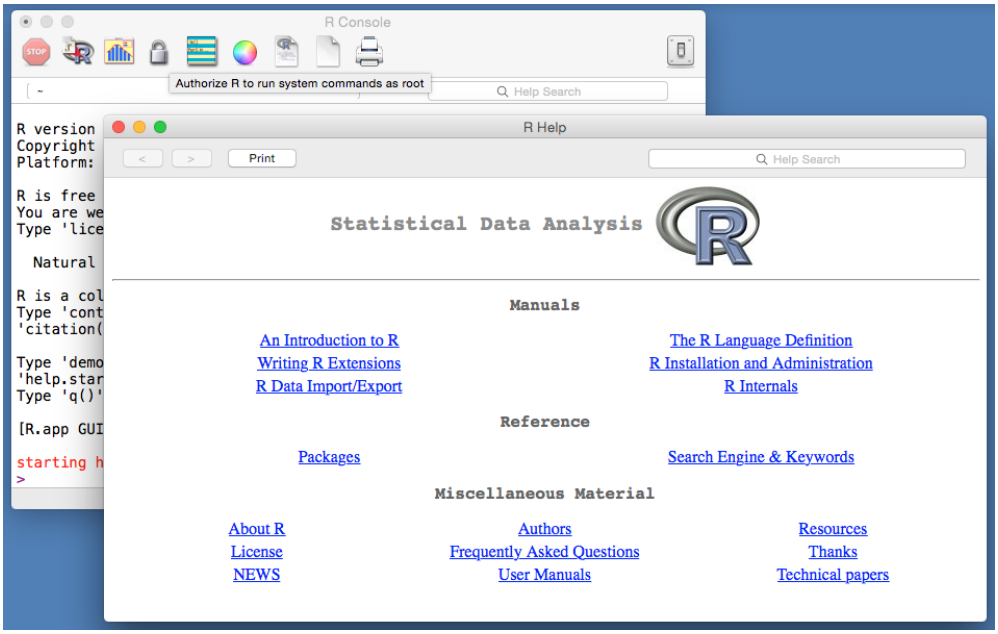


Figure 1.4: The R help window on Mac OS X.

```
> setwd("C:/Users/blee/Documents/book")
```

Note that pathnames are always specified with "/" in R, even on Windows; using "\" is a common mistake among beginning users who are accustomed to Windows.

## 1.4 Getting Help

R has a comprehensive built-in help system which may be accessed in several ways, one of which is to select Help » R Help from the menu bar. This presents a window similar to that shown in Figure 1.4, wherein users can search for help on various commands and find links to several manuals, in particular the highly recommended "An Introduction to R" by Venables, Smith and the R Core Team [13]. The `help(topic)` command can alternatively be used to invoke the help system. For example, to display the documentation on the `print` command, either type `help(print)` or `?print` at the command prompt, where the question mark ("`?`") is a shortcut for the command `help`.

There is also a plethora of internet resources for learning R and getting

 

help with problems. A few of these are listed below.

- CRAN, which contains the links shown in Figure 1.4 and more.
- R-Bloggers (http://www.r-bloggers.com/), where daily news and tutorials about R can be found.
- Rseek (http://rseek.org/), an R-specific search engine.
- Stack Overflow (http://stackoverflow.com/tags/r), a site with a very active R community asking and answering questions about R.

## 1.5 Exercises

1. By default, a short introductory message is displayed in the console window when R starts.

   (a) Write down the version number of R installed on your computer.
   (b) Write down the platform under which R is running on your computer.
   (c) List the four commands displayed in the console window just above the first command prompt.

2. The keyboard shortcuts for some commands can be found in the drop-down menus in the menu bar (usually to the right of the listed commands). For example, one way to save an R script is to select File ⟩ Save from the menu bar; the corresponding keyboard shortcut is ⌘ + S on OS X or Ctrl + S on Windows. List the keyboard shortcuts for the following.

   (a) File ⟩ Open Document on OS X or File ⟩ Open script on Windows, which opens a dialog box to load a previously saved script into the R editor.
   (b) Edit ⟩ Clear Console, which clears the screen in the console window.

3. Create an R script that contains the following two lines:

   ```
   1 - 2 + 3 * 4 / 6
   print(1 - 2 + 3 * 4 / 6)
   ```

   Create a folder named `compStat` in the default working directory of R and save the script as `ex1-3.R` in the folder.

   (a) In the R editor, execute the first line of `ex1-3.R` and note the result displayed in the R console.

 

(b) Source `ex1-3.R` from the R console and note the result displayed. Which line of command does the result correspond to? Explain.

 

 

# Chapter 2

# Fundamental Objects

Everything that exists in R is an object. This includes constants, data sets, functions and graphs. But what is an object in R? Put simply, it is a container for information referred to as *value* and it is self-describing—like a labeled food jar, except that it usually has a name. Among the descriptions attached to an object is its *class*, which defines what it contains as well as the way its content is organized. This basic concept of an object is one of the keys to understanding how to work with data in R, for a common mistake in R is to attempt to perform an operation on an object of a class that is incompatible with the operation. In this chapter, we present the rudiments of working with some of the fundamental objects in R, including assignments, special values, environments, functions, atomic vectors and lists.

## 2.1 Expressions and Assignment

Any command that is typed into the console is an *expression*, a symbol or a combination of symbols that evaluates to a value. For example, `1 + 2` is an expression which evaluates to the value 3. When R evaluates an expression, an object is created somewhere in the computer memory to store the value of the expression. The object (and hence the value it contains) is only accessible by name; an anonymous object gets deleted from the computer memory by a process called *garbage collection*. Thus, most objects are created by an operation called *assignment*, which establishes a *binding* or an association between an object and a name. This is usually done with the use of a symbol

composed of a less-than sign followed by a minus sign ("`<-`"), pronounced as "gets". For example:

```
> x <- 1 + 2 - 3
```

Here an object is created when R first evaluates the expression "`1 + 2 - 3`". The object holds the value of the expression and is bound to the name `x` by assignment. It is now accessible by name. For example, typing its name into the console displays its value (to the screen by default):

```
> x
[1] 0
```

It can also be used to create further objects:

```
> y <- x - 2
> y
[1] -2
```

There is a special object named `.Last.value` that contains the value of the last evaluated expression:

```
> 1 + 2 - 3
[1] 0
> .Last.value
[1] 0
> 3 + 2 - 1
[1] 4
> .Last.value
[1] 4
```

It might come in handy when a computationally intensive expression has been evaluated but, by an oversight, its value has not been assigned a name.

An object name can contain letters, digits, periods ("`.`") and underscores ("`_`"), with the restriction that it must begin with a letter or a period. If it begins with a period, it cannot be followed by a digit. R is case sensitive, for example, it considers `y` and `Y` to refer to different objects. It is useful to keep in mind that the name of an object is just a symbolic representation of the location of the object in the computer memory, analogous to a postal address that is used to indicate the location of a building. In other words, an object and its name are separate entities.[1] In the preceding listings, for

[1] In fact, the name of an object itself is an object, one which a user rarely needs to deal with and will not be discussed here.

 

example, any subsequent changes to the value of `x` does not affect the value of `y`:

```
> x <- 100
> y
[1] -2
```

It is the object that contains the value of `x - 2`, at the time of assignment, that is bound to `y`.

Assignments can alternatively be made using the symbol "`=`":

```
> x = 1 + 2 - 3
```

Some may prefer the symbol "`=`" to the symbol "`<-`" because it requires one less keystroke, but the two are not equivalent: "`<-`" always represents assignment whereas "`=`" can represent assignment (and more).

## 2.2 Special Values

Certain words in R are reserved and may not be used to name an object. Among these are *special values* that represent mathematical abstractions, rather than values in the usual sense, described as follows.

**`Inf`** Most students of mathematics recognize the symbol "$\infty$", called the lemniscate, which represents the concept of infinity. While infinity can be an elusive mathematical concept,[2] it means one of two things in R: either the value of an expression is too large (positively or negatively) to be represented by the computer, or a nonzero value is being divided by zero. These are indicated by the special value `Inf`. For example:

```
> 10^400
[1] Inf
> 1 - 10^400
[1] -Inf
> 1 / 0
[1] Inf
```

[2] A persistant myth in the history of mathematics is that Georg Cantor, a pre-eminent mathematician, descended into isolation and insanity as a consequence of his inability to resolve important questions about infinity.

 

Every computer has a well-defined range of values that it can represent; for most, the largest number allowed is approximately $1.79 \times 10^{308}$:

```
> 1.79 * 10^308
[1] 1.79e+308
> 1.8 * 10^308
[1] Inf
```

Note that R uses the notation `aeb`, where `a` and `b` are numbers, to stand for `a` $\times 10^{\texttt{b}}$; that is, the output `1.79e+308` means $1.79 \times 10^{308}$. This notation can also be used at the command prompt.

**NaN** When the result of a computation is undefined, R returns the special value `NaN`, which stands for "not a number." For example:

```
> 0 / 0
[1] NaN
> Inf / Inf
[1] NaN
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
```

Notice that a warning message is printed only when the expression "`sqrt(-1)`" is evaluated. The reason is, unlike $0/0$ or $\infty/\infty$, both of which are mathematically undefined, $\sqrt{-1}$ is defined in the complex number system, which is understood by R if the number $-1$ is explicitly expressed as a complex number (see page 36).

**NA** Missing values are a common occurrence in real-world data sets, which arise because some values are corrupted or unobserved to begin with. In R, missing values are represented with the special value `NA`, which stands for "not available." In general, any computations involving an `NA` results in an `NA`:

```
> NA - NA
[1] NA
> sqrt(NA)
[1] NA
> NA / 0
[1] NA
```

 

`NULL` refers to a special object in R that is used to indicate that an object does not exist. It is often confused with the special value `NA`. To illustrate the difference between `NULL` and `NA`, the command `length(`*`object`*`)` can be used to find out the number of elements each object contains:

```
> length(NA)
[1] 1
> length(NULL)
[1] 0
```

Note that `NULL` is not the only object in R that contains no elements. It is useful to think of `NA` as a placeholder for a value that should have been there, whereas `NULL` is a nonexistent value.

## 2.3 Classes

The class of an object describes its content, so that the object can be handled in an appropriate manner. Specifically, the same command, when applied to objects of different classes, can lead to different operations and produce different results. For example, consider the addition of two values using "`+`". If both values are ordinary numbers, "`+`" is just a basic arithmetic operation:

```
> x <- 25
> x + 7
[1] 32
```

If one of the values is a date and the other is a number, a different operation is performed:

```
> y <- Sys.Date()
> y
[1] "2015-05-25"
> y + 7
[1] "2015-06-01"
```

Here the command `Sys.Date()` returns an object that contains the current date, which is bound to the name `y`. When the value `7` is added to the current date, the result is the date seven days later. How does R know that a different operation is called for? Because R first queries an object for its class, then selects an operation that is appropriate for that class of objects, if available. The command `class(`*`object`*`)` returns the class of an object:

 

```
> class(x)
[1] "numeric"
> class(y)
[1] "Date"
```

Several classes of objects will be introduced in Section 2.6 and 2.7.

## 2.4 Environments

When R evaluates an expression that contains a name, say, "`x + 1`", how does R locate the object associated with the name "`x`" to retrieve the value? Effective use of R for data analysis requires an understanding of how R organizes objects created within it. The concept is a familiar one, as will be seen shortly.

During an assignment, the binding (or association) between an object and its name is stored in a specialized object called *environment*. More than one environment exists during an R session; in fact, in a typical R session, environments are constantly being created and destroyed, as discussed in the next section (page 30). In addition to a set of bindings, an environment contains a pointer to another environment, called the *enclosing* or *parent* environment. This is illustrated in Figure 2.1. It follows that there is a chain of environments, like a hierarchy of folders used to store files on a computer, where each folder contains a single file that catalogs a set of names and the locations of the associated objects, and a folder with a similar content. The chain of environments ends with the *empty* environment, the only environment without an enclosing environment and, as the name suggests, contains no bindings.

Whenever an expression is evaluated, one of the environments is "active", referred to as the *local* environment. To resolve a name that appears in the expression, R searches the local environment for a name that matches. If a matching name is found, the associated object is located and returned; otherwise, the enclosing environment is examined and the process repeated. An error is signalled if all the enclosing environments have been searched and a matching name cannot be found:

```
> x + 1
Error: object 'x' not found
```

 

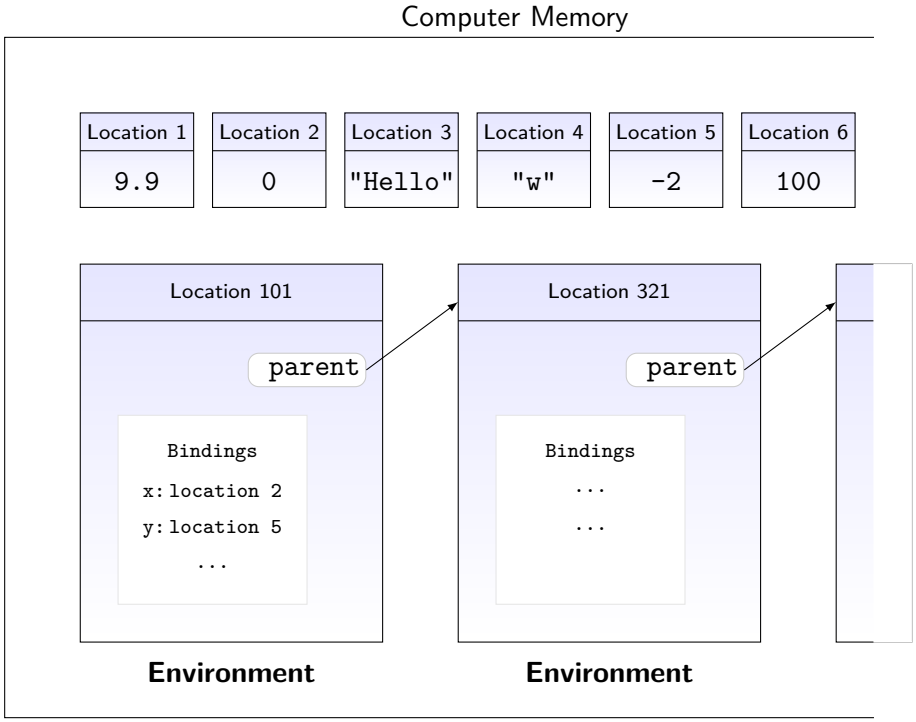


Figure 2.1: A schematic representation of an environment and its enclosing environment.

An analogy would be a "full-service" library, where a patron simply walks to the front desk to ask for a book by name. The librarian first searches the library catalog for a matching name to locate the shelf that holds the book. If found, the book is retrieved and handed to the patron; otherwise, the librarian proceeds to search the catalog of a branch library, until all branch libraries are exhausted.

Most users do not need to explicitly deal with environments. For now, it suffices to know that when R starts, a *global* environment or *workspace* is created, and every subsequent assignment that takes place at the command prompt adds an entry to it. In other words, the workspace is the local environment of all assignments that take place at the command prompt. The command `ls()` may be used to display the names contained in the workspace, which is usually empty at the beginning of an R session:

```
> ls()
```

 

```
character(0)
```

Here "`character(0)`" may be interpreted as "empty" for now. New assignments create new bindings that populate the workspace:

```
> x <- 1
> y <- 2
> z <- 3
> ls()
[1] "x" "y" "z"
```

To remove one or more bindings, use the command `rm(`*`objects`*`)`:

```
> rm(x)
> ls()
[1] "y" "z"
> rm(y, z)
> ls()
character(0)
```

When R evaluates the command "`rm(x)`", for example, it dissociates the name `x` and the object that was originally bound to it. The name `x` is removed from the workspace, like erasing an entry in a catalog, but nothing is done to the object itself.

## 2.5 Functions

A *function* in R is similar to a function in mathematics, in that it may take one or more *arguments* as input, performs some operations and produces an output. The input corresponds to a set of objects and the output is another object. Unlike a mathematical function, however, an R function may require no arguments.

### Calling Functions

R has many built-in functions, including the mathematical functions found on most scientific calculators, such as power, exponential and logarithmic. A function *call* is a command to execute the code of a function, like pressing one or more buttons on a calculator to perform a calculation. It usually

 

consists of the name of the function followed by a set of arguments enclosed in parentheses and separated by commas:

*name*`(`*argument*$_1$`,` *argument*$_2$`, ...,` *argument*$_n$`)`

For example, the command `exp(-1)` is a call to the exponential function `exp` with a single argument `-1` to compute $e^{-1}$:

```
> exp(-1)
[1] 0.3678794
```

The command `rm(y, z)` encountered earlier is a call to the function `rm` with two arguments, `y` and `z`. Arguments in function calls can in fact be any expression:

```
> exp(-0.2 * 5)
[1] 0.3678794
```

The parentheses following the name of a function are still required even when there are no arguments, as with `ls()`. If the parentheses are omitted, R displays the code that is executed when the function is called:

```
> ls
function (name, pos = -1L, envir = as.environment(pos),
all.names = FALSE, pattern)
{
    if (!missing(name)) {
        pos <- tryCatch(name, error = function(e) e)
...
```

The reason is, like everything else in R, a function is an object and typing the name of an object into the console displays its content. There are exceptions: some functions, called *primitive* functions, are implemented in another language (such as C or Fortran) and the compiled code cannot be meaningfully displayed. These functions can usually be identified by a one-line R code that calls, for example, the `.Primitive` or `.Internal` interface to the compiled code. For example, the function `exp` is primitive:

```
> exp
function (x)  .Primitive("exp")
```

For a list of built-in mathematical functions in R, type `?Math` into the console and click on the hyperlink "S3 Group Generic Functions" in the help window. The list can be found in the "Details" section.

 

## Specifying Arguments

When a function takes several arguments, it is useful to have a quick reminder of what they are. The function `args`, which takes the name of the function in question as the only argument, serves this purpose. For example:

```
> args(choose)
function (n, k)
NULL
```

The first line of the output indicates that the function `choose` takes two arguments, named `n` and `k`; it computes the binomial coefficient

$$\binom{\mathtt{n}}{\mathtt{k}} = \frac{\mathtt{n}!}{\mathtt{k}!(\mathtt{n}-\mathtt{k})!}\,.$$

The second line of the output, `NULL`, can be ignored and will be omitted from subsequent displays. In programming, the named arguments are called the *formal arguments*; they are used in the code of the function and serve as placeholders for the actual values specified by the user during a function call, referred to as the *actual arguments*. For example, in the following call to the function `choose`,

```
> choose(4, 2)
[1] 6
```

the actual arguments are 4 and 2, and they are matched with the formal arguments, `n` and `k`, respectively. Hereafter, unless there is ambiguity, actual arguments will be referred to simply as arguments.

In the preceding example, the actual arguments are bound to the formal arguments based on a default order, such as indicated in the output of a call to the function `args`. That is, the first and second actual arguments are bound to the first and second formal arguments, respectively. This is referred to as *positional matching*. Alternatively, an actual argument can be named, to associate it with a particular formal argument:

```
> choose(k = 2, n = 4)
[1] 6
```

Here the first actual argument, `2`, is named `k` and the second actual argument, `4`, is named `n`. For each named actual argument, R will search the set of formal arguments for one whose name matches exactly and, when found,

 

the two are bound. This is referred to as *exact matching*. As demonstrated in the preceding listing, exact matching allows the actual arguments to be given in any order. Although exact matching requires more typing, it is recommended because it improves the readability of a code and prevents the error of specifying the actual arguments out of order.

Like many programming languages, R allows functions to have optional arguments, which have predefined values and are not required to be specified in a function call. For example:

```
> args(log)
function (x, base = exp(1))
```

The `log` function takes two arguments, the number of which the logarithm is desired and the base of the logarthm, named `x` and `base`, respectively. The second argument is optional and has a default value e (Euler's number), as indicated by the expression `base = exp(1)`. That is, if the function is called with only one argument, R will compute its natural logarithm:

```
> log(4)
[1] 1.386294
```

If a different base is desired, the function is called with two arguments:

```
> log(4, base=2)
[1] 2
```

Notice that in the above function call, the first argument is not named whereas the second argument is. This is a common practice since it is usually obvious what the first argument is. In such a case where there is a mix of positional and exact matching in a function call, R first matches the arguments by their names, followed by positional matching for the remaining unmatched arguments.[3]

## Operators

An *operator* is a special symbol that performs a specific operation on values. For example, the symbol "`+`" is an operator that adds two values, and the symbol "`<-`" is an operator that binds the value of an expression to a name.

[3]There is actually a third method of matching, called *partial* matching, which is now deprecated.

 

Operators, in fact, are just functions in disguise. For example, the multiplication operator "`*`" is a function that takes two arguments, the values that appear before and after the symbol "`*`":

```
> 2 * 3
[1] 6
> `*`(2, 3)
[1] 6
```

In the preceding listing, the expression at the second command prompt is a call to the function named "`*`". As an illegal name (recall that only letters, numbers, periods and underscores are allowed in a name), the symbol "`*`" must be enclosed in backticks ("`` ` ``") to signal to R that the enclosed is a name. The backtick, also called backquote, is easily mistaken for the apostrophe.

Table 2.1 lists the arithmetic operators in R, along with examples. The operators are listed in order of *precedence*, which specifies the order in which the operators are applied when more than one appears in the same expression. An operator with a higher precedence is applied first, for example:

```
> 2 + 3 * 4
[1] 14
```

Here, because the multiplication operator has higher precedence than the addition operator, it is first applied to the values `3` and `4`, which evaluates to the value 12; after that, the addition operator is applied.

Parentheses may be used to force precedence:

```
> 2 + 3 * 4
[1] 14
> (2 + 3) * 4
[1] 20
```

The operator "`(`" is actually a function that returns the result of evaluating the expression it contains; it has higher precedence than the arithmetic operators. An operator of the same precedence is "`{`", which groups together a series of expressions that are separated by new lines or semicolons; when the grouped expressions are evaluated, only the value of the last expression is returned:

```
> {1 + 1; 1 - 1; {2 + 3} * 4}
[1] 20
```

 

Table 2.1: Arithmetic operators in R, listed in order of precedence from highest to lowest.

| Operator | Description | Example |
|---|---|---|
| `^` or `**` | exponentiation | `> 2^3`<br>`[1] 8`<br>`> 2**3`<br>`[1] 8` |
| `-` | negation | `> -1`<br>`[1] -1` |
| `*` | multiplication | `> 1 * 2`<br>`[1] 2` |
| `/` | division | `> 1 / 2`<br>`[1] 0.5` |
| `+` | addition | `> 1 + 2`<br>`[1] 3` |
| `-` | subtraction | `> 1 - 2`<br>`[1] -1` |
| `%%` | modulo | `> 4 %% 2`<br>`[1] 0` |
| `%/%` | integer division | `> 5 %/% 3`<br>`[1] 1` |

In Table 2.1, operators that are not separated by a horizontal line have the same precedence, in which case the order of evaluation is determined by *associativity*. Arithmetic operators are typically *left-associative*; that is, a series of operators of the same precedence is evaluated from left to right. For example:

```
> 1 - 2 - 3
[1] -4
```

Here R first evaluates `1 - 2`, the result of which is $-1$; after that, the second subtraction operator is applied. In other words, the order of evaluation is the same as `(1 - 2) - 3`. The only *right-associative* operator in Table 2.1 is the

 

Table 2.2: Comparison operators in R.

| Operator | Description | Example |
|---|---|---|
| `<` | less than | `> 1 < 2`<br>`[1] TRUE` |
| `>` | greater than | `> 1 > 2`<br>`[1] FALSE` |
| `<=` | less than or equal to | `> 1 + 2 <= 2 + 1`<br>`[1] TRUE` |
| `>=` | greater than or equal to | `> 1 + 2 >= 2 + 1`<br>`[1] TRUE` |
| `==` | equal to | `> 1 + 2 == 2 + 1`<br>`[1] TRUE` |
| `!=` | not equal to | `> 1 + 2 != 2 + 1`<br>`[1] FALSE` |

exponentiation operator, which is evaluated from right to left. For example, `2^3^4` first evaluates to $2^{81}$ rather than $8^4$. That is, the order of evaluation is equivalent to that of `2^(3^4)`.

Table 2.2 lists the comparison operators in R, which, as the name indicates, compare two values. The result of a comparison is either `TRUE` or `FALSE`. Comparison operators have lower precedence than arithmetic operators. For example, the expression "`1 + 2 < 2 + 1`" evaluates to the same value as the expression "`(1 + 2) < (2 + 1)`". They are non-associative, which means that a series of comparison operators in the same expression is not permitted unless parentheses are used:

```
> 1 < 2 == 2 < 3
Error: unexpected '==' in "1 < 2 =="
> (1 < 2) == (2 < 3)
[1] TRUE
```

To summarize the order of evaluations in R, function calls of the form *`name`*`(`*`arguments`*`)` as well as the operators "`(`" and "`{`" have the highest precedence. This is followed by the arithmetic, comparison, and assignment operators, in that order. While the use of parentheses is unnecessary if one

 

is acutely familiar with operator precedence, it enhances the clarity of code and minimizes the possibilities of error. For a complete list of operators in R and precedence, type `?Syntax` at the command prompt.

## Creating Functions

Once a user becomes familiar with R, it is natural to progress from using predefined functions to creating new functions. The reason is often a need for customized code to accomplish a specific task and to avoid repeating the code throughout an R session or a script. The function named "`function`" is used to create a function, using the following syntax:

`function(`*formal arguments*`)` *body*

Unlike the functions that have been introduced so far, this function takes two sets of arguments, namely, a pair of parentheses enclosing any formal arguments, and one or more expressions that will perform the desired computations, referred to as the *body* of the function. If the body of the function contains multiple expressions separated by semicolons or new lines, they must be grouped into a single expression with a pair of curly braces, "`{`" and "`}`". A user-defined function is stored as an object; although not necessary, it is usually assigned a name.

As a simple example, the following creates a function that takes as an argument a temperature measured on the Fahrenheit scale and converts it to the Celsius scale.

```
> celciusToFahrenheit <- function(x) x * 9 / 5 + 32
```

There is only one formal argument, `x`, and the body of the function is the expression "`x * 9 / 5 + 32`". The function can be called as usual:

```
> celciusToFahrenheit(100)
[1] 212
```

Notice that the result is printed, even without an explicit instruction to do so. This is because an evaluated expression, "`x * 9 / 5 + 32`" in the current example, is an object, and the value of an unassigned object is printed to the screen by default. Printing can be suppressed by calling the function `invisible`:

 

```
> celciusToFahrenheit <- function(x)
    invisible(x * 9 / 5 + 32)
```

Assignment also suppresses printing. If printing is desired, simply surround the assignment with parentheses:

```
> ( x <- celciusToFahrenheit(100) )
[1] 212
```

The spaces after the first "`(`" and before the last "`)`" are added for visual clarity only. This is done in the book whenever parentheses are used to print the result of an assignment.

A function can return only a single object. This is not a limitation since there are objects, to be introduced later, that can hold multiple objects. By default, the object that holds the value of the last evaluated expression is returned. For example:

```
> addTo123 <- function(x) {x + 1; x + 2; x + 3}
> addTo123(0)
[1] 3
```

Generally, the use of semicolons to separate expressions is not recommended.

When creating a new function, an important principle of R that should be kept in mind is that function calls should have no *side effects* on its arguments; this way, users "can be confident that successive computations will see consistent data" in the objects that are supplied as arguments. As an illustration of this principle, consider the following user-defined function, which attempts to modify its argument:

```
> setToZero <- function(x) {
    x <- 0
    x
  }
> a <- 1
> setToZero(a)
[1] 0
> a
[1] 1
```

Clearly, the attempt was unsuccessful. The reason is that, in a function call, any attempt to modify an object that is supplied as an argument causes a

 

copy of the object to be created. The copy, rather than the supplied object, is modified and returned. To demonstrate that this is case, the function `setToZero` is updated to print the memory address of the object `x` before and after assignment:

```
> setToZero <- function(x) {
    print(memAddr(x))
    x <- 0
    print(memAddr(x))
  }
> memAddr(a)
[1] "7fdbab3cf8d8"
> setToZero(a)
[1] "7fdbab3cf8d8"
[1] "7fdbab3cfa88"
```

The wrapper function `memAddr` can be found in the appendix. Notice that before the assignment "`x <- 0`", `x` refers to the same memory location as the actual argument `a`; afterward, it refers to a different memory location.

## Conditional Evaluation

By default, the expressions in the body of a function are evaluated sequentially, that is, one after the other. There are, however, many occasions where one or more expressions are to be evaluated only if a specified condition is met. R provides the `if` construct for conditional evaluation:

`if (`*`expression1`*`)` *`expression2`*

If *`expression1`* evaluates to `TRUE`, then *`expression2`* is evaluated; otherwise, `expression2` is not evaluated.

```
> grade <- NULL
> score <- 55
> if (score >= 60) grade <- "CR"
> grade
NULL
> score <- 80
> if (score >= 60) grade <- "CR"
> grade
[1] "CR"
```

 

An extension of the `if` construct is the `if-else` construct:

`if (`*`expression1`*`)` *`expression2`* `else` *`expression3`*

If *`expression1`* evaluates to `FALSE`, then *`expression3`* is evaluated.

```
> score <- 55
> if (score >= 60) grade <- "CR" else grade <- "NC"
> grade
[1] "NC"
```

A pitfall that awaits users with programming experience in C or C++ is to begin `else` on a new line:

```
> if (score >= 60) grade <- "CR"
>   else grade <- "NC"
Error: unexpected 'else' in "  else"
```

The error occurs because, unlike C or C++, R does not use a statement terminator to demarcate the end of a statement. The moment an expression is syntactically complete, as with the preceding `if` expression, R evaluates the expression. Once this is done, R proceeds to read the next line of code and encounters the keyword `else`, which cannot begin an expression.

## Environments

All user-defined functions in R are *closures*, which are objects that store both a function and an enclosing environment, the environment that was active when the function was created. The function `typeof` returns the internal storage type of an object, and the function `environment` returns the enclosing environment of a function:

```
> typeof(setToZero)
[1] "closure"
> environment(setToZero)
<environment: R_GlobalEnv>
```

Here "`R_GlobalEnv`" stands for the global environment. Primitive functions do not have an enclosing environment and are not closures, for example:

```
> typeof(exp)
[1] "builtin"
> environment(exp)
NULL
```

 

The enclosing environment of a function determines the set of objects that are visible in a call to the function. When a function is called, the formal arguments are first matched with the actual arguments; following that, the body of the function is evaluated sequentially. During the evaluation, if R encounters a name that is not bound to any object within the body of the function, it searches the enclosing environment of the function for a matching name, and the search continues up through the chain of environments if necessary. The value of the first match is used in the evaluation. As an illustration:

```
> f <- function(x) {
    y <- 2
    x + y + z
  }
> environment(f)
<environment: R_GlobalEnv>
> z <- 3
> ls()
[1] "f" "z"
> f(1)
[1] 6
```

When the call `f(1)` is evaluated, within the body of `f`, `x` is bound to the value 1, `y` is bound to the value 2, and `z` is unbound. To evaluate the expression "`x + y + z`", R searches the enclosing environment of `f`, the global environment, for `z`; a matching name is found and the associated value, 3, is used to evaluate the expression.

## 2.6 Atomic Vectors

Underlying every object in R is a *data structure* in the C language,[4] a method of storing, organizing and accessing data in the computer memory. Users rarely have to explicitly deal with such C structures unless there is a need to make use of functions written in other languages within R. This awareness, however, could be useful in understanding certain properties of objects in R. For example, the C data structure determines, among others, the data type or simply type, of an R object.

[4] R is primarily written in the C language.

 

The basic data structure in R is a *vector*, a sequence of elements of the same type, such as a sequence of integers or a sequence of characters. The function `c`, which stands for "concatenate", is used to construct a vector:

```
> ( x <- c(3, 1, 4, 1, 5, 9) )
[1] 3 1 4 1 5 9
```

Because a vector is a one-dimensional data structure, a vector of vectors is still a vector:

```
> c(x, c(2, 6, 5))
[1] 3 1 4 1 5 9 2 6 5
```

Associated with each element in a vector is an integer, called an *index*, which identifies the position of the element in the sequence. For those who are familiar with languages such as C or C++, the index of the first element is 1 rather than 0. Elements of a vector may be accessed by appending to the name of the vector an index or an index vector enclosed in a pair of brackets, "`[`" and "`]`". For example,

```
> x[3]
[1] 4
> x[c(1, 3, 5)]
[1] 3 4 5
```

Negative indices produce the opposite result, that is, all elements except those corresponding to the negative indices are extracted:

```
> x[c(-1, -3, -5)]
[1] 1 1 9
```

An index vector must be either all positive or all negative:

```
> x[c(1,-2)]
Error in x[c(1, -2)] : only 0's may be mixed with negative
subscripts
```

The number of elements a vector contains can be queried with the function `length`:

```
> length(x)
[1] 6
```

The function `length` can also be used to resize a vector, although it is rather unusual to do so. If a vector is shortened, extra elements are discarded (and lost); and if a vector is lengthened, it is padded with the special value `NA`:

 

```
> length(x) <- 3
> x
[1] 3 1 4
> length(x) <- 6
> x
[1]  3  1  4 NA NA NA
```

Names can be assigned to the elements of a vector when the vector is being created, in the same way a set of actual arguments are named in a function call. The purpose often is to provide a descriptive display:

```
> ( z <- c(length = 3, breadth = 1, height = 4) )
 length breadth  height
      3       1       4
```

Names may also be assigned after the vector has been created via the function `names`:

```
> z <- c(3, 1, 4)
> names(z) <- c("length", "breadth", "height")
> z
 length breadth  height
      3       1       4
```

To remove the names, assign the special value `NULL`:

```
> names(z) <- NULL
> z
[1] 3 1 4
```

## Classes

Technically, a vector containing a sequence of elements of the same type is referred to as an *atomic* vector, so called because it cannot be decomposed into simpler objects: even a single number is a one-element vector, possessing all the associated characteristics. To hold different types of elements, there are six classes of atomic vectors in R, described as follows.

**`numeric`** vectors hold numbers with decimal fractions, such as 3.14159. This is the default class for a sequence of numbers, even if none of them have a fractional part. For example, the vector `x` created earlier is a `numeric` vector, not a vector of integers:

 

```
> class(x)
[1] "numeric"
```

**integer** vectors contain integers. In most data analyses, it is not necessary to force numbers into integers, but if such a need do arise, the numbers must be explicitly expressed as integers using a suffix character "L":

```
> y <- c(3L, 1L, 4L, 1L, 5L, 9L)
> class(y)
[1] "integer"
```

or the function `as.integer`:

```
> z <- as.integer(c(3, 1, 4, 1, 5, 9))
> class(z)
[1] "integer"
```

R usually performs automatic conversions when necessary; for example, the vector `c(1, 3, 5)` is `numeric`, but it was used earlier as an index vector without causing any error.

**logical** vectors store logical values, `TRUE` and `FALSE`, which are reserved words in R; that is, these cannot be used to name objects:

```
> TRUE <- 1
Error in TRUE <- 1 : invalid (do_set) left-hand side to
assignment
```

R allows `TRUE` and `FALSE` to be abbreviated to `T` and `F`, respectively:

```
> c(T == TRUE, F == FALSE)
[1] TRUE TRUE
```

Here a logical vector with two elements is created via comparisons; the first element of the vector is the value of the expression "`T == TRUE`" and the second element is the value of the expression "`F == FALSE`", both of which evaluate to "`TRUE`", confirming the equality of the corresponding values. The abbreviations `T` and `F`, while commonly used, are best avoided because they are not reserved by R. Rather, they are pre-assigned the values `TRUE` and `FALSE`, respectively, at the start of an R session. They can be re-assigned:

```
> T <- FALSE
> c(T == TRUE, F == FALSE)
[1] FALSE  TRUE
```

 

One of the uses of logical vectors is *indexing*, that is, selecting one or more elements from a given vector. When a logical vector is supplied as an index vector, R returns those elements of the given vector whose index is `TRUE`:

```
> x <- c(3, 1, 4, 1, 5, 9)
> index <- c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)
> x[index]
[1] 3 4 5
```

**`character`** vectors have *strings*, or sequences of characters enclosed in quotes, as elements. For example:

```
> a <- "Wonder Pets! is a children's animated series."
> class(a)
[1] "character"
> length(a)
[1] 1
```

The above example illustrates that, unlike most programming languages, a string in R is not a vector of characters. That is, `a` is not a vector with 45 elements, where each element is a single character; rather, it is a one-element vector that contains a single string. The function `nchar` counts the number of characters in a string:

```
> nchar("Wonder Pets! is a children's animated series.")
[1] 45
```

As with any vectors, a character vector containing more than one element can be created using the function `c`:

```
> ( wonder.pets <- c("Linny", "Tuck", "Ming-Ming") )
[1] "Linny"     "Tuck"      "Ming-Ming"
```

A function that is frequently used for manipulating strings is `paste`. It can be used for element-wise concatenation of vectors:

```
> animals <- c("guinea pig", "turtle", "duckling")
> paste(wonder.pets, animals, sep = " is a ")
[1] "Linny is a guinea pig"   "Tuck is a turtle"
[3] "Ming-Ming is a duckling"
```

The optional argument `sep` is a string for separating the elements of the vectors to be concatenated; by default, the separator is a space, but is replaced by the string `" is a "` here.

 

Table 2.3: Classes of atomic vectors in R.

| Class | Elements | Examples |
|---|---|---|
| `numeric` | Numbers with decimal fractions. | `1`, `1.96` |
| `integer` | Integers. | `1L`, `as.integer(1)` |
| `logical` | Logical values. | `TRUE`, `FALSE` |
| `character` | A sequence of characters enclosed in quotes. | `"123"`, `"A"`, `"Hello!"` |
| `complex` | Complex numbers. | `1 + 2i` |
| `raw` | Bytes. | |

The remaining two classes of atomic vectors, `complex` and `raw`, rarely arise in statistical applications and will not be discussed in this book. They are briefly described in Table 2.3, which summarizes the six classes of atomic vectors.

For each class of atomic vectors, there is an identically named function that generates a vector from that class. The function takes an optional argument that specifies the desired length of the vector. For example:

```
> numeric(2)
[1] 0 0
> integer(2)
[1] 0 0
> logical(2)
[1] FALSE FALSE
> character(2)
[1] "" ""
```

If these functions are called without an argument, an empty vector of the corresponding class is created:

```
> character()
character(0)
```

Note that an empty vector is different from the `NULL` object. It is useful to think of both as empty food jars, the former is labeled whereas the latter is not. In other words, an empty vector possesses all the properties associated with an object of its class whereas the `NULL` object has none.

 

## Coercion

The individual elements in an atomic vector can only be described in terms of the implementation in the C language; that is, each element has a type defined in C. The function `typeof` provides this description. For example:

```
> x <- c(1, 2, 3)
> typeof(x)
[1] "double"
```

The individual elements in a vector of class "`numeric`" are of type "`double`", which are double-precision representations of real numbers, accurate to about 16 decimal digits. This can sometimes lead to unexpected results:

```
> (0.1 + 0.1 + 0.1) == 0.3
[1] FALSE
```

This is a limitation of computer arithmetic in general. For the other five classes of atomic vectors, the description returned by the function `typeof` is identical to that returned by the function `class`.

Because an atomic vector can only contain elements of the same type, when elements of different types are mixed, R automatically converts all of them to the same type. This process is called *coercion*, which is performed according to the following order of precedence, from the highest to the lowest: `"character"`, `"complex"`, `"numeric"`, `"integer"`, `"logical"`. For example, if an atomic vector contains an element of type `"character"`, all the other elements in the vector will be coerced into strings:

```
> x <- c(1, 2, 3)
> x[2] <- "two"
> typeof(x)
[1] "character"
```

When a logical vector and an integer vector are concatenated, the result is an integer vector:

```
> x <- c(TRUE, FALSE)
> y <- c(1L, 2L, 3L)
> z <- c(x, y)
> typeof(z)
[1] "integer"
> z
[1] 1 0 1 2 3
```

 

There are several functions available to perform explicit coercions. These have names that begin with "`as.`" followed by the respective class or type, for example, `as.numeric` or `as.character`.

## Vectorized Operations and Loops

Most operations in R are *vectorized*, whereby the operations are performed elementwise across one or more vectors. For example, the addition operator is vectorized:

```
> x <- c(1, 2, 3)
> y <- c(4, 5, 6)
> x + y
[1] 5 7 9
```

This may seem hardly surprising since the concept is familiar in mathematics: if $\mathbf{x}$ and $\mathbf{y}$ are defined as

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix},$$

the addition of $\mathbf{x}$ and $\mathbf{y}$ is

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} 1+4 \\ 2+5 \\ 3+6 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ 9 \end{bmatrix}.$$

However, the multiplication of two vectors with the operator "`*`" usually surprises users who are strangers to vectorized operations:

```
> x * y
[1]  4 10 18
```

In R, the operator "`*`" performs the Hadamard product, rather than the inner product, of two vectors:

$$\mathbf{x} \circ \mathbf{y} = \begin{bmatrix} 1 \times 4 \\ 2 \times 5 \\ 3 \times 6 \end{bmatrix} = \begin{bmatrix} 4 \\ 10 \\ 18 \end{bmatrix}.$$

 

This is consistent with the definition of a vectorized operation, that the operator is applied elementwise. All arithmetic and comparison operators in R are vectorized.

For functions, the word "vectorized" has two meanings. The first is the same as that for operators (which are, after all, functions):

```
> x <- c(1, 2, 3)
> sqrt(x)
[1] 1.000000 1.414214 1.732051
> z <- log(x)
> round(z, digits=2)
[1] 0.00 0.69 1.10
```

The second is that a vectorized function takes a vector as an argument, operates on the entire vector and returns a single value. The following example computes the sample mean, variance[5] and standard deviation, respectively, of the vector `z`:

```
> mean(z)
[1] 0.5972532
> var(z)
[1] 0.308634
> sd(z)
[1] 0.5555484
```

Why are there two seemingly unrelated meanings of "vectorized", apart from the fact that the operands of vectorized functions are vectors? Because vectorization refers to the use of *implicit loops* when operating on vectors. In programming, a *loop* is a sequence of statements that are repeated a certain number of times until one or more conditions are met. A loop is almost always necessary if an entire vector is to be operated on. For example, in the addition of two vectors `x` and `y`, the corresponding elements must be explicitly indexed and added: `x[i] + y[i]`; a loop automatically iterates over all indices `i`, sparing the user this repetitive task. In vectorized operations, the loops are implicit in the sense that they are not implemented by the user, rather, they occur in compiled code written in lower level languages.

Vectorized operations are usually faster than the use of equivalent explicit loops written in R. However, it is not always possible to vectorize, such

[5] In R, the sample variance is calculated using the divisor $n-1$, where $n$ is the number of sample values.

 

as when the calculations in each iteration depend on the results from the previous iteration. R provides three constructs for explicit looping, namely, `for`, `while`, and `repeat`. The syntax of the `for` loop is

`for (`*`name`* `in` *`vector`*`)` *`expression`*

First, an object named *`name`* is created. Then, for each element in *`vector`*, *`name`* is assigned the value of that element and *`expression`* (which can be grouped expressions) is evaluated. Such a loop is called a *counting loop* because the number of iterations is specified in advance, controlled by an object that counts from an initial value to an end value. The following example calculates the first six Fibonacci numbers, wherein the function `cat` is used to display the value of the counter `i`. The arguments to `cat` can be any number of objects separated by a comma, and the optional argument `sep` specifies a vector of strings to append after each element.

```
> fib.num <- numeric(6)
> fib.num[1] <- 1
> fib.num[2] <- 1
> for (i in c(3, 4, 5, 6)) {
    fib.num[i] <- fib.num[i-1] + fib.num[i-2]
    cat("i = ", i, "\n", sep = "")
  }
i = 3
i = 4
i = 5
i = 6
> fib.num
[1] 1 1 2 3 5 8
```

The syntax of the `while` loop is

`while (`*`condition`*`)` *`expression`*

It is called a *pre-test loop* because *`condition`* is evaluated first, and if it evaluates to `TRUE`, *`expression`* is evaluated. Then *`condition`* is evaluated again, and the process is repeated until *`condition`* evaluates to `FALSE`. Why might *`condition`* change from `TRUE` to `FALSE`? Presumably *`expression`* updates *`condition`* at each iteration; otherwise, an *infinite loop* (or endless loop) results. The following example calculates the first Fibonacci number that exceeds 10.

 

```
> fib.num <- numeric(2)
> fib.num[1] <- 1
> fib.num[2] <- 1
> i <- 2
> while (fib.num[i] <= 10) {
    i <- i + 1
    fib.num[i] <- fib.num[i-1] + fib.num[i-2]
  }
> cat("The first Fibonacci number that exceeds 10 is ",
      fib.num[i], "\n",
      "It is the ", i, "th number in the sequence.\n",
      sep = "")
The first Fibonacci number that exceeds 10 is 13
It is the 7th number in the sequence.
```

Notice that `fib.num` started out as a vector with 2 elements, yet an assignment beyond the end of the vector in the `while` loop does not trigger an error. Although it is convenient to have R extend a vector automatically, indexing beyond the length of a vector should be avoided. A reasonable upper bound on the length of a vector that "grows" inside a loop can usually be determined.

The syntax of the `repeat` loop is

`repeat` *`expression`*

It is an infinite loop that repeatedly evaluates *`expression`*. Therefore, *`expression`* should include a test of some condition that, when met, would terminate the loop. The `repeat` loop is called a *post-test loop* because the condition is usually tested at the end of a series of expressions. The reserved word `break` can be used to terminate any of the three loops. The following example reproduces the preceding calculation with a `repeat` loop.

```
> fib.num <- numeric(100)
> fib.num[1] <- 1
> fib.num[1] <- 1
> i <- 2
> repeat {
    i <- i + 1
    fib.num[i] <- fib.num[i-1] + fib.num[i-2]
```

 

```
    if (fib.num[i] > 10) break
  }
> length(fib.num) <- i
> fib.num
[1]  1  0  1  1  2  3  5  8 13
```

## Recycling

In mathematics, two or more vectors may be added or subtracted only if they have the same dimension. This is not the case in R. In vectorized operations involving vectors of different lengths, the shorter vectors are replicated, possibly partly, until the lengths of the resulting vectors match the length of the longest vector. This is referred to as *recycling*. For example:

```
> x <- c(1, 2)
> y <- c(3, 4, 5, 6)
> x + y
[1] 4 6 6 8
```

Here the vector `x` is replicated once, such that the operation is similar to:

```
> c(1, 2, 1, 2) + y
[1] 4 6 6 8
```

Recycling is silent, unless a vector is partly replicated:

```
> z <- c(7, 8, 9, 10, 11)
> x + y + z
[1] 11 14 15 18 15
Warning message:
In x + y + z :
  longer object length is not a multiple of shorter object
length
```

Here both `x` and `y` are partly replicated, such that "`x + y + z`" produces the same result as

```
> c(1, 2, 1, 2, 1) + c(3, 4, 5, 6, 3) + z
[1] 11 14 15 18 15
```

 

While recycling can be a source of error in computations when it is unintended, it can also be a useful feature when used purposefully. Recycling makes it easy to generate logical indices for selecting one or more elements of a vector. For example, to select elements of a vector that are divisible by two:

```
> z <- c(101, 102, 103, 104, 105, 106)
> z[z %% 2 == 0]
[1] 102 104 106
```

Here the index vector is generated from the expression "`z %% 2 == 0`". The operator "`%%`" computes the remainder after division of the left operand by the right operand. Because it is vectorized, the one-element vector `2` is replicated six times to match the length of the vector `z`:

```
> z %% 2
[1] 1 0 1 0 1 0
```

Next, the operator "`==`" compares each element of the above vector to the value `0`, which is also replicated six times:

```
> z %% 2 == 0
[1] FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

As another example, recycling also makes it easy to generate descriptive labels for factor levels in *analysis of variance* (ANOVA):

```
> paste("ageGroup", c(1, 2, 3), sep = "")
[1] "ageGroup1" "ageGroup2" "ageGroup3"
```

The function `paste` is vectorized, and so it operates on the vectors elementwise. The first vector has one element, `"ageGroup"`, while the second vector has three numeric elements, which R coerces into strings, `"1"`, `"2"` and `"3"`, prior to concatenation. The one-element character vector is replicated three times.

## 2.7 Lists

Lists are *generic* vectors. Like an atomic vector, a *list* is an ordered collection of elements, referred to as components. Unlike an atomic vector, the components of a list are not required to be of the same type. In fact, the

 

components can be arbitrary objects, even lists. For this reason, lists are also referred to as *recursive* vectors. As an extreme example, the following creates a list, using an identically named function, that contains a numeric vector, a character vector, a function and a list:

```
> ( unusual.list <- list(c(3, 1, 4, 1, 5, 9, 2, 6),
    c("three", "one", "four", "one", "five", "nine"), exp,
    list(c(3, 1), c("four", "one"))) )
```

```
[[1]]
[1] 3 1 4 1 5 9 2 6

[[2]]
[1] "three" "one"   "four"  "one"   "five"  "nine"

[[3]]
function (x)  .Primitive("exp")

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

[[4]][[2]]
[1] "four" "one"

> class(unusual.list)
[1] "list"
```

Thus, lists provide a means for functions to return multiple objects, possibly of different classes.

Displaying the content of a list by typing its name can produce a very long output, as can be seen from the preceding listing. The function `str`, which stands for "structure", displays the (abbreviated) content of an object in a compact form:

```
> str(unusual.list)
```

```
List of 4
 $ : num [1:8] 3 1 4 1 5 9 2 6
 $ : chr [1:6] "three" "one" "four" "one" ...
```

 

```
4  $ :function (x)
5  $ :List of 2
6   ..$ : num [1:2] 3 1
7   ..$ : chr [1:2] "four" "one"
```

The first line of the output informs the user that `unusual.list` is a list containing four components, described in lines 2–5, where "`num`" stands for "numeric" and "`chr`" stands for "character". A dollar sign ("`$`") precedes the description of each component. Notice, in lines 6–7, that the dollar signs corresponding to the components of the nested list are indented. This helps the user visualize nested structures, especially when there are multiple levels of nesting. The maximal level of nesting to display is controlled by the optional argument `max.level`:

```
> str(unusual.list, max.level = 1)
List of 4
 $ : num [1:8] 3 1 4 1 5 9 2 6
 $ : chr [1:6] "three" "one" "four" "one" ...
 $ :function (x)
 $ :List of 2
```

There are several optional arguments to the function `str` to customize display, type `?str` at the command prompt to see the R documentation.

## Indexing: Single Versus Double Brackets

A list is a vector in the sense that it is indexable, that is, its components can be accessed via integer or logical indices. However, indexing with single brackets, "`[`" and "`]`", as is done with atomic vectors, returns a list even if only one component is referenced:

```
> unusual.list[1]
[[1]]
[1] 3 1 4 1 5 9 2 6
> class(unusual.list[1])
[1] "list"
```

Operations that could be performed on vectors would produce an error on this list:

 

```
> a <- unusual.list[1]
> mean(a)
[1] NA
Warning message:
In mean.default(a) : argument is not numeric or logical:
returning NA
```

Sometimes a list can be converted into an atomic vector with the function `unlist`:

```
> unlist(a)
[1] 3 1 4 1 5 9
> mean(unlist(a))
[1] 3.833333
```

Many users do not realize that R provides cues to accessing the content of an object. For example, consider the display of `unusual.list` on page 44. The first line of the output cues the user that the first component of the list (displayed in line 2) is accessed using the symbol "`[[1]]`"; that is, the index "`1`" is enclosed in a pair of *double brackets*. More precisely, one appends the symbol "`[[i]]`" to the name of a list to access its `i`th component:

```
> unusual.list[[1]]
[1] 3 1 4 1 5 9 2 6
> class(unusual.list[[1]])
[1] "numeric"
> unusual.list[[2]]
[1] "three" "one"   "four"  "one"   "five"  "nine"
> class(unusual.list[[2]])
[1] "character"
```

In each case, indexing with double brackets returns the vector itself, rather than a list containing the vector. The elements of `unusual.list[[1]]` can be accessed as usual, by appending one or more indices enclosed in single brackets. For example:

```
> unusual.list[[1]][3]
[1] 4
> unusual.list[[1]][c(1,2,3)]
[1] 3 1 4
```

Lines 7–13 in the listing on page 44 may look confusing or intimidating at first, but the same principle applies:

 

- Line 7 gives the symbol for accessing the fourth component of the list, which is a nested list with two components.
- Lines 8 and 10 give the symbols for accessing the first two components of the nested list:

```
> unusual.list[[4]][[1]]
[1] 3 1
> unusual.list[[4]][[2]]
[1] "four" "one"
```

- The elements of the each component in the nested list, as shown in lines 9 and 11, may be accessed by further appending one or more indices enclosed in single brackets. For example:

```
> unusual.list[[4]][[1]][2]
[1] 1
```

There is another important difference between indexing with single and double brackets: only a single component may be indexed with double brackets. If more than one indices are given, they are applied recursively:

```
> unusual.list[[c(1, 3)]]
[1] 4
```

The first index, `1`, accesses the first component of `unusual.list`, the numeric vector; the second index, `3`, accesses the third element of the numeric vector. Additional indices will produce an error:

```
> unusual.list[[c(1, 3, 2)]]
Error in unusual.list[[c(1, 3, 2)]] :
  recursive indexing failed at level 2
```

## Accessing Components By Names

There is a more natural way to access the components of a list. When a list is being created, its components can be named, in the same way the elements of a vector are named. As a simple example, consider a list containing a numeric vector and a character vector:

```
> ( rhyme <- list(line1 = c(1, 2), line2 = "buckle my shoe") )
```

 

```
1 $line1
2 [1] 1 2

3 $line2
4 [1] "buckle my shoe"
```

In the first line of the output, R cues the user that the first component of the list (displayed in line 2) may be accessed by appending the symbol "`$line1`" to the name of the list:

```
> rhyme$line1
[1] 1 2
```

Similarly, the second component (displayed in line 4) may be accessed as `rhyme$line2`. In other words, list components can be accessed using two-part names: the name of the list and the name of the component separated by the operator "`$`".

The components of a list may also be named after the list has been created, through the function `names`:

```
> rhyme <- list(c(1, 2), "buckle my shoe")
> names(rhyme) <- c("line1", "line2")
> rhyme
$line1
[1] 1 2

$line2
[1] "buckle my shoe"
```

## Applying Functions Componentwise

There are two functions that may be used to apply a function to a list componentwise, namely, `lapply` and `sapply`, which stand for "list apply" and "simplified apply", respectively. The difference between these two functions is that, by default, the former returns the result as a list, whereas the latter simplifies the result to a vector when appropriate. Both take a list (or an atomic vector) as the first argument and a function to be applied as the second argument:

 

```
> lapply(rhyme, length)
$line1
[1] 2

$line2
[1] 1

> sapply(rhyme, length)
line1 line2
    2     1
```

Each component of the list `rhyme` is passed as an argument to the function `length`.

Sometimes the function to be applied takes more than one argument. Consider, for example, the function `summary`, which provides information about an object based on its class. The first argument of this function is an object for which a summary is desired. The remaining arguments control the formatting of output, one of which is `digits`, an integer specifying the number of significant digits to be used for printing numbers.

```
> lapply(rhyme, summary, digits = 2)
$line1
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     1.2     1.5     1.5     1.8     2.0

$line2
   Length     Class      Mode
        1 character character
```

Here `sapply` would produce a list too.

## 2.8 Saving and Loading Objects

Objects created during an R session can be saved to a file in a platform-independent binary format, so that they can later be retrieved in another R session, possibly on another computer with a different operating system. The function `save` allows an arbitrary number of objects to be saved into a single file, and the function `load` restores the objects:

 

```
> ls()
[1] "fib.num"      "rhyme"        "unusual.list"
> save(unusual.list, rhyme, file = "sampleLists.RData")
> rm(fib.num, unusual.list, rhyme)
> ls()
character(0)
> load("sampleLists.RData")
> ls()
[1] "rhyme"        "unusual.list"
```

A potential pitfall of restoring objects in this manner is that existing objects with the same names are replaced:

```
> rhyme$line3 <- c(3, 4)
> rhyme$line4 <- "shut the door"
> str(rhyme)
List of 4
 $ line1: num [1:2] 1 2
 $ line2: chr "buckle my shoe"
 $ line3: num [1:2] 3 4
 $ line4: chr "shut the door"
> load("sampleLists.RData")
> str(rhyme)
List of 2
 $ line1: num [1:2] 1 2
 $ line2: chr "buckle my shoe"
```

The function `load` accepts an optional argument, `envir`, which specifies an environment to store the retrieved bindings. A new environment can be created using the function `new.env()` and, like any object in R, it can be assigned a name. The object associated with a name in an environment is retrieved using two-part names: the name of the environment and the name of the object separated by the operator "`$`".

```
> rhyme$line3 <- c(3, 4)
> rhyme$line4 <- "shut the door"
> compStat <- new.env()
> load("sampleLists.RData", envir = compStat)
> str(rhyme)
List of 4
```

 

```
 $ line1: num [1:2] 1 2
 $ line2: chr "buckle my shoe"
 $ line3: num [1:2] 3 4
 $ line4: chr "shut the door"
> str(compStat$rhyme)
List of 2
 $ line1: num [1:2] 1 2
 $ line2: chr "buckle my shoe"
```

 

 

# Chapter 3

# Managing Data

R offers several classes of objects for organizing data, the choice of which depends on whether the data are *univariate*, consisting of observations on a single variable, or *multivariate*, which arises when more than one variable is observed. A further consideration is that a variable can be numerical or categorical, where the latter has observations that are grouped into different categories according to some nonnumeric characteristic. This chapter discusses basic data management, including the choice of data structure for a data set, importing the data set into the chosen structure, as well as accessing the data elements. The attributes of an object, which can be a very useful feature, are gradually introduced.

## 3.1 Importing Univariate Data

The atomic vectors introduced in Chapter 2 can be used to hold univariate data comprising values of the same type. The function `c` that was also introduced is useful for manually entering a small amount of data into a vector:

```
> x <- c(3, 1, 4, 1, 5, 9)
```

For moderate to large data sets, however, it would be impracticable to do so. Chances are such data sets have already been saved as files on the computer or can be found on the internet, in which case the function `scan` may be used to import the data sets.

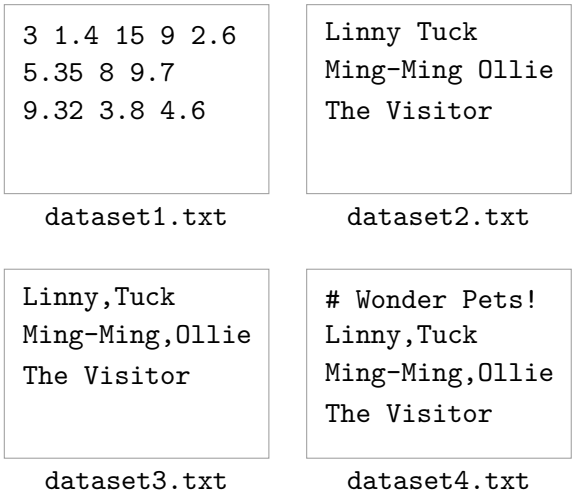


Figure 3.1: Examples of plain-text files.

Public data sets often come as *plain-text* files, usually saved with the extension `.txt`. These files contain only upper- and lower-case letters, digits and punctuation symbols, such as those depicted in Figure 3.1. In what follows, suppose that these files are located in the working directory, so that they can be referred to by their relative pathnames (see Section 1.3). To import data from a local plain-text file, the first argument to the function `scan` is a string containing the name of the file:

```
> scan("dataset1.txt")
Read 11 items
 [1]  3.00  1.40 15.00  9.00  2.60  5.35  8.00  9.70  9.32
[10]  3.80  4.60
```

The numeric vector returned by `scan` must be assigned a name or it will not be accessible (except through `.Last.value`, see Section 2.1). Thus, a typical use of `scan` occurs in an assignment:

```
> dataset1 <- scan("dataset1.txt")
Read 11 items
```

In order to accomodate different types of data and plain-text formats, the function `scan` has several optional arguments, some of which are described as follows.

**`what`** Numeric data are expected by default. Consequently, the following attempt to import the data from `dataset2.txt` produces an error:

 

```
> scan("dataset2.txt")
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines,
na.strings,  :
  scan() expected 'a real', got 'Linny'
```

When the values are not numeric, the argument `what` must be included in a call to the function `scan`. It can be specified with an example of a value of the desired type, or a call to one of the functions `integer`, `logical` or `character`:

```
> scan("dataset2.txt", what="a")
Read 6 items
[1] "Linny"     "Tuck"      "Ming-Ming" "Ollie"
[5] "The"       "Visitor"
> scan("dataset2.txt", what=character(0))
Read 6 items
[1] "Linny"     "Tuck"      "Ming-Ming" "Ollie"
[5] "The"       "Visitor"
```

**sep** Notice that in the preceding listing, the single string "`The Visitor`" has been incorrectly imported as two separate strings, "`The`" and "`Visitor`". The reason is, by default, the function `scan` assumes that the values are separated by spaces. One way to overcome this problem is to enclose each string that contains one or more spaces in a pair of quotes. Another is to use a different delimiter. For example, `dataset3.txt` uses commas ("`,`") to separate values. The argument `sep` allows the user to specify a character, enclosed in quotes, as a delimiter:

```
> scan("dataset3.txt", what = character(0), sep = ",")
Read 5 items
[1] "Linny"       "Tuck"        "Ming-Ming"   "Ollie"
[5] "The Visitor"
```

Regardless of the delimiter, a new line always separate values.

**skip** Sometimes there is one or more lines of description at the beginning of a data file, such as the first line of `dataset4.txt`. Although the symbol "`#`" is recognized by R as a comment character when it appears in an R script or at the command prompt, the function `scan` turns off the interpretation of comments by default:

 

```
> scan("dataset4.txt", what = character(0), sep = ",")
Read 6 items
[1] "# Wonder Pets!" "Linny"          "Tuck"
[4] "Ming-Ming"      "Ollie"          "The Visitor"
```

The argument `skip` instructs `scan` to skip a specified number of lines before reading the data:

```
> scan("dataset4.txt", what = character(0), sep = ",",
  skip = 1)
Read 5 items
[1] "Linny"       "Tuck"        "Ming-Ming"   "Ollie"
[5] "The Visitor"
```

**`comment.char`** Alternatively, the user can specify a character, enclosed in quotes, for the optional argument `comment.char`, so that whatever follows the specified character on the same line will be ignored:

```
> scan("dataset4.txt", what = character(0), sep = ",",
  comment.char = "#")
Read 5 items
[1] "Linny"       "Tuck"        "Ming-Ming"   "Ollie"
[5] "The Visitor"
```

To import data from a website, simply substitute the site's URL for the file name:

```
> scan("http://www.math.sjsu.edu/~lee/dataset1.txt")
Read 11 items
 [1]  3.00  1.40 15.00  9.00  2.60  5.35  8.00  9.70  9.32
[10]  3.80  4.60
```

## 3.2 Generating Patterned Data

It is useful to be able to generate a patterned sequence of values, perhaps to code an existing data set. For example, anonymized data do not contain personal identifier such as name; instead, each individual is assigned a code, say, $1, 2, 3$, and so on. If multiple observations are made on each individual, say, three, the corresponding code sequence for the observations could be $1, 1, 1, 2, 2, 2, 3, 3, 3, \ldots$ or $1, 2, 3, \ldots, 1, 2, 3, \ldots, 1, 2, 3, \ldots$.

 

There are several functions in R for creating patterned sequences, the simplest of which is the colon operator (“`:`”). The expression “`m:n`”, where `m` and `n` are integers, produces a sequence of integers from `m` to `n`, with a difference of one between successive values:

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 2.5:10
[1] 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
> 5:(-5)
 [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

Note that the parentheses surrounding `-5` in the last expression are added for clarity only, since the unary minus operator—for negation—has a higher precedence (see Section 2.5) than the colon operator. On the other hand, the colon operator precedes all binary arithmetic operators. For example, the expression “`1:5+1`” produces the same sequence as “`(1:5) + 1`”, rather than “`1:(5 + 1)`”:

```
> 1:5+1
[1] 2 3 4 5 6
```

The functions `seq` and `seq.int` provide more flexibility than the colon operator. Both have the same set of arguments, the former is written in R while the latter is a primitive. Typical usages of `seq.int` include:

```
> seq.int(from = 2, to = -1, by = -0.5)
[1]  2.0  1.5  1.0  0.5  0.0 -0.5 -1.0
> seq.int(from = 2, to = -1, length.out = 7)
[1]  2.0  1.5  1.0  0.5  0.0 -0.5 -1.0
```

In each case, a decreasing sequence of equally spaced values from 2 to $-1$ is generated. The first expression uses the argument `by`, which specifies the difference between the successive values in the sequence. The second expression uses the argument `length.out`, often abbreviated to `length`, which specifies the desired length of the sequence. If the third argument is omitted, the function `seq.int` behaves like the colon operator:

```
> seq.int(2, -1)
[1]  2  1  0 -1
```

Often in programming, the length of an object is supplied as the second operand of the colon operator, or the `to` argument of `seq.int`, to generate a vector of indices for the object:

 

```
> x <- c(3, 1, 4, 1, 5, 9, 2, 6, 5)
> 1:length(x)
[1] 1 2 3 4 5 6 7 8 9
> seq.int(from = 1, to = length(x))
[1] 1 2 3 4 5 6 7 8 9
```

However, users tend to be caught unawares by the result when the object has length zero:

```
> y <- NULL
> 1:length(y)
[1] 1 0
> seq.int(from = 1, to = length(y))
[1] 1 0
```

In this case, an empty vector should have been generated. For this reason, the function `seq_along` is preferred for generating indices:

```
> seq_along(x)
[1] 1 2 3 4 5 6 7 8 9
> seq_along(y)
integer(0)
```

The function `rep` is useful for repeating a vector. It has three optional arguments, usually only one is specified:

```
> a <- 1:3
> rep(a, times = 2)
[1] 1 2 3 1 2 3
> rep(a, each = 2)
[1] 1 1 2 2 3 3
> rep(a, length.out = 5)
[1] 1 2 3 1 2
```

If `each` is specified along with either of the other two optional arguments, its replication is performed first:

```
> rep(a, times = 2, each = 3)
 [1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3
```

 

## 3.3 Factors For Categorical Data

*Categorical data* refers to data whose values consist of a set of categories. For example, in an opinion poll, data collected on the sex of the respondents have two possible categories, "female" and "male"; the respondents may be asked to select one of the following to represent their opinion on a particular issue: "strongly disagree", "disagree", "neutral", "agree", or "strongly agree". These categories are intrinsically non-numeric, but for ease in data processing, they are often assigned numerical codes. For example, "female" and "male" could be coded numerically as 0 and 1; the responses may be coded as $-2$ for "strongly disagree", $-1$ for "disagree", 0 for "neutral", and so on. These numbers, however, are abitrary in the sense that a different set of numbers could have been used instead.

### Creating Factors

In R, categorical data can be stored as *factors*, a class of objects that contain only a predefined set of distinct values, referred to as the *levels* of a factor. To illustrate, suppose the vector `poll` contains six responses to the opinion poll:

```
> poll <- c("Strongly agree", "Strongly agree", "Neutral",
  "Neutral", "Agree", "Disagree")
```

The function `factor` is used to create a factor from a vector:

```
> ( poll.fac <- factor(poll) )
[1] Strongly agree Strongly agree Neutral
[4] Neutral        Agree          Disagree
Levels: Agree Disagree Neutral Strongly agree
```

There are several optional arguments to `factor`, some of which are described as follows.

**`levels`** By default, the levels of a factor are determined by the distinct values in the data. Notice that, in the foregoing example, the data did not include the response "`Strongly disagree`"; as a result, it is not among the levels of `poll.fac`. If a mistake had been made in data entry, say, the fourth response should be "`Strongly disagree`" rather than "`Neutral`", the following attempt to correct the mistake would be unsuccessful:

 

```
> poll.fac[4] <- "Strongly disagree"
Warning message:
In `[<-.factor`(`*tmp*`, 4, value = "Strongly disagree") :
  invalid factor level, NA generated
> poll.fac
[1] Strongly agree Strongly agree Neutral
[4] <NA>           Agree          Disagree
Levels: Agree Disagree Neutral Strongly agree
```

Because levels other than those originally created are not allowed in a factor, R replaces the value of `poll.fac[4]` by the special value `NA`. The optional argument `levels` allows a set of possible values to be specified:

```
> choices <- c("Strongly disagree", "Disagree", "Neutral",
  "Agree", "Strongly agree")
> ( poll.fac <- factor(poll, levels = choices) )
[1] Strongly agree Strongly agree Neutral        Neutral
[5] Agree          Disagree
5 Levels: Strongly disagree Disagree ... Strongly agree
```

**`ordered`** Categorical data are either *nominal* or *ordinal*. If the categories have a natural ordering, the data are ordinal; otherwise, they are nominal. For example, data on the sex of poll respondents are nominal since there is no inherent ordering between "female" and "male". On the other hand, the five response categories ranging from "strongly disagree" to "strongly agree" can be meaningfully ordered. Disregarding such order information often results in a loss of statistical power. The optional argument `ordered` should be specified as `TRUE` for ordinal data, so that they will be handled appropriately by the statistical functions in R:

```
> ( poll.ord <- factor(poll, levels = choices,
    ordered = TRUE) )
[1] Strongly agree Strongly agree Neutral
[4] Neutral        Agree          Disagree
5 Levels: Strongly disagree < Disagree < ... < Strongly agree
```

The ordering of the categories, as shown in the last line of preceding listing, is determined by the order in which they were specified in the optional argument `levels`. If `levels` is not specified, R arranges the categories alphabetically.

 

**labels** Suppose the categories are numerically coded:

```
> poll.code <- c(2, 2, 0, 0, 1, -1)
> ( poll.code.fac <- factor(poll.code,
    levels = c(-2, -1, 0, 1, 2)) )
[1] 2  2  0  0  1  -1
Levels: -2 -1 0 1 2
```

A set of labels for the codes can be specified through the optional argument `labels`, which would provide a more descriptive display of the data:

```
> ( poll.code.fac <- factor(poll.code,
    levels = c(-2, -1, 0, 1, 2), labels = choices) )
[1] Strongly Agree Strongly Agree Neutral
[4] Neutral        Agree          Disagree
5 Levels: Strongly disagree Disagree ... Strongly Agree
```

## Understanding Factors and Attributes

A factor stores data as a vector of integer codes, which range from one to the number of distinct values in the data. In addition, a character vector is used to store the distinct values in their original form. Each integer code is associated with a unique string in the character vector, which R uses to display the data. The character vector associated with a factor is an *attribute* of the factor. In general, an attribute of an object is an auxiliary information attached to the object. There are two *intrinsic* attributes that are shared by all objects in R, namely, *mode* and *length*; atomic vectors possess only these attributes. The mode attribute is rarely used, it exists for compatibility with S, the predecessor of R.

A factor is an integer vector with two additional attributes, named `levels` and `class`. The function `attributes` returns all the non-intrinsic attributes of an object as a list:

```
> ( poll.fac <- factor(poll, levels = choices) )
> attributes(poll.fac)
$levels
[1] "Strongly disagree" "Disagree"          "Neutral"
[4] "Agree"             "Strongly agree"
```

 

```
$class
[1] "factor"
```

An attribute that is predefined by R can be accessed via an identically named function:

```
> length(poll.fac)
[1] 6
> levels(poll.fac)
[1] "Strongly disagree" "Disagree"          "Neutral"
[4] "Agree"             "Strongly agree"
> class(poll.fac)
[1] "factor"
```

Like the label on a food jar, an attribute can be attached to or removed from an object without affecting its content. Stripping a factor of all its attributes reveals an integer vector as its underlying structure:

```
> attributes(poll.fac) <- NULL
> poll.fac
[1] 5 5 3 3 4 2
> class(poll.fac)
[1] "integer"
```

Note that while the `class` attribute has been removed from `poll.fac`, every atomic vector has an implicit class, as described in Section 2.6. In the last line of the preceding listing, the function `class` reports the implicit class of the underlying vector.

The function `attr` is used to attach an attribute to an object. The first argument is the name of the object and the second argument is a string specifying the name of the attribute to be attached (or accessed):

```
> attr(poll.fac, "levels") <- choices
> poll.fac
[1] 5 5 3 3 4 2
attr(,"levels")
[1] "Strongly disagree" "Disagree"          "Neutral"
[4] "Agree"             "Strongly agree"
```

The `levels` attribute is now re-attached to `poll.fac`; however, it is simply being displayed along with the content of `poll.fac`, a set of integer codes used to represent the data. The reason is, `poll.fac` remains an integer vector. In fact, one can perform (meaningless) arithmetic on `poll.fac`:

 

```
> sum(poll.fac)
[1] 22
```

The following demonstrates that, when an object is of the class `factor`, R uses the `levels` attributes to display its content and inhibits it from being used in arithmetic operations:

```
> attr(poll.fac, "class")  <- "factor"
> poll.fac
[1] Strongly agree Strongly agree Neutral        Neutral
[5] Agree          Disagree
5 Levels: Strongly disagree Disagree Neutral ... Strongly agree
> sum(poll.fac)
Error in Summary.factor(c(5L, 5L, 3L, 3L, 4L, 2L),
na.rm = FALSE) :
  'sum' not meaningful for factors
```

The functions `attributes` and `attr` are rarely used in data analysis. Understanding the associated concepts, as described above, makes factors less confusing, or perhaps less intimidating, to work with: they are simply integer vectors disguised as character vectors.

## 3.4 Structures For Tabular Data

Data comprising multiple observations of more than one variable are often arranged in rows and columns, referred to as *tabular data*. There are two possible arrangements of tabular data: *stacked* and *unstacked*. Data are stacked if each column contains the values of a single variable across all observations, so that each variable forms a column and each observation forms a row. Many model-fitting functions in R expect data to be arranged as such. An example is given in Table 3.1a, which shows a portion of the data from a study of the effect of caffeine on the performance of a simple task, finger tapping [5]. The data contain thirty values that correspond to two variables and thirty observations, where the variables are the amount of caffeine received (0, 100 or 200 ml) and the number of finger taps per minute. The complete data is shown in Table 3.1b in an unstacked arrangement, which is the common form of presentation in articles and books due to its compactness. With unstacked data, the values are grouped into columns (or rows) according to the values of one variable, in this case, the amount

 

Table 3.1: Number of finger taps per minute achieved by thirty male college students receiving designated doses of caffeine.

(a) Stacked data.

| Caffeine (ml) | Taps per minute |
|---|---|
| 0 | 242 |
| 0 | 245 |
| ⋮ | ⋮ |
| 100 | 248 |
| 100 | 246 |
| ⋮ | ⋮ |
| 200 | 246 |
| 200 | 248 |
| ⋮ | ⋮ |
| 200 | 250 |

(b) Unstacked data.

| Caffeine (ml) | | |
|---|---|---|
| 0 | 100 | 200 |
| 242 | 248 | 246 |
| 245 | 246 | 248 |
| 244 | 245 | 250 |
| 248 | 247 | 252 |
| 247 | 248 | 248 |
| 248 | 250 | 250 |
| 242 | 247 | 246 |
| 244 | 246 | 248 |
| 246 | 243 | 245 |
| 242 | 244 | 250 |

of caffeine received. More generally, it is useful to think of stacked data as a format in which the column headers are variable names, whereas for unstacked data, the column headers are values [14]. These two formats can be converted from one to the other. This will be discussed after two classes of objects for storing tabular data are introduced.

## Matrices

A *matrix* in R is similar to a matrix in linear algebra: it organizes a collection of values into rows and columns. Tabular data may be stored as a matrix provided the values are of the same type, such as all numeric or all characters. The reason for this limitation is that matrices are atomic vectors in yet another disguise, with an added attribute named `dim`, which stands for "dimension". The dimension attribute determines the number of rows and columns the data are organized into. This understanding can greatly facilitate working with matrices. To illustrate, the following first creates an integer vector and shows that it does not possess a dimension attribute, which is retrieved or set using the function `dim`:

```
1 > ( a <- 1:6 )
2 [1] 1 2 3 4 5 6
```

 

```
> class(a)
[1] "integer"
> dim(a)
NULL
```

Next, the vector `a` is converted into a matrix simply by altering its dimension attribute (currently `NULL`), specified as a two-element vector containing the number of rows followed by the number of columns:

```
> dim(a) <- c(2,3)
> attributes(a)
$dim
[1] 2 3
> class(a)
[1] "matrix"
> a
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Notice that `dim` is simultaneously the name of an attribute and the name of the function to access the attribute. This is true for many objects in R. The preceding example also illustrates the internal storage of a matrix: comparing line 2 and lines 11–13 of the preceding two listings, it can be seen that the elements of a matrix are stored in *column-major order*, that is, consecutive elements of the columns are stored contiguously in a vector (imagine stacking the columns on top of one another to form a vector).

Because of the limitation that the data values must be of the same type, matrices are seldom the structure of choice to store primary tabular data; rather, they are commonly used to store covariances or correlations, as well as in model fitting. That said, simpler is better, for operations with matrices can be much faster than the same operations on more sophisticated data structures.

### Creating Matrices

A matrix is usually created using the function `matrix`:

```
> args(matrix)
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE,
dimnames = NULL)
```

 

The first argument, `data`, is a vector containing the data. The arguments `nrow` and `ncol` specify the number of rows and columns, respectively. For example:

```
> ( m1 <- matrix(1:6, nrow = 2, ncol = 3) )
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

The number of rows and columns of a matrix can be queried using identically named functions:

```
> nrow(m1)
[1] 2
> ncol(m1)
[1] 3
```

It suffices to specify either `nrow` or `ncol` when creating a matrix, for one can be inferred from the other based on the length of the supplied vector. Also, if the vector has fewer elements than specified by `nrow` and `ncol`, R recycles the vector. This provides a convenient way to preallocate memory to a matrix:

```
> matrix(0, nrow = 2, ncol = 3)
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
```

Preallocating memory to objects, while not necessary in R, can improve the speed of an R code. This will be discussed shortly.

By default, a matrix is filled column by column. It is possible to fill a matrix row-wise by specifying `byrow = TRUE`:

```
> ( m2 <- matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE) )
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Even then the elements are still stored in column-major order:

```
> attributes(m2) <- NULL
> m2
[1] 1 4 2 5 3 6
```

 

A matrix can also be created by binding two or more vectors column-wise or row-wise, using the functions `cbind` and `rbind`, respectively:

```
> ( m3 <- cbind(c(1, 2), c(3, 4)) )
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> ( m4 <- rbind(c("one", "two"), c("three", "four")) )
     [,1]    [,2]
[1,] "one"   "two"
[2,] "three" "four"
```

Additionally, these functions may be used to append a row or column, or even a matrix, to an existing matrix, provided that the dimensions are compatible. For example:

```
> cbind(m3, m4)
     [,1] [,2] [,3]    [,4]
[1,] "1"  "3"  "one"   "two"
[2,] "2"  "4"  "three" "four"
```

By the rules of coercion (see page 37), the elements of the resulting matrix are converted to the type "`character`".

It is important to keep in mind that repeated use of `cbind` or `rbind` to build a large matrix could result in significantly longer execution time than preallocating memory to the matrix. The reason is that if the memory allocated to the initial matrix is insufficient to accomodate an additional row or column, the matrix would have to be copied to a larger memory block. Such overhead could be incurred several times over the course of building a large matrix using `cbind` or `rbind`.

### Example: Importing Unstacked Data

The following example demonstrates how to import the unstacked data in Table 3.1b and convert it to stacked data. Suppose the data file, named `caffeine.txt`, is located in the working directory. The first three lines of the file are shown below:

 

caffeine.txt

```
  0 100 200
242 248 246
245 246 248
```

The doses are first imported into a vector with the use of the optional argument `nlines` to the function `scan`, which specifies the maximum number of lines of data to import:

```
> dose <- scan("caffeine.txt", what = 0, nlines = 1)
Read 3 items
```

Next, the remaining data are imported into another vector:

```
> taps <- scan("caffeine.txt", skip = 1)
Read 30 items
```

To create a matrix of stacked data, the vector `dose` needs to be replicated to the same length as the vector `taps`, such that the corresponding elements form an observation:

```
> nc <- length(dose)
> nr <- length(taps)
> ( dose <- rep(dose, nr/nc) )
 [1]   0 100 200   0 100 200   0 100 200   0 100 200
[13]   0 100 200   0 100 200   0 100 200   0 100 200
[25]   0 100 200   0 100 200
```

Finally, the vectors `dose` and `taps` are bound together to form a matrix using the function `cbind`:

```
> caffeine <- cbind(dose, taps)
> class(caffeine)
[1] "matrix"
> dim(caffeine)
[1] 30  2
```

The functions `head` and `tail` may be used to examine the first and last few rows of a matrix, respectively. The optional argument "`n`" specifies the number of rows to display.

```
> head(caffeine, n = 3)
     dose taps
```

 

```
[1,]    0  242
[2,]  100  248
[3,]  200  246
> tail(caffeine, n = 3)
      dose taps
[28,]    0  242
[29,]  100  244
[30,]  200  250
```

Although not necessary, the rows of the matrix `caffeine` may be sorted to reproduce the stacked data shown in Table 3.1a. This will be discussed in Section 3.5. Notice that the first line of the output contains the names of the component vectors. By default, if the component vectors are named, the function `cbind` (or `rbind`) will name the columns (or rows) identically. If this is undesirable, it can be inhibited by specifying the optional argument "`deparse.level = 0`":

```
> caffeine <- cbind(dose, taps, deparse.level = 0)
> head(caffeine, n = 3)
     [,1] [,2]
[1,]    0  242
[2,]  100  248
[3,]  200  246
```

### Indexing Matrices

The following $2 \times 3$ matrix will be used to illustrate indexing, that is, accessing matrix elements via integer or logical indices:

```
> ( mat <- matrix(LETTERS[1:6], ncol = 3) )
     [,1] [,2] [,3]
[1,] "A"  "C"  "E"
[2,] "B"  "D"  "F"
```

The vector `LETTERS` is a built-in constant which contains the 26 upper-case letters of the Roman alphabet; thus, `LETTERS[1:6]` extracts the first six letters.

Recall that a matrix is a vector in disguise. Therefore, it can be indexed just as if one were indexing a vector:

 

```
> mat[3]
[1] "C"
> mat[6]
[1] "F"
```

However, it is more natural and less error prone to use two indices, as in linear algebra, where the row and column indices (in that order) are separated by a comma and enclosed in a pair of brackets:

```
> mat[1, 2]
[1] "C"
> mat[2, 3]
[1] "F"
```

The space after the comma is not required but makes an R code easier to read. To access an entire row (or column) of a matrix, simply leave out the column (or row) index but retain the comma:

```
> mat[2, ]
[1] "B" "D" "F"
> mat[, 2]
[1] "C" "D"
```

Notice that the result of selecting a single row or column is a vector. This may seem reasonable but it could produce unexpected errors. For example, certain functions expect a matrix as an argument, and if the vector is used in a call to one of these functions, any attempt to access its elements using two indices would result in an error. To preserve the matrix structure, the optional argument "`drop = FALSE`" must be specified:

```
> mat[1, 2, drop = FALSE]
     [,1]
[1,] "C"
> mat[2,  , drop = FALSE]
     [,1] [,2] [,3]
[1,] "B"  "D"  "F"
```

More than one column (or row) of a matrix may be selected by specifying a vector of column (or row) indices:

```
> mat[ , c(2, 3)]
     [,1] [,2]
[1,] "C"  "E"
[2,] "D"  "F"
```

 

In order to access multiple elements of a matrix that do not form complete rows or columns, the indices of these elements are specified as a two-column matrix. Each row of the two-column matrix contains the row and column indices of an element to be accessed. For example, suppose the $(1, 1)$, $(2, 2)$ and $(2, 3)$ elements of the matrix `mat` are to be extracted:

```
> ( indx <- matrix(c(1, 1, 2, 2, 2, 3), ncol = 2,
    byrow = TRUE) )
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    2    3
> mat[indx]
[1] "A" "D" "F"
```

### Accessing Elements By Names

The rows and columns of a matrix can be named, as seen in the example on page 68, where the columns of the matrix `caffeine` were named `dose` and `taps` by the function `cbind`. Although it is more natural to refer the rows and columns of a matrix by indices, names can provide a descriptive display in data analysis.

Names may be assigned when a matrix is being created, through the optional argument `dimnames`, specified as a list. The list contains two components, a vector of names for the rows and a vector of names for the columns. Doing so adds an attribute called `dimnames` to the matrix:

```
> ( mat <- matrix(LETTERS[1:6], ncol = 3,
    dimnames = list(c("row 1", "row 2"),
    c("col 1", "col 2", "col 3"))) )
      col 1 col 2 col 3
row 1 "A"   "C"   "E"
row 2 "B"   "D"   "F"
> str(attributes(mat))
List of 2
 $ dim     : int [1:2] 2 3
 $ dimnames:List of 2
  ..$ : chr [1:2] "row 1" "row 2"
  ..$ : chr [1:3] "col 1" "col 2" "col 3"
```

 

The rows and columns of the matrix may now be accessed by names:

```
> mat["row 1", ]
col 1 col 2 col 3
  "A"   "C"   "E"
> mat[, "col 3"]
row 1 row 2
  "E"   "F"
```

With data sets, the rows form observations and are rarely named. In this case, the first component of `dimnames` can be specified as `NULL`:

```
> matrix(LETTERS[1:6], ncol = 3,
  dimnames = list(NULL, c("col 1", "col 2", "col 3")))
     col 1 col 2 col 3
[1,] "A"   "C"   "E"
[2,] "B"   "D"   "F"
```

The easiest way to assign names to the columns and rows of a matrix after it has been created is via the functions `colnames` and `rownames`:

```
> mat <- matrix(LETTERS[1:6], ncol = 3)
> colnames(mat) <- c("col 1", "col 2", "col 3")
> rownames(mat) <- c("row 1", "row 2")
> mat
      col 1 col 2 col 3
row 1 "A"   "C"   "E"
row 2 "B"   "D"   "F"
```

The attribute `dimnames`, and hence all names, can be removed with the assignment of `NULL` using the function `dimnames`:

```
> dimnames(mat) <- NULL
> mat
     [,1] [,2] [,3]
[1,] "A"  "C"  "E"
[2,] "B"  "D"  "F"
```

Notice, again, that `dimnames` is simultaneously the name of an optional argument, an attribute, and a function to access (retrieve or set) that attribute.

 

### Applying Functions To Rows And Columns

In data analysis, it is often necessary to apply a function to a matrix column-wise or row-wise. For example, consider again the data on the effect of caffeine on the performance of a simple task, finger tapping. The number of finger taps per minute were previously imported into the vector `taps` (page 68). The unstacked format of the data will be used for illustration:

```
> caffeine <- matrix(taps, ncol = 3, byrow = TRUE,
  dimnames = list(NULL, c("0 ml", "100 ml", "200 ml")))
> head(caffeine, n = 2)
     0 ml 100 ml 200 ml
[1,]  242    248    246
[2,]  245    246    248
```

Applying the function `mean` to the matrix gives the overall sample mean:

```
> mean(caffeine)
[1] 246.5
```

To compute the group sample means, the function `mean` has to be applied column-wise to the matrix. The function `apply` can be used to accomplish the task:

```
> args(apply)
function (X, MARGIN, FUN, ...)
```

The first argument, `X`, is a matrix. The second argument, `MARGIN`, specifies whether the columns or rows are to be passed on to the function specified by the third argument, `FUN`; a value of `1` indicates rows while a value of `2` indicates columns.

```
> apply(caffeine, 2, mean)
  0 ml 100 ml 200 ml
 244.8  246.4  248.3
```

Note that, in this case, the function `colMeans` should have been used instead to calculate the column means:

```
> colMeans(caffeine)
  0 ml 100 ml 200 ml
 244.8  246.4  248.3
```

 

The reason is that `colMeans` is vectorized whereas `apply` uses a for loop to perform the calculations, hence slower.

The symbol "`...`" in the formal argument list, called *ellipsis* or simply "dot-dot-dot", is a special argument that can contain any number of arguments. It allows additional arguments to be passed on to another function. For example, the function `mean` has an optional argument `trim` which specifies the proportion (0 to 0.5) of observations to be trimmed from each end of a sorted data set before the mean is computed (one of the advantages of the trimmed mean is its robustness with respect to outliers). To compute the trimmed mean:

```
> apply(caffeine, 2, mean, trim = 0.1)
   0 ml  100 ml  200 ml
244.750 246.375 248.250
```

## Data Frames

A *data frame* is a structure that is tailored for tabular data. Like a matrix, it arranges data in rows and columns. Unlike a matrix, different columns of a data frame can contain values of different types. It turns out that a data frame is a list in disguise, and, for most practical purposes, the components of the list are vectors restricted to be of the same length.

### Data Frames Are Lists

A data frame can be created just as a list is created, using the function `data.frame` in place of `list`:

```
> ( wonder.pets <- data.frame(
      name = c("Linny", "Tuck", "Ming-Ming"),
      age  = c(5, 4, 3),
      common.name = c("guinea pig", "turtle", "duckling")) )
       name age common.name
1     Linny   5  guinea pig
2      Tuck   4      turtle
3 Ming-Ming   3    duckling
> class(wonder.pets)
[1] "data.frame"
```

 

```
> typeof(wonder.pets)
[1] "list"
```

The last line of the preceding display shows that a data frame is indeed a list. It follows that the columns of a data frame may be accessed in the same manner the components of a list are accessed, either by indexing with a pair of double brackets or by names:

```
> wonder.pets[[1]]
[1] Linny     Tuck      Ming-Ming
Levels: Linny Ming-Ming Tuck
> wonder.pets$name
[1] Linny     Tuck      Ming-Ming
Levels: Linny Ming-Ming Tuck
```

It may come as a surprise that the vector `name`, intended as a character vector, is stored as a factor. So is the vector `common.name`:

```
> class(wonder.pets$common.name)
[1] "factor"
```

By default, all character vectors are converted to factors when a data frame is created. This can be prevented by specifying the optional argument "`stringsAsFactor = FALSE`":

```
> wonder.pets <- data.frame(
    name = c("Linny", "Tuck", "Ming-Ming"),
    age  = c(5, 4, 3),
    common.name = c("guinea pig", "turtle", "duckling"),
    stringsAsFactors = FALSE)
> class(wonder.pets$name)
[1] "character"
> class(wonder.pets$common.name)
[1] "character"
```

The functions `lapply` and `sapply` are used to apply a function to a data frame column-wise, since the columns of a data frame are the components of the underlying list. For example:

```
> sapply(wonder.pets, class)
       name         age common.name
"character"   "numeric" "character"
```

 

### Data Frames Are Like Matrices

Data frames share many properties of matrices. Firstly, the elements of a data frame may be accessed using matrix indexing:

```
> wonder.pets[1, ]
   name age common.name
1 Linny   5  guinea pig
> wonder.pets[, 1]
[1] "Linny"     "Tuck"      "Ming-Ming"
```

Notice that the first indexing operation returns a data frame whereas the second returns a vector. Generally, when an object has more than one dimension, the simplest possible data structure is used to store the result of an indexing operation. In the first case, the elements are of different types and they cannot be stored as an atomic vector without coercion. In the second case, the optional argument `drop = FALSE` may be used to preserve the data frame structure:

```
> wonder.pets[, 1, drop = FALSE]
       name
1     Linny
2      Tuck
3 Ming-Ming
```

Secondly, the functions `cbind` or `rbind` may be used to append a column or row, respectively, to an existing data frame. To append a column, the number of rows must match:

```
> cbind(wonder.pets, is.mammal = c(TRUE, FALSE, FALSE))
       name age common.name is.mammal
1     Linny   5  guinea pig      TRUE
2      Tuck   4      turtle     FALSE
3 Ming-Ming   3    duckling     FALSE
```

Similarly, to append a row, the number of columns must match:

```
> rbind(wonder.pets, list("Ollie", 4, "rabbit"))
       name age common.name
1     Linny   5  guinea pig
2      Tuck   4      turtle
3 Ming-Ming   3    duckling
4     Ollie   4      rabbit
```

 

Two or more data frames may be combined in a similar manner.

Thirdly, data frames have column names and row names as attributes, named `names` and `row.names`, respectively, and these can be accessed using the functions `colnames` and `rownames`:

```
> str(attributes(wonder.pets))
List of 3
 $ names    : chr [1:4] "name" "age" "common.name" "is.mammal"
 $ class    : chr "data.frame"
 $ row.names: int [1:5] 1 2 3 4 5
> colnames(wonder.pets)
[1] "name"        "age"         "common.name"
> rownames(wonder.pets)
[1] "1" "2" "3"
```

### Importing and Exporting Data

The function `read.table` imports tabular data into R and returns a data frame. The first argument, a file name or URL, is the only required argument if the data are arranged in the standard format:

- Each observation appears as one line of the file.
- Each line contains the same number of values, separated by one or more spaces or tabs, where each value corresponds to a variable.
- Missing values are represented as `NA`.

To illustrate, consider importing the following data, saved as a plain-text file named "`wonderPets1.txt`" in the working directory:

wonderPets1.txt

```
Linny          5  "guinea pig" TRUE
Tuck           4  turtle       FALSE
Ming-Ming      3  duckling     FALSE
Ollie          4  rabbit       TRUE
"The Visitor" NA alien        NA
```

```
> ( wonder.pets <- read.table("wonderPets1.txt") )
           V1 V2         V3    V4
1       Linny  5 guinea pig  TRUE
```

 

```
2        Tuck  4     turtle FALSE
3   Ming-Ming  3   duckling FALSE
4       Ollie  4     rabbit  TRUE
5 The Visitor NA      alien    NA
> class(wonder.pets)
[1] "data.frame"
```

There are several optional arguments to accomodate file formats that deviate from the standard, some of which are familiar: `sep`, `skip`, and `comment.char` (see pages 55–56). A selected few of the remaining optional arguments are described as follows.

**`header`** By default, the columns are named "`V1`", "`V2`", and so on, where "`V`" stands for "variable":

```
> colnames(wonder.pets)
[1] "V1" "V2" "V3" "V4"
```

These can be changed afterward using the function `colnames`. Alternatively, the first line of the file may include the column names, in which case the argument `header = TRUE` must be specified. For example:

wonderPets2.txt

```
name age common.name is.mammal
Linny         5  "guinea pig" TRUE
Tuck          4  turtle       FALSE
Ming-Ming     3  duckling     FALSE
Ollie         4  rabbit       TRUE
"The Visitor" NA alien        NA
```

```
> wonder.pets <- read.table("wonderPets2.txt",
  header = TRUE)
> colnames(wonder.pets)
[1] "name"        "age"         "common.name" "is.mammal"
```

**`as.is`** As with the function `data.frame`, all character vectors are automatically converted to factors:

```
> sapply(wonder.pets, class)
       name         age common.name   is.mammal
   "factor"   "integer"    "factor"   "logical"
```

 

This can be inhibited with the argument `as.is`, either for all columns by specifying `as.is = TRUE`:

```
> wonder.pets <- read.table("wonderPets2.txt",
  header = TRUE, as.is = TRUE)
> sapply(wonder.pets, class)
       name         age common.name   is.mammal
"character"   "integer" "character"   "logical"
```

or for selected columns by supplying a vector of column indices:

```
> wonder.pets <- read.table("wonderPets2.txt",
  header = TRUE, as.is = 3)
> sapply(wonder.pets, class)
       name         age common.name   is.mammal
   "factor"   "integer" "character"   "logical"
```

Alternatively, conversion of character vectors to factors can be suppressed for the entire R session using `options("stringsAsFactors" = FALSE)`.

**`colClasses`** This argument is similar to the argument `what` of the function `scan`. It specifies the desired class for each column, which can be one of the six classes of atomic vectors ("`numeric`, "`character`", and so on) or "`factor`". It can also be used to skip one or more columns by specifying "`NULL`" as the class. For example, to skip the third column, "`common.name`":

```
> wonder.pets <- read.table("wonderPets2.txt",
  header = TRUE,
  colClasses = c("character", "numeric", "NULL", "logical"))
> sapply(wonder.pets, class)
       name         age   is.mammal
"character"   "numeric"   "logical"
```

**`strip.white`** *Comma-separated-values* (CSV) file format is frequently used to export data from spreadsheets. As the name suggests, commas are typically used to separate values. If a user manually creates a CSV file and (habitually) adds a space after each comma, the data may not be imported correctly. This is illustrated using the following file:

 

wonderPets3.txt

```
name, age, common.name, is.mammal
Linny, 5, guinea pig, TRUE
Tuck, 4, turtle, FALSE
Ming-Ming, 3, duckling, FALSE
Ollie, 4, rabbit, TRUE
The Visitor, NA, alien, NA
```

```
> wonder.pets <- read.table("wonderPets3.txt",
  header = TRUE, as.is = TRUE, sep = ",")
> sapply(wonder.pets, class)
       name         age common.name   is.mammal
"character" "character" "character" "character"
```

The second and fourth columns are incorrectly imported as character vectors. The reason is that the spaces are considered to be part of the values that follow a comma; as a result, the value " NA" is interpreted as a string rather than a missing value, and the numerical values are coerced to strings:

```
> wonder.pets$age
[1] " 5"  " 4"  " 3"  " 4"  " NA"
```

The argument `strip.white` instructs `read.table` to remove leading and trailing spaces from unquoted character values:

```
> wonder.pets <- read.table("wonderPets3.txt",
  header = TRUE, as.is = TRUE, sep = ",", strip.white = TRUE)
> sapply(wonder.pets, class)
       name         age common.name   is.mammal
"character"   "integer" "character"   "logical"
```

There are several wrapper functions[1] to the function `read.table`. For example, the functions `read.csv` and `read.csv2` set the default value of "`header`" as "`TRUE`", and that of "`sep`" as comma and semicolon, respectively. These functions accepts all the optional arguments to `read.table`.

Like every object in R, a data frame can be saved to a file in a binary format (see Section 2.8). However, some users may prefer to save the data in a format that can be viewed or edited in a text editor. This can be done using the function `write.table`. For example:

[1] A wrapper function is a function that simplifies the calling of another function, and performs little or no additional computation.

 

```
> write.table(wonder.pets, file = "wonderPets3Out.txt")
```

This produces the following text file in the working directory.

wonderPets3Out.txt

```
"name" "age" "common.name" "is.mammal"
"1" "Linny" 5 "guinea pig" TRUE
"2" "Tuck" 4 "turtle" FALSE
"3" "Ming-Ming" 3 "duckling" FALSE
"4" "Ollie" 4 "rabbit" TRUE
"5" "The Visitor" NA "alien" NA
```

If an absolute pathname (see Section 1.3) is specified, it should be kept in mind that the function `write.table` cannot create new folders on your computer. In other words, all folders in the absolute pathname must already exist or R will signal an error:

```
> write.table(wonder.pets, file = "/toonLand/wonderPetsOut.txt")
Error in file(file, ifelse(append, "a", "w")) :
  cannot open the connection
...
```

Below are some options for customizing the output.

**`quote`** By default, if the class of a column is "`character`" or "`factor`", the corresponding elements will be surrounded by quotes. Quoting can be disabled by specifying `quote = FALSE`:

```
> write.table(wonder.pets, file = "wonderPets3Out.txt",
  quote = FALSE)
```

**`col.names`** and **`row.names`** The column names and row names are written to the file unless otherwise indicated using the arguments `col.names = FALSE` and `row.names = FALSE`. For example, to suppress the writing of row names:

```
> write.table(wonder.pets, file = "wonderPets3Out.txt",
  row.names = FALSE)
```

Notice, in the preceding display of the file "`wonderPets3Out.txt`", that the column containing the row names does not have a column name. This could create a problem if the file is imported by a program that expects a name for each column. The argument `col.names = NA` instructs `write.table` to add an empty string to the header line for the row names, provided `quote = TRUE`.

 

### Stacking and Unstacking

Once a data frame has been created or imported into R, it may be necessary to *reshape* the data frame, such as stacking or unstacking its columns, as discussed in the beginning of Section 3.4. R provides the functions `stack` and `unstack` to perform these operations.

For ease of illustration, we use a small subset of the data on the effect of caffeine on the performance of a simple task (page 73):

```
> data1.u
  0 ml 100 ml 200 ml
1  242    248    246
2  245    246    248
```

Throughout, the suffix "`u`" serves to remind us that the data are unstacked and the suffix "`s`" indicates that the data is stacked. To stack `data1.u`:

```
> ( data1.s <- stack(data1.u) )
  values    ind
1    242   0 ml
2    245   0 ml
3    248 100 ml
4    246 100 ml
5    246 200 ml
6    248 200 ml
```

The data frame `data1.s` contains two columns that are automatically named "`values`" and "`ind`" (which stands for "indicator"). The former contains the result of stacking the rows of `data1.u`, and the latter contains the corresponding column names in `data1.u`.

As a slightly more elaborate example, suppose that the subjects of the study included both female and male students, and we append an additional column to `data1.u` to incorporate this information:

```
> ( data2.u <- cbind(data1.u, sex = c("female", "male")) )
  0 ml 100 ml 200 ml    sex
1  242    248    246 female
2  245    246    248   male
```

It is no longer meaningful to stack all the columns. The function `stack` accepts an optional argument, `select`, which specifies the columns that are to be stacked; in this example, it would be the first three columns:

 

```
> ( data2.s <- stack(data2.u, select = 1:3) )
  values    ind
1    242   0 ml
2    245   0 ml
3    248 100 ml
4    246 100 ml
5    246 200 ml
6    248 200 ml
```

For a simple two-column data frame such as `data1.s`, reversing the process of stacking is straightforward:

```
> ( data1.s.u <- unstack(data1.s) )
  X0.ml X100.ml X200.ml
1   242     248     246
2   245     246     248
```

The column names of `data1.s.u` look puzzling. This puzzle is left as an exercise. When there are more than two columns in a stacked data frame, the optional argument `form`, which stands for "formula", must be supplied. In R, the operator "`~`" is used to express a relationship between two or more objects, where the exact meaning of "relationship" depends upon the function that is called. With the function `unstack`, for example, the symbol on the left-hand side of "`~`" indicates the column to be unstacked, while the symbol on the right-hand side indicates the column whose distinct values are used to create groups. To illustrate, we begin by appending an additional column to `data2.s`:

```
> ( data2.s <- cbind(data2.s, sex = c("female", "male")) )
  values    ind    sex
1    242   0 ml female
2    245   0 ml   male
3    248 100 ml female
4    246 100 ml   male
5    246 200 ml female
6    248 200 ml   male
> unstack(data2.s, values ~ ind)
  X0.ml X100.ml X200.ml
1   242     248     246
2   245     246     248
```

 

```
> unstack(data2.s, values ~ sex)
  female male
1    242  245
2    248  246
3    246  248
```

The functions `stack` and `unstack` perform basic reshaping of data frames. For more complex reshaping, R provides the function `reshape`. There are also user-contributed functions such as `cast` and `melt`, which can be downloaded from CRAN and requires installation (see Section 3.6, page 90).

## 3.5 Sorting Data

A common operation in data analysis is *sorting*, whereby data are rearranged in ascending or descending order with respect to a variable. A numeric or integer vector can be sorted using the function `sort`:

```
> x <- c(3, 1, 4, NA, 1, 5, 9)
> sort(x)
[1] 1 1 3 4 5 9
```

By default, the values are sorted in ascending order and "`NA`" values are omitted. Note that the vector `x` remains unchanged:

```
> x
[1]  3  1  4 NA  1  5  9
```

It follows that the sorted vector must be assigned a name in order to be accessible. To sort `x` in decreasing order, specify the optional argument "`decreasing`" as "`TRUE`":

```
> sort(x, decreasing = TRUE)
[1] 9 5 4 3 1 1
```

The optional argument "`na.last`" controls the handling of "`NA`" values:

```
> sort(x, na.last = TRUE)
[1]  1  1  3  4  5  9 NA
> sort(x, na.last = FALSE)
[1] NA  1  1  3  4  5  9
```

 

A related function is `order`, which takes a vector as the first argument, and returns an index vector that can be used to select the elements of the given vector in ascending or descending order. It also accepts the optional arguments, `decreasing` and `na.last`. For example:

```
> ( sort.index <- order(x, na.last = TRUE) )
[1] 2 5 1 3 6 7 4
```

To understand the return value, `sort.index`, consider the following schematic representation of the vector `x`:

| 3 | 1 | 4 | NA | 1 | 5 | 9 |
|---|---|---|---|---|---|---|
| x[1] | x[2] | x[3] | x[4] | x[5] | x[6] | x[7] |

The $i$th element of the vector `sort.index` gives the index of the $i$th smallest value in the vector `x`; ties are left in their original ordering. For example, the smallest value in `x`, the value `1`, occurs in the second and fifth position; therefore, the first two elements of `sort.index` are `2` (second position) and `5` (fifth position). The next smallest value is `3`, which occurs in the first position in the vector `x`, and so the third element of `sort.index` is `1`. Using `sort.index` as an index vector to select the elements of `x` returns a vector sorted in ascending order:

```
> x[sort.index]
[1]  1  1  3  4  5  9 NA
```

While the use of the function `order` to sort a vector may seem like a roundabout way, it is the only means to sort a matrix or a data frame. We use the data frame `data2.s` created on page 82 for illustration (the columns are renamed to provide a descriptive display). The function `sort` cannot be applied to `data2.s` since it operates on a single vector:

```
> sort(data2.s)
Error in `[.data.frame`(x, order(x, na.last = na.last,
decreasing = decreasing)) :
  undefined columns selected
```

Using the function `lappy` to sort `data2.s` column-wise would not be meaningful since the correspondence between the columns would be destroyed:

 

```
> lapply(data2.s, sort)
$taps
[1] 242 245 246 246 248 248

$dose
[1] 0 ml   0 ml   100 ml 100 ml 200 ml 200 ml
Levels: 0 ml 100 ml 200 ml

$sex
[1] female female female male   male   male
Levels: female male
```

To sort a matrix or a data frame, the function `order` is first used to obtain an index vector that will select the rows in the desired order. Sorting is then achieved by selecting the rows using the index vector. For example, suppose the rows of `data2.s` are to be sorted by increasing values of the first column, `taps`:

```
> index1 <- order(data2.s[, 1])
> data2.s[index1, ]
  taps   dose    sex
1  242   0 ml female
2  245   0 ml   male
4  246 100 ml   male
5  246 200 ml female
3  248 100 ml female
6  248 200 ml   male
```

Notice that there are tied values, `246` and `248`, in the first column; these are left in their original ordering. Additional vectors may be supplied to the function `order` to break the ties:

```
> index2 <- order(data2.s[, 1], data2.s[, 3])
> data2.s[index2, ]
  taps   dose    sex
1  242   0 ml female
2  245   0 ml   male
5  246 200 ml female
4  246 100 ml   male
3  248 100 ml female
6  248 200 ml   male
```

 

Here the column `sex` is used to break the ties and "`female`" comes before "`male`". The function `order` may be supplied with any number of vectors as long as they have the same number of rows; the vectors are used sequentially to break any unresolved ties.

## 3.6 Built-in Data Sets and Packages

R contains many built-in data sets, a list of which can be displayed by typing the command `data()` at the console; the output is shown in Figure 3.2. These data sets are directly accessible by name. For example, the first data set on the list is named "`AirPassengers`":

```
> str(AirPassengers)
 Time-Series [1:144] from 1949 to 1961: 112 118 132 129 121 135
 148 148 136 119 ...
```

Built-in data sets come as part of a bundle referred to as a *package*, which is a collection of related files organized into folders. These files may contain R code or compiled code written in other languages (or both), illustrative data sets, and documentation, among others.

### Loading Packages

A standard set of packages is included with R, such as the package that contains the data sets described in the preceding paragraph, named "`datasets`". To list all available packages, the function `library` is called without any arguments:

```
> library()
```

This brings up a window similar to that shown in Figure 3.3, which lists the names of the packages along with a brief description. The command `help(package = "`*`package name`*`")` is used to display the documentation for a specified package in the help window, for example, `help(package = "datasets")`.

Not all available packages are loaded into the computer memory when R starts, since time and memory would be wasted on loading packages that are not used. The content of a package is not accessible until it is loaded and attached to the *search path*, a set of locations that R will search for an

 

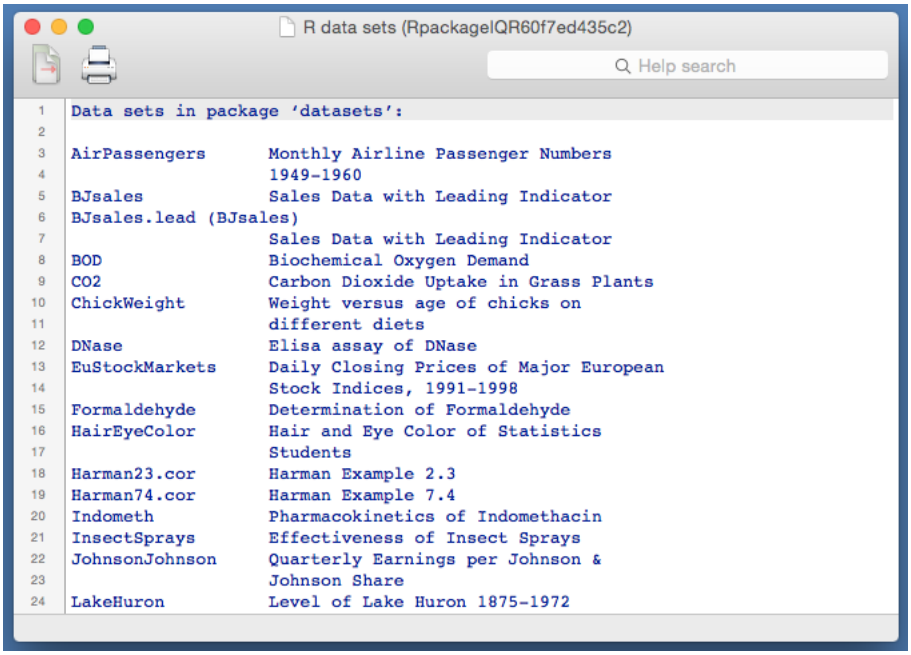


Figure 3.2: Built-in data sets.

object. To find out the packages that are currently loaded and attached to the search path:

```
> search()
 [1] ".GlobalEnv"        "tools:RGUI"        "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

R searches for an object by traversing the search path in the order listed. The global environment or workspace, "`.GlobalEnv`", is searched first and the package "`base`" is searched last. For example, if a function named "`c`" is defined in the workspace, it will take precedence over the concatenate function `c`, which is found in the package "`base`":

```
> c <- function(x) x + 1
> c(3, 1, 4, 1, 5, 9)
Error in c(3, 1, 4, 1, 5, 9) : unused arguments (1, 4, 1, 5, 9)
```

 

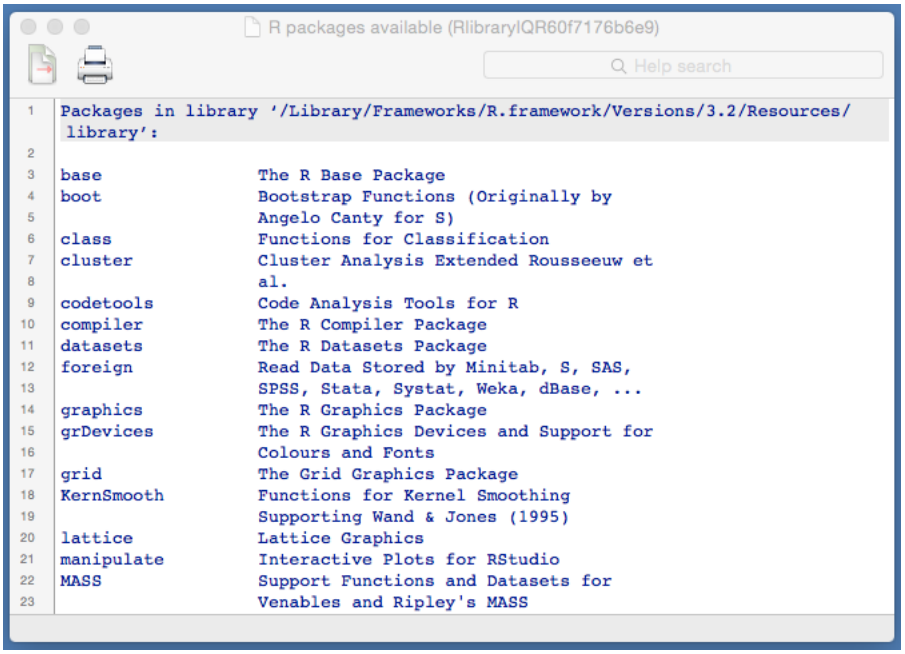


Figure 3.3: Packages included with R.

Here the user-defined function is said to *mask* the concatenate function. To unmask the concatenate function, simply remove the user-defined function from the workspace:

```
> rm(c)
> c(3, 1, 4, 1, 5, 9)
[1] 3 1 4 1 5 9
```

A package can be loaded into the memory without being attached to the search path, but it is rare to do so. The function `library` is used to load and attach a package by supplying the name of the package as an argument, with or without quotes. Consider the package "`MASS`", which was developed to accompany the book *Modern applied statistics with S* by Venables and Ripley [12]. The package contains many illustrative data sets, one of which is `geyser`, a data frame containing 299 observations of the waiting time for the next eruption of the Old Faithful geyser in Yellowstone National Park, as well as the duration of these eruptions. Before the package is attached, any attempt to access its content produces an error:

 

```
> str(geyser)
Error in str(geyser) : object 'geyser' not found
> library(MASS)
> str(geyser)
'data.frame':        299 obs. of  2 variables:
 $ waiting : num  80 71 57 80 75 77 60 86 77 56 ...
 $ duration: num  4.02 2.15 4 4 4 ...
```

Whenever a package is attached, it is inserted after the workspace and before previously attached packages:

```
> head(search(), n = 5)
[1] ".GlobalEnv"       "package:MASS"     "tools:RGUI"
[4] "package:stats"    "package:graphics"
```

This may result in some functions in the existing packages being masked by functions of the same name in the newly loaded package. If this is undesirable or the masked functions are to be restored without restarting R, simply *detach* the package:

```
> detach(package:MASS, unload = TRUE)
> head(search(), n = 5)
[1] ".GlobalEnv"        "tools:RGUI"        "package:stats"
[4] "package:graphics"  "package:grDevices"
```

## Installing and Removing Packages

There are thousands of packages contributed by users from all over the world, which are available for download from CRAN (https://cran.r-project.org/web/packages). It is impossible to learn to use all the packages, most of which are developed to address a specific problem. It is also time consuming to wade through all the packages to find one that will address the problem at hand. A better approach to find a package would be to ask a question on, for example, Stack Overflow, or use search engines such as Google.

To illustrate the installation of packages, consider the package `reshape`, which provides functions for reshaping data frames (see page 84). The package documentation appears as one of the first few results returned by Google when the phrase "R package reshape data frame" is searched. One way to install the package is to use the function `install.packages`. If this is the

 

first time a package is installed, R will prompt the user to select a *mirror*, a website that is identical to the original (CRAN, by default) but placed under a different URL:

```
> install.packages("reshape")
--- Please select a CRAN mirror for use in this session ---
```

A window with a list of mirrors should be presented for selection; selecting one that is geographically close to the user's location can potentially increase the speed of download. Upon selection, installation should proceed, and the package will be installed to a location specified by the object `.Library`. For example, on the author's computer:

```
> .Library
[1] "/Library/Frameworks/R.framework/Resources/library"
```

To remove the package, locate the directory specified by `.Library` and delete the folder named "`reshape`". Alternatively:

```
> remove.packages("reshape", .Library)
Updating HTML index of packages in '.Library'
Making 'packages.html' ... done
```

## 3.7 Exercises

1. Create the following plain-text file in your working directory and import the data using the function `scan`.

```
Linny,Tuck,
Ming-Ming,Ollie,
The Visitor
```

dataset3b.txt

   Explain why the imported data contain two empty strings:

```
> scan("dataset3b.txt", what = character(), sep = ",")
Read 7 items
[1] "Linny"       "Tuck"        ""            "Ming-Ming"
[5] "Ollie"       ""            "The Visitor"
```

 

2. Refer to page **??**. When the data contained in `caffeine.txt` is imported using the function `read.table` with the option `header = TRUE`, an "`X`" is prepended to the column names supplied in the first line of the file. Explain.

 

# Chapter 4

# Visualizing Data

 

# Bibliography


[1] Adler, J. (2010). *R in a Nutshell: A Desktop Quick Reference*. O'Reilly.

[2] Burns, P. (2012). *The R Inferno*. Chapman & Hall.

[3] Chambers, J. M. (2008). *Software for Data Analysis: using computer software: Programming with R*. Springer.

[4] Cotton, R. (2013). *Learning R*. O'Reilly.

[5] Draper, N. R. and H. Smith (1998). *Applied Regression Analysis* (3rd ed.). John Wiley & Sons.

[6] Gentleman, R. (2009). *R Programming for Bioinformatics*. Chapman & Hall.

[7] Jones, O., R. Maillardet, and A. Robinson (2009). *Introduction to Scientific Programming and Simulation Using R*. Chapman & Hall.

[8] Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press.

[9] R Core Team (2015). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. http://www.R-project.org/.

[10] Revolution Analytics (2015). Companies using R. http://www.revolutionanalytics.com/companies-using-r. Accessed on February 2015.

[11] Spector, P. (2008). *Data Manipulation with R*. Springer.

[12] Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*. Springer.

[13] W. Venables, D. Smith and R Core Team (2015). *An Introduction to R Version 3.2.1 (2015-06-18)*. https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf.

[14] Wickham, H. (2014). Tidy data. *Journal of Statistical Software 59*. http://www.jstatsoft.org/v59/i10/paper.

[15] Wickham, H. (2015). *Advanced R*. CRC Press.

 