An Introduction to the Fundamentals & Functionality of the R Language

SECTION I: Some Language Essentials

Theresa A. Scott, MS
Biostatistician III
Department of Biostatistics
Vanderbilt University

theresa.scott@vanderbilt.edu

Table of Contents

Preface

Unlike some existing R documentation, these documents are not intended to be companion manuscripts to an introductory statistics course. The purpose of these documents is to introduce the fundamentals and functionality of the R language, not statistical concepts and techniques.

This document and its three companion documents are available on my website, `http://biostat.mc.vanderbilt.edu/TheresaScott`, under the Current Teaching Material heading. These documents are by no means an 'original' piece of work. I have merely assembled much of the existing documentation into my own group of documents.

I hope you find my group of documents helpful and allow you to start using R confidently. Feel free to contact me at `theresa.scott@vanderbilt.edu` with any questions and/or comments. I also welcome any suggestions and (constructive) criticism. Lastly, I would like to thank everyone who has come to me with questions in the past, and who attend my weekly R Clinic – I continue to learn something new about R every day. GOOD LUCK!

This document was written using the following versions of R, the operating system, and add-on packages:

- Version 2.3.1 (2006-06-01), `i486-pc-linux-gnu`
- Base packages: base, datasets, graphics, grDevices, methods, stats, tools, utils
- Other packages: chron 2.3-6, foreign 0.8-15, Hmisc 3.0-12

Chapter 1

Introduction

R is a *free interactive programming language and environment*, created as an integrated suite of software facilities for data manipulation, calculation, and graphical display. Even though many people use R as a statistics system, the creators prefer to think of it as an environment within which many classical and modern statistical techniques have been implemented.

R is an independent implementation of the S programming language. The S language has also been used to develop S-PLUS, a commercial product distributed by the Insightful Corporation. Although there are some minor differences between R and S-PLUS (mostly in the graphical user interface), they are essentially identical.

The benefits of using R include:

- Its availability as *free* software – 'free' in terms of price, and (more importantly) in terms of freedom to run, copy, distribute, study, change, and improve the software.
- Its ability to run on several UNIX platforms, Linux platforms, Windows, and MacOS.
- Its excellent graphing capabilities.
- The ability to completely reproduce your analysis and results, if properly documented, since the language is code driven. This is not always true with menu driven analysis packages; they are often much harder to document.
- Its extreme flexibility and extendability.
  - Not limited to a simple interface in terms of menus and forms.
  - Hundreds of libraries of functions to use, as well as the ability to write your own functions.
  - Unlike some classical software programs (e.g., SAS and SPSS), which display all the results of an analysis, R allows you to assign any results to a (symbolic) variable, so that an analysis can be done with minimal or no output, and the parts of the results of interest can be extracted and used in subsequent analyses.

1.1 Installing R

Both the complete instructions and the necessary accompanying files needed to install R are distributed by the *Comprehensive R Archive Network* (CRAN) on their website `http://www.cran.r-project.org`.[1] The *source code*, which allows users to compile the program to their liking, is available for Linux/Unix platforms,

[1] CRAN is a collection of sites which carry identical materials, consisting of the R distribution(s), the contributed extensions, documentation for R, and binaries. The sites were created as mirror sites to lessen the load on any one server, so choose a site close to you. A list can be found at `http://www.cran.r-project.org/mirrors.html`.

while a convenient *pre-compiled binary package* is available for Windows (95 and later) and MacOS X users.

See any of the following for detailed instructions – the first three are available through links from the Documentation section of the CRAN or R websites (`http://www.r-project.org`):

- The '*R Installation and Administration*' document (pdf) – the Manuals link.
- The FAQ links (both general and Windows and Mac specific).
- The '*Installing R under Windows*' article in the June 2001 (Vol. 1/2; pages 11 -14) edition of the *R News* newsletter – the Newsletter link.

1.2 Existing R documentation

The majority of existing R documentation can be found through the Documentation section of the R website – only a few books may exist that are published but not listed. It is important to know that the information given in the links is updated often.

All of the following manuals are downloadable as pdfs through the Manuals link:

- *'An Introduction to R'* by WN Venables, DM Smith, and the R Development Core Team includes information on data types, programming elements, statistical modeling, and graphics (also a book).
- *'The R Language Definition'* by the R Development Core Team explains evaluation, parsing, object oriented programming, computing in the language, and so forth.
- *'Writing R Extensions'* by the R Development Core Team covers how to create R add-on packages, write R help files and documentation, and the foreign language interfaces (e.g. C, C++, and Fortran).
- *'R Data Import/Export'* by the R Development Core Team is a guide to importing and exporting data to and from R.
- *'R Installation and Administration'* by the R Development Core Team gives detailed instructions for installing R and its packages.
- *'The R Reference Index'* 'contains all help files of the R standard and recommended packages in printable form' (very long).

The FAQs link contains three collections of answers to frequently asked questions. The R FAQ is the general collection and contains useful information for users on all platforms (Linux, Mac, Unix, Windows), and is downloadable as a pdf. The R FAQ also gives additional references to existing documentation for R and S/S-Plus than listed here. The R MacOS X FAQ is specific to the Apple operating systems, and the R Windows FAQ is specific to the Microsoft operating systems. These latter two are complementary to the general R FAQ, implying you should read both the general FAQ and your platform-specific one.

The *R News* Newsletter, accessed via the Newsletter link, features short to medium length articles covering topics that might be of interest to users or developers of R, including

- *Changes in R:* new features of the latest release.
- *Changes on CRAN:* new add-on packages, manuals, binary distributions, mirrors, and more.
- *Add-on packages:* short introductions to or reviews of R extension packages.
- *Programmer's Niche:* nifty hints for programming in R (or S).
- *Hints for newcomers:* Explaining sides of R that might not be so obvious from reading the manuals and FAQs.

- *Applications:* Examples of analyzing data with R.

The Wiki link transfers you to the *R Wiki* website (`http://wiki.r-project.org/`), which is a Wiki website[2] dedicated to the collaborative writing of R documentation. The website contains a slowly growing wealth of information including information about getting started, graphs, and packages.

The Books link contains a list of books related to R (and S) including

- *'Introductory Statistics with R'* by Peter Dalgaard.
- *'An R and S-Plus Companion to Multivariate Analysis'* by Brian Everitt.
- *'A Handbook of Statistical Analyses using R'* by Brian Everitt and Torsten Hothorn.
- *'Linear Models with R'* and *'Extending Linear Models with R'* by Julian Faraway.
- *'An R and S-Plus Companion to Applied Regression'* by John Fox.
- *'Regression Modeling Strategies'* by Frank E Harrell.
- *'Data Analysis and Graphics Using R'* by John Maindonald and John Braun.
- *'R Graphics'* by Paul Murrell.
- *'Using R for Introductory Statistics'* by John Verzani.

Lastly the Other link contains further links to HTML versions of the manuals and help pages, contributed documentation, other R related publications, and much more.

[2] A wiki website is a type of website that allows visitors to easily add, remove, or otherwise edit and change some available content.

Chapter 2

Sample R session: Using R as a calculator

Before going further, it is helpful to interact with R on the simplest level – that is, using it as a calculator. The following session is intended to introduce you to some of the features of the R environment by using them.

2.1 Starting R

When using the Windows version of R, launch R by double-clicking the R icon on the desktop, or by finding the R program under the start menu. This will start R in a new *console window* with a *command line subwindow*.

When using R under Unix/Linux, launch R by simply typing 'R' at the shell command prompt (i.e., '$ R' if we assume that the shell prompt is '$'). This will cause R to start up as an interactive program in the current terminal window.

2.2 Entering expressions

There are several ways to interact with R, but the simplest is to type *expressions* at the cursor following the *command line prompt*, which is denoted by the "greater than" symbol, >. To evaluate the expression, we simply press the ENTER key.

The simplest expressions to enter at the command line are arithmetic expressions involving numbers and algebraic operators. For example,

```
> 2 * 10
[1] 20
> 5/3
[1] 1.666667
> 10 + 13 - 21
[1] 2
> 2^3
[1] 8
```

The output of each evaluated expression may appear odd. Specifically, the `[1]` in front of the result is part of R's way of printing intrinsic *data structures* to the screen. Don't worry if it doesn't make a lot of sense right now; we'll cover data structures in detail in future sections.

ASIDE: There are several packages and projects, for example the `Rcmdr` package, the JGR project (`http://stats.math.uni-augsburg.de/JGR/`), and the SciViews-R project (`http://www.sciviews.org/SciViews-R/`), which provide more sophisticated windows-and-dialogs front-ends to R. These lecture notes focus on interacting with R by typing expressions at the command line.

2.3 Expression evaluation

R evaluates expressions using a type of *question-and-answer model*. Specifically, when you type an expression at the command line prompt and press ENTER, the command is first transformed by R into an internal representation. If the expression is syntactically complete, the transformed expression is then executed and R returns (prints) the result value of the expression to the screen (if relevant).[1] Once executed, R then asks for more input by printing the command line prompt and cursor.

INCOMPLETE EXPRESSIONS: If an expression is not syntactically complete when ENTER is pressed, R will print the *continuation prompt*, `+`, at the beginning of the second and subsequent lines and continue to read input until the expression is syntactically complete. For example, type in the mathematical expression `2 + 3 + 5 - 2` at the command line by hitting the `ENTER` key after each arithmetic operator:

```
> 2 +
+ 3 +
+ 5 -
+ 2
[1] 8
```

In the Windows version of R, use the Esc key to cancel an incomplete expression, which will print a new command line prompt and cursor. In the Unix/Linux version of R, use Ctrl-C.

GROUPING EXPRESSIONS: *Parentheses* – '`(`' and '`)`', can be used to group expressions. Similar to arithmetic, the parentheses alter the order in which the expressions are evaluated. Notice the differing results of the following mathematical expressions depending on where the parentheses are placed:

```
> 1 - 2 * 3

[1] -5

> (1 - 2) * 3

[1] -3

> 2^4 + 5

[1] 21

> 2^(4 + 5)

[1] 512
```

[1] In future sections, we will see that not all expressions return (print) a result value.

2.4 Recall/correction of previous expressions

Using the command line in R can involve a fair amount of typing. However, there are ways to reduce the amount of necessary typing. Specifically, the R console keeps a history of the expressions entered, which is known as the *command history*. Individually, the expressions can be accessed using the up- and down-*arrow keys*. Repeatedly pushing the up arrow will scroll backwards through the command history.

As you can imagine, this can be extremely useful, as we can reuse previous expressions. With the arrow keys we can access the desired previous expression and then edit it as desired using keys like Backspace and Delete.

The following table summarizes some keyboard editing shortcuts:

Key	Action
↑ (up arrow)	Recalls the previously entered expression from the command history; multiple pushes scrolls through the command history
↓ (down arrow)	Scrolls forward in the history list
← (left arrow)	Moves cursor to the left
→ (right arrow)	Moves cursor to the right
Home (Ctrl-a)	Moves cursor to beginning of current line
End (Ctrl-e)	Moves cursor to end of current line
Ctrl-k	'Kills' (i.e., deletes) the current line; must be at the beginning of the current line (use Ctrl-a in conjunction)

2.5 Quitting R

To quit R, type `q()` at the command line prompt. You will then be asked `Save workspace image?`. For now, answer "No" by either clicking the `No` button, or by typing `n` at the prompt. The meaning of this question will be discussed in future sections.

NOTE: When we type `q()` and hit ENTER, we are actually executing the quit *function*. As we will see in future sections, a function *always* needs to be written with parentheses, even if there are no arguments explicitly specified between them.[2]

[2]Typing the name of a function without the parentheses tells R to print the definition of the function rather than to execute it. For illustration, see what happens when you type `q` at the command line prompt instead of `q()`.

Chapter 3

More than a calculator: The grammar of expressions

Up to this point, all of the expressions we have entered at the command line have been simple mathematical expressions involving only numbers. Also, up to this point, the result of each evaluated expression has been printed and then discarded.

Obviously, we need to be able to use R as more than just a calculator. Specifically, we need to be able to (among other things): (1) retain the results of specific evaluated expressions; (2) use data that consists of more than one number; and (3) possess tools that carry out desired tasks.

The solution to these desires will involve using expressions that include (1) *assignment*, (2) *data structures*, and/or (3) *functions*. Expressions can also involve *operators* and special *flow control statements*.

TERMINOLOGY: We will use the term *object* quite often in future sections. You can think of an object as anything in R that is returned (printed) by an evaluated expression, anything that you define via assignment, or anything that is already defined by R (i.e., functions).

3.1 Assignment

R allows values to be assigned to (symbolic) *variables*; in other words, a value can be associated with a name and that name can be used to represent that value in subsequent expressions.

The simplest example would be to assign the value of 5 to the variable 'x' using the *assignment operator* (`<-`) :[1]

```
> x <- 5
```

The expression `x <- 5` can be read as "the variable `x` is assigned the value `5`." The assignment operator `<-` consists of a less than sign (`<`) followed by a minus sign (`-`) *without any spaces between them*. The less than sign 'points' to the variable receiving the value.

Assignments can also be made in the other direction, using the obvious change in the assignment operator (e.g., `5 -> x`), or the `assign()` function (e.g., `assign("x", 5)`). The usual operator, `<-`, can be thought of as a syntactic short-cut to this.

In general, no result is printed when a variable is assigned to a value. In order to print the value of the evaluated expression, we must enter the name of the variable at the command line:

[1] See the 'Operators' section for more general information.

```
> x

[1] 5
```

As mentioned, once a value has been assigned to a name, that name can be used to represent the value in subsequent expressions:

```
> x + 3

[1] 8

> 10 * x + 2

[1] 52
```

An object can also be '*re-assigned*' at any time, and the old value is overwritten with the new.

```
> x

[1] 5

> x <- 2
> x

[1] 2
```

Spacing around operators is generally disregarded by R, but notice that adding a space in the middle of a `<-` changes the meaning to 'less than' followed by 'minus' (conversely, omitting the space when comparing a variable to a negative number has unexpected consequences).

```
> x < -5

[1] FALSE
```

IMPORTANT: The value we assign to a variable is not limited to a single value. In general, the evaluated value of any expression can be assigned to a variable. For example, in future sections, we will assign a name to our read-in data set, which will allow us to refer to our data set by name in all subsequent expressions. In the same sense, any expression is also allowed to be on the target side (left side) of an assignment, not just a (symbolic) variable name. For example, we can replace any missing values of a *vector* (a type of data structure) with zeros using the following code: `x[!is.na(x)] <- 0`. The details of this expression will make more sense after reading future sections.

NAMING OBJECTS: There are some limitations as to what can be used as a variable name. Specifically, variable names (1) may consist of letters (`A-Z` and `a-z`), digits (`0-9`), dots ('`.`'), and the underscore ('`_`'); and (2) must not start with a digit nor underscore, nor with a period followed by a digit. Mathematical operators, such as `+`, `-`, $*$, and `/`, in addition to other special characters are also not allowed to be used in object names. And *DON'T FORGET*, R is case sensitive – so, `x` and `X` can name two distinct objects.

It is important to know whether a variable name is already being used before assigning it to a new value. After reading the 'Functions' section, see the 'Object Management' section and its discussion of the `exists()` function.

3.2 Data types and data structures

As we have seen, R recognizes a number when we type one. This also includes negative numbers (e.g., `-2`). To specify a piece of text (also called a *character string*), type it within either single or double quotes, such as `"cat"` or `"Drug X"`. R also recognizes *logical values* (also called boolean values), which are typed as `TRUE` and `FALSE`. And there is a special value, `NA`, which represents a missing or unknown value. There are also three other special constants: (1) `NULL`, which is used to indicate an empty object; (2) `Inf`, which denotes infinity; and (3) `NaN` ('Not-a-Number'), which is produced by numerical computations whose result is undefined (e.g., `1/0`, `0/0`, or `Inf - Inf`).

In addition to these basic types of data, R provides a number of *data structures* that allow multiple values to be specified as a single object. Specifically, the data structures consist of *vectors*, *matrices*, *arrays*, *data frames*, and *lists*.

The *vector* is the simplest data structure in R. For example, a single value in R (i.e., the logical value `TRUE` or the numeric value `2`) is actually just a vector of length 1. Vectors are one dimensional, meaning they have only a length attribute, and consist of an ordered collection of elements. All elements of a vector must be the same data type – i.e., all numeric, all character (text strings), or all logical – but can also include missing elements designated with the `NA` value.

In addition to vectors of numbers, character strings, and logical values, R also recognizes vectors of *categorical* values, where the elements of the vector may only take one of a finite set of values, such as `"male"` and `"female"`. Such a vector is called a *factor* and the set of possible values are the *levels* of the factor. Factors are most often automatically generated when data is imported into R, but a factor may also be generated explicitly.

A *matrix* is a two-dimensional data structure that consists of rows and columns – think of a vector with dimensions. Like vectors, all the elements of a matrix must be the same data type, and can include missing elements designated with the `NA` value. An *array* is a generalization of a matrix to allow more than two dimensions. In general, an array is k-dimensional.

A *data frame* in R corresponds to what other statistical packages call a 'data matrix' or a 'data set' – the 2-dimensional data structure used to store a complete set of data, which consists of a set of variables (columns) observed on a number of cases (rows). In other words, a data frame is a generalization of a matrix. Specifically, the different columns of a data frame may be of different data types, but all the elements of any one column must be of the same data type. As with vectors, the elements of any one column can also include missing elements designated with the `NA` value. Most types of data you will want to read into R and analyze are best described by data frames.

Lastly, a *list* has the ability to combine a collection of objects into a larger composite object. Formally, a list in R is a data structure consisting of an ordered collection of objects known as its *components*. Each component can contain any type of R object and the components need not be of the same type. In fact, a data frame can be thought of as a list of vectors and/or factors of the same length, which are related 'across,' such that data in the same position come from the same experimental unit (subject, animal, etc.). As a more general example, a list might contain vectors of several different data types and lengths, matrices or more general arrays, data frames, functions, and/or other lists. Because of this, a list provides a convenient way to return the results of a statistical computation – in fact, the results that many R functions return are structured as a list.

3.3 Functions

Almost everything in R is done by invoking *functions*. Functions in R can do three things: (1) have values passed to them; (2) return a value; and/or (3) generate *side effects*, which are anything that is not the

returning of a value. Examples of functions that generate side effects are the printing and plotting functions.

Every function in R, whether intrinsic to the language or user-written, is defined using the same basic statement: `FUNname <- function( arglist ) { body }`, where `FUNname` is the name of the function; *`arglist`* is a *comma* separated list of zero or more arguments that can be passed to the function; and *`body`* contains the statements that perform the actions of the function. In turn, the format to invoke a function is to type its name followed by a set of parentheses containing zero or more arguments. In other words, we can think of calculus with its mathematical functions like `f(x)` or `g(x, y)`.

As John Verzani put it in his book, 'functions are like pets' – they don't come (aren't invoked) unless we call them by name (case-sensitive and spelled properly); they have a mouth (the parentheses) that like to be fed (the arguments to the function), and they will complain if they are not fed properly (the output of warnings and errors).

BE AWARE:

1. Because R is case sensitive, so are the names of functions. So, the `reshape` and `reShape` functions are two distinct functions.

2. In R, in order to be executed, a function *always* needs to be invoked with parentheses, even if there are no arguments explicitly specified between them. As with any other assigned object, typing the name of an object at the command line and pressing ENTER causes the assigned value of the object to be printed. In the case of a function, typing only the name of the function (and not including the parentheses) prints the body of the function. For example, compare the following:

```
> date

function ()
.Internal(date())
<environment: namespace:base>

> date()

[1] "Wed Oct  4 09:09:41 2006"
```

You will notice in this document, and all to come, that we typeset all of the function names with the parentheses in order to remind us of this.

NESTED FUNCTIONS: As mentioned, some functions, when invoked, return an evaluated value. This means that the expression invoking a function can be passed as an argument to yet another function invocation. This is the concept of *nested functions*. An example would be to pass the result of an invocation of the `seq()` (sequence generation) function, which is an integer vector of at least one element, to the main argument of the `rep()` (replication) function, which has to be a vector of any type and any length. Using nested functions also saves you from having to assign each interim step to a variable name. We will use nested functions often throughout this series of documents.

3.3.1 Specifying arguments

A function's arguments are how values are passed to the function when it is invoked. The arguments of a function can be an `ARGname` or an `ARGname = VALUE` construct. The argument list can also contain a special type of argument: `...`.

An `ARGname` argument is often the first argument in a function's argument list and often represents the main data object being passed to the function. For example:

```
> ourFUN <- function(x) {
+     x + 5
+ }
> ourFUN(5)

[1] 10
```

The `ARGname = VALUE` construct is used to specify a default value for an argument. If you do not specify a value for that argument when the function is invoked, the default value will be used and evaluated in the body of the function, as if you passed `ARGname = VALUE` in the function invocation. However, if you do specify a value for that argument, the specified value will be used instead of the default value. For example,

```
> anotherFUN <- function(x, y = 5) {
+     x + y
+ }
> anotherFUN(5)

[1] 10

> anotherFUN(5, y = 7)

[1] 12
```

The `...` argument can hold a variable number of arguments, and is mostly used for passing arguments to other functions invoked in the body of the outer function. For example,

```
> lastFUN <- function(z, ...) {
+     ourFUN(z) - anotherFUN(z, ...)
+ }
> lastFUN(5)

[1] 0

> lastFUN(5, y = 10)

[1] -5
```

In `lastFUN(5, y = 10)`, the `y = 10` argument is passed to the `anotherFUN()` function in the body of the `lastFUN()` function.

The `args()` function displays the names and corresponding default values of a function (if any). For example,

```
> args(read.table)

function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    row.names, col.names, as.is = FALSE, na.strings = "NA", colClasses = NA,
    nrows = -1, skip = 0, check.names = TRUE, fill = !blank.lines.skip,
    strip.white = FALSE, blank.lines.skip = TRUE, comment.char = "#",
    allowEscapes = FALSE, flush = FALSE)
NULL
```

This is very helpful when you need to verify the argument name and/or default value of an argument before invoking the function, but it does not give you any further details regarding each argument. This information is given in the `Arguments` section of a function's help file – see the 'Finding help' section.

When invoking a function, arguments may be specified by name using their `ARGname` *tag*, or they may be specified by their *position* in the function invocation. An argument's position in the order of the list of arguments is determined by using the commas separating the arguments as placeholders. Because there is

a limited number of arguments for a function, `ARGname` tags may be abbreviated. Named arguments may also be specified in any order. And recall, if you are not changing the default value, arguments with default values can be omitted from the invocation.

Since not all arguments need to be given when a function is invoked, there needs to be a mechanism to match the function arguments to those arguments given in the function invocation. The general rule for matching the given argument to the function arguments is: (1) `ARGname` tags that match exactly are paired up; (2) `ARGname` tags that partially match are paired; and (3) remaining arguments in the function invocation are matched to the remaining arguments in the function *in order*. If the function's argument list contains `...`, the arguments before the `...` are matched as usual (by name matching and/or position), arguments that appear after the `...` are matched only by full name, and all unmatched arguments are passed to `...`. If any arguments remain unmatched, an error is declared.

Obviously, specifying arguments by their position can be dangerous, especially if you reference the order of the arguments incorrectly. Therefore, it is safest to *always* specify all arguments using their `ARGname` tag. You can also safely specify the first argument only by its position, but specify all other arguments by their name. Specifying arguments by their name also makes your code more easily readable – both for yourself and others.

3.3.2 Packages

R functions are organized into *packages*, which has two benefits. First, this makes it possible to load only the packages containing the functions that are required, so that R runs faster and uses less memory. Second, it is very easy to make use of functions that other people have written and put into a package.

Currently, over 400 of these 'contributed' packages are available on the CRAN website via the Packages link under the Software section. The packages are listed in alphabetical order, but there is also a `CRANTaskViews` link, which allows you to browse through some of the packages by topic and provide tools to automatically install all packages for special areas of interest. As you can imagine, the packages cover a wide range of applications including Bayesian inference, cluster analysis and finite mixture models, statistical genetics, multivariate statistics, analysis of spatial data, and graphics. The site for each package contains a brief description, the source files, an index of contents, and a downloadable reference manual.

More information regarding packages can be found in several documents. The 'Installation and Administration' manual is the main resource. In addition, 'The R FAQ' devotes a section to package management, while the complementary 'R for Windows FAQ' explains a couple of Windows specific issues related to package management. The 'Writing R Extensions' manual is also available for those wanting to develop their own package. See the 'Existing R documentation' section for more details.

The `install.packages()` function, as the name suggests, is used to download and install specific packages from the command line. Once a package is installed, it can be *loaded*, so that the functions in the package can be used, using the `library()` function – `library(`*`pkg`*`)`, where *`pkg`* is the installed package. You need to install a package only once, but the package must be loaded during every R session you wish to use it in. The `library()` function can also be used to list all the available packages for installation when specified without any arguments (i.e., `library()`).

Regardless of whether the package has been loaded (but it has to be installed), you can use `library(help = `*`pkg`*`)` or `help(package = `*`pkg`*`)` to list the functions it contains. You can also read a brief description of the package by typing `packageDescription("`*`pkg`*`")`.

The following packages are automatically installed when you install R:

Package	Description
base	Base R functions
datasets	Base R datasets
grDevices	Graphics devices for base and grid graphics
graphics	R functions for base graphics
grid	A rewrite of the graphics layout capabilities, plus some support for interaction
methods	Formally defined methods and classes for R objects, plus other programming tools
splines	Regression spline functions and classes
stats	R statistical functions
stats4	More statistical functions
tcltk	Interface and language bindings to Tcl/Tk GUI elements
tools	Tools for package development and administration
utils	R utility functions

A subset of these packages are also automatically loaded when you start an R session. The `search()` function can be used to list all loaded packages (whether automatically or explicitly loaded with `library()` function). Each package name is preceded with '`package:`':

```
> search()

 [1] ".GlobalEnv"        "package:foreign"   "package:chron"     "package:Hmisc"
 [5] "package:tools"     "package:methods"   "package:stats"     "package:graphics"
 [9] "package:grDevices" "package:utils"     "package:datasets"  "Autoloads"
[13] "package:base"
```

In addition to the packages automatically installed and/or loaded by default, I personally recommend installing *and loading* the following add-on packages: `Hmisc`, `Design`, `chron`, and `foreign`. The `chron` package is very useful when manipulating date variables, and the `foreign` package contains functions to read in data files from various sources (e.g., SAS, SPSS, and STATA). The `Hmisc` package (i.e. 'Harrell Miscellaneous'), which was developed by Dr. Frank E. Harrell, contains many functions useful for analyzing data, producing high-level graphics, performing utility operations, computing sample size and power, importing data sets, imputing missing values, making advanced tables, determining variable clusters, manipulating characters strings, converting R objects to LaTeXcode, and recoding variables.The `Design` library, which was also developed by Dr. Frank E. Harrell, is a collection of functions that assist and streamline regression modeling, testing, estimation, validation, graphics, prediction, and typesetting. More information about these last two packages are available from two sources: Frank Harrell's book 'Regression Modeling Strategies', and the pdf 'An Introduction to S and the Hmisc and Design Libraries'.

It is recommended that you either reinstall your packages or *update* the existing packages each time the newest version of R is released, or when a newer version of the package(s) is/are released. Luckily, even if you are unaware of any updates in the package versions, the `update.packages()` function can be used to update a single package (`update.packages("pkg")`), or to update all of your installed packages (`update.packages()` – no arguments specified). If you are using the Windows version of R, you will find a menu called Packages that provides an interface to the `install.packages()`, `update.packages()` and `library()` functions .

You can *unload* a specific loaded package using the `detach()` function – `detach("package:pkg)`. And installed packages can be removed (i.e., *uninstalled*) using the `remove.packages()` function – `remove.packages(c("pkg1", "pkg2"))`, where *pkg1* and *pkg2* are the package you want to unload.

And lastly, the `remove.packages()` function can be used to uninstall a package.

3.3.3 Functions that construct data structures

NOTE: Assignment can be appropriately incorporated into all the following functions invocations. That is, any of the constructed data structures can be assigned a name, and/or previously assigned data structures

may be specified as the data arguments to construct subsequent data structures.

3.3.3.1 Vectors and factors

There are a number of functions (and an operator) that can be used to easily construct *vectors* of any length, including the `c()` function, the `rep()` function, the `sample()` function, the `seq()` function (and `:` operator), and the `paste()` function.

The `c()` (concatenate) function constructs a vector by joining the supplied elements end-to-end. The elements supplied can be single numeric, character, logical, and/or missing values. The elements can also be vectors themselves. The only stipulation is that all the supplied elements must be of the same data type. To use the `c()` function we merely separate the elements by commas:

```
> c(2, 3, 5, 2, 7, 1)

[1] 2 3 5 2 7 1

> c(TRUE, FALSE, FALSE, TRUE)

[1]  TRUE FALSE FALSE  TRUE

> c("cat", "dog", "bird", "horse")

[1] "cat"   "dog"   "bird"  "horse"

> x <- c(2, NA, 5, NA, NA, 7)
> y <- c(10, 15, 12)
> c(y, x)

[1] 10 15 12  2 NA  5 NA NA  7
```

If different types of elements are mixed, all will be coerced into a common type, which is usually character. For example,

```
> c(1:3, "cat")

[1] "1"   "2"   "3"   "cat"
```

When constructing a vector of character strings, an alternative to the `c()` function is the `Hmisc` package's `Cs()` function. The supplied elements to the `Cs()` function are *unquoted* character strings *that do not contain any spaces*. For example,

```
> Cs(dog, cat, horse)

[1] "dog"   "cat"   "horse"
```

DON'T FORGET to load the `Hmisc` package with the `library()` function (i.e., `library(Hmisc)`) if you haven't already done so. If you haven't previously installed the package, you must do this first.

The `rep()` (replicate) function constructs a vector of repeated values.

```
> args(rep)

function (x, times, ...)
NULL
```

The elements to repeat are supplied as a vector of any data type, and can include missing values (`x=`). In addition to the main data argument (`x=`), the `rep()` (replicate) function has three other formal arguments: (1) `times=`, which specifies the number of times to repeat the supplied vector (`x=`) if given as an integer, or each element of the supplied vector (`x=`) if given as a vector of the same length as `x=`; (2) `length.out=`, which specfies (as an integer) the desired length of the constructed vector; and (3) `each=`, which specifies (as an integer) the number of times each element of the supplied vector (`x=`) should be repeated. Obviously, these three arguments can be used in various combinations. For example,

```
> rep(c(1, 2, 3, 4), times = 2)

[1] 1 2 3 4 1 2 3 4

> rep(c(1, 2, 3, 4), times = c(2, 4, 3, 1))

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

> rep(c("M", "F"), length.out = 5)

[1] "M" "F" "M" "F" "M"

> rep(c(TRUE, FALSE), each = 2, length.out = 10)

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

The `sample()` function constructs a vector by taking a random sample (with or without replacement) of the specified size from the elements that are supplied.

```
> args(sample)

function (x, size, replace = FALSE, prob = NULL)
NULL
```

The elements to sample from (`x=`) are supplied as either a vector of more than one element (all elements of the same data type, and can include missing values), or as a positive integer. If a single positive integer is supplied as the data argument, the sample is drawn from the integer values from 1 to the supplied integer. The `size=` argument specifies (as a non-negative integer) the number of elements to choose. By default `size = length(x)`, which means a sample of the same length as the elements being sampled is constructed (i.e., a random permutation of the supplied elements). The `replace=` argument specifies whether (`replace = TRUE`) or not (`replace = FALSE`, default) the sampling should be done with replacement. If `replace=FALSE`, the length of the elements being sampled must be greater than or equal to the value specified by `size=`. If the length of the elements being sampled is shorter, `replace=` must be set equal to `TRUE` or else you'll get an error (`Can't take a sample larger than the population`). Lastly, the `prob=` argument specifies a vector of probability weights for obtaining the elements of the vector being sampled. By default, the elements of the vector being sampled are sampled with equal probability (i.e., by default, `prob = NULL`). The probabilities need not sum to 1, but they should be non-negative and not all zero. If `replace = FALSE` these probabilities are applied sequentially, that is the probability of choosing the next item is proportional to the probabilities amongst the remaining items. The number of nonzero weights must be at least the value specified by `size=`.

```
> sample(10)

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

> sample(c(TRUE, FALSE), size = 5, replace = TRUE)

[1]  TRUE  TRUE FALSE  TRUE  TRUE

> sample(c("A", "B", "C"), size = 10, replace = TRUE, prob = c(0.75, 0.5,
+     0.25))
```

```
 [1] "A" "B" "A" "A" "C" "C" "B" "B" "B" "C"
```

BE AWARE: Because the `sample()` function takes a *random* sample, a different sample will be returned every time you invoke the sample function. For example,

```
> sample(5)

[1] 3 1 2 4 5

> sample(5)

[1] 2 4 5 1 3
```

The `seq()` (sequence) function is a general tool for generating equidistant series of numbers. The `seq()` function has five formal arguments, but only some of them may be specified in any one function invocation. This causes the `seq()` function to be invoked in one of four ways:

1. `seq(value)`, where `value` is an integer. If `value` is a positive integer, `seq(value)` generates the sequence `1`, `2`, `...`, `value`. If `value` is a negative integer, `seq(value)` generates the sequence `1`, `0`, `...`, `-value`.

   ```
   > seq(5)

   [1] 1 2 3 4 5

   > seq(-5)

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

2. `seq(from, to)`, where `from` and `to` specify the beginning and end of the sequence, respectively. This form generates the sequence `from`, `from+1`, `(from+1)+1`, `...`, `to` if `to` > `from`, or the sequence `from`, `from-1`, `(from-1)-1`, `...`, `to` if `to` < `from`. The same sequences can be generated using the colon (`:`) operator. In the following examples, the first two invocations and the second two invocations are equivalent.

   ```
   > seq(from = 2, to = 10)

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

   > 2:10

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

   > seq(from = 10, to = 3)

   [1] 10  9  8  7  6  5  4  3

   > 10:3

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

   `from` and `to` need not be integers. In this case, the sequence increments by 1 up to the sequence value less than or equal to `to` if `to` > `from`, or the sequence decrements by 1 down to the sequence value greater than or equal to `from` if `to` < `from`.

   ```
   > 5.7:20.2

    [1]  5.7  6.7  7.7  8.7  9.7 10.7 11.7 12.7 13.7 14.7 15.7 16.7 17.7 18.7 19.7

   > 10.5:3.25
   ```

```
[1] 10.5  9.5  8.5  7.5  6.5  5.5  4.5  3.5
```

3. `seq(from, to, by)`, where `from` and `to` are the same as before, and `by` specifies the increment of the sequence (default, `by = 1`). This form generates the sequence `from`, `from+by`, `(from+by)+by`, ..., up to the sequence value less than or equal to `to` if `to` > `from` and `by` is positive, or the sequence `from`, `from-by`, `(from-by)-by`, ..., down to the sequence value greater than or equal to `from` if `to` < `from` and `by` is negative. For example,

```
> seq(from = -1, to = 1, by = 0.2)

 [1] -1.0 -0.8 -0.6 -0.4 -0.2  0.0  0.2  0.4  0.6  0.8  1.0

> seq(from = 9, to = 1, by = -2)

[1] 9 7 5 3 1

> seq(from = 5.7, to = 20.2, by = 0.4)

 [1]  5.7  6.1  6.5  6.9  7.3  7.7  8.1  8.5  8.9  9.3  9.7 10.1 10.5 10.9 11.3 11.7 12.1
[18] 12.5 12.9 13.3 13.7 14.1 14.5 14.9 15.3 15.7 16.1 16.5 16.9 17.3 17.7 18.1 18.5 18.9
[35] 19.3 19.7 20.1
```

4. `seq(from, to, length)`, where `from` and `to` are the same as before, and `length` specifies the desired length of the sequence. This form generates the sequence of `length` equally spaced values from `from` to `to`. As we saw in the second form `seq(from, to)`, if `length` is not specified, the default `by = 1` is used. For example,

```
> seq(from = 2, to = 15, length = 7)

[1]  2.000000  4.166667  6.333333  8.500000 10.666667 12.833333 15.000000
```

Lastly, the `paste()` function can be used to construct a *character* vector whose elements represent the interaction of the supplied arguments.[2] For example,

```
> paste(c("Treatment"), c("A", "B", "C"))

[1] "Treatment A" "Treatment B" "Treatment C"
```

As seen, by default, the arguments are separated in the results by a single blank character (i.e., `sep = " "`), but it can be specified otherwise. Like the `c()` (concatenate) function, any numbers given among the arguments are coerced into character strings. For example,

```
> paste(c("X", "Y"), 1:10, sep = ".")

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

As mentioned, factors are most often automatically generated when data is imported into R, but a factor may also be generated explicitly with the `factor()` function.

```
> args(factor)

function (x = character(), levels = sort(unique.default(x), na.last = TRUE),
    labels = levels, exclude = NA, ordered = is.ordered(x))
NULL
```

The elements of the supplied vector that will be encoded as the factor (`x=`) can be numeric, character, or logical. For example,

[2] Actually, the codepaste() function incorporates the '*recycling rule*' – see the 'Vectorization of functions' section for more details.

```
> factor(sample(3, size = 10, replace = TRUE))

 [1] 2 2 2 3 3 3 2 1 1 1
Levels: 1 2 3

> factor(sample(c("female", "male"), size = 10, replace = TRUE))

 [1] male   male   female male   male   male   female female male   female
Levels: female male

> factor(sample(c(TRUE, FALSE), size = 10, replace = TRUE))

 [1] TRUE  FALSE FALSE TRUE  TRUE  FALSE TRUE  TRUE  FALSE FALSE
Levels: FALSE TRUE
```

As seen, by default, the sorted unique values of the vector are used to define the *levels* of the factor. The levels are sorted based on the data type of the supplied vector (i.e., either alphabetical or numerical order). Also, by default, missing values (`NA`s) are not defined as a level (`exclude = NA`).

The `levels=` argument can be used to explicitly define and/or order the levels of a factor. As examples, we construct a factor by default, we construct a factor by specify the desired order of the levels, and we construct a factor with an additional level that is not present in the data:

```
> race <- sample(c("B", "W", "O"), size = 10, replace = TRUE)
> race

 [1] "O" "W" "O" "B" "W" "W" "O" "O" "B" "W"

> factor(race)

 [1] O W O B W W O O B W
Levels: B O W

> factor(race, levels = c("W", "B", "O"))

 [1] O W O B W W O O B W
Levels: W B O

> factor(race, levels = c("W", "B", "O", "H"))

 [1] O W O B W W O O B W
Levels: W B O H
```

When using either the `levels=` argument, take care to spell the levels correctly, and ***REMEMBER*** R is case sensitive. The elements of a factor not matching any element of the `levels=` argument will be coded as `NA` (missing). For example,

```
> factor(race, levels = c("w", "B", "O"))

 [1] O    <NA> O    B    <NA> <NA> O    O    B    <NA>
Levels: w B O
```

Once a factor has been created, the `levels()` function can be used to return (print), reorder, and/or redefine the levels. To redefine the levels of a factor using the `levels()` function, we can specify a vector of character strings with a length of at least the number of levels of the created factor, or we can specify a *named* list specifying how to redefine the levels (i.e., `list(NEWlevel = "OLDlevel")`). The list can also specify new levels. For example,

```
> test <- factor(sample(c("positive", "negative"), size = 10, replace = TRUE))
> levels(test)

[1] "negative" "positive"

> levels(test) <- c("positive", "negative")
> test

 [1] negative negative positive negative negative positive positive positive negative
[10] negative
Levels: positive negative

> levels(test) <- list(Positive = "positive", Negative = "negative")
> test

 [1] Negative Negative Positive Negative Negative Positive Positive Positive Negative
[10] Negative
Levels: Positive Negative

> levels(test) <- list(Undetermined = "Undetermined", Positive = "Positive",
+     Negative = "Negative")
> test

 [1] Negative Negative Positive Negative Negative Positive Positive Positive Negative
[10] Negative
Levels: Undetermined Positive Negative
```

The `labels=` argument can be used to specify a vector of *labels* for the levels of the factor. ***IMPORTANT:*** The labels given in the `labels=` argument must be in the same *order* as the factors levels. For example, suppose we want to add labels to the `race` factor levels, which are (in order) `W`, `B`, `O`, and `H`. Therefore, I would need to specify `labels = c("White", "Black", "Other", "Hispanic")` to properly match the order of the levels. If I had specified `labels = c("Black", "Hispanic", "Other", "White")`, none of the level labels would match the corresponding levels. Here's another example:

```
> factor(sample(3, size = 10, replace = TRUE), levels = c(3, 1, 2), labels = c("US",
+     "Canada", "Mexico"))

 [1] US     US     Canada US     Mexico Mexico Mexico US     Canada US
Levels: US Canada Mexico
```

Lastly, specify `ordered = TRUE` to specify that the levels of the factor should be regarded as *ordered*. The levels will be ordered *in the order given*, so use the `levels=` argument in conjunction with the `ordered=` argument to make sure the correct order is specified. For example,

```
> stress <- factor(sample(c("low", "medium", "high"), size = 10, replace = TRUE),
+     ordered = TRUE, levels = c("low", "medium", "high"))
> stress

 [1] medium high   low    high   low    medium high   medium high   high
Levels: low < medium < high
```

As can be seen, < signs are added to the levels to illustrate the ordering, which can be useful. For example,

```
> stress <= "medium"

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

If we had not specified the `levels=` argument but had specified `ordered = TRUE`, the levels would have been ordered in *alphabetical* order (i.e., `high`, `low`, `medium`).

Here's another example that uses the `levels=`, `labels=`, and `ordered=` arguments to create a factor that represents the months of the year. The months are originally coded with numeric values (`1:12`), but character month labels are then added. Specifically, the `levels=` argument specifies the order of the levels for `ordered = TRUE` (from September to August), and then the character labels are added to the ordered levels with the `labels=` argument.

```
> factor(1:12, ordered = TRUE, levels = c(9:12, 1:8), labels = c("Sep",
+     "Oct", "Nov", "Dec", "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
+     "Aug"))

 [1] Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
Levels: Sep < Oct < Nov < Dec < Jan < Feb < Mar < Apr < May < Jun < Jul < Aug
```

The `ordered()` function can also be used to construct an ordered factor. Additional function that can be used to construct a factor include the `interaction()` function, the `cut()` function and the `Hmisc` package's `cut2()` function, the `gl()` (generate levels) function, and the `Hmisc` package's `score.binary()` function. The `relevel()` and `reorder()` functions can also be used to reorder the levels of a factor. And the `nlevels()` function can be used to return (print) the number of levels of a factor.

DON'T FORGET to load the `Hmisc` package with the `library()` function (i.e., `library(Hmisc)`) if you haven't already done so. If you haven't previously installed the package, you must do this first.

See the help files of each of the functions mentioned in this section for more details – see the 'Finding help' section.

3.3.3.2 Other data structures

The `matrix()` function constructs an `nrow` x `ncol` *matrix* from a supplied vector (`data=`).

```
> args(matrix)

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

If no data vector is supplied, a matrix is created, but each element is designated missing with `NA`. For example,

```
> matrix(nrow = 2, ncol = 4)

     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
```

By default, the matrix is filled by columns (`byrow = FALSE`), but specifying `byrow = TRUE` fills the matrix by rows. For example,

```
> matrix(1:6, ncol = 3)

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

> matrix(1:6, ncol = 3, byrow = TRUE)

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

Row and column names can also be specified using the `dimnames=` argument. The `dimnames=` argument is specified as a list of two components giving the row and column names, respectively. For example,

```
> matrix(c(1, 2, 3, 11, 12, 13), nrow = 2, ncol = 3, byrow = TRUE, dimnames = list(c("row1",
+     "row2"), c("C.1", "C.2", "C.3")))

     C.1 C.2 C.3
row1   1   2   3
row2  11  12  13
```

The `cbind()` and `rbind()` functions can also be used to construct a *matrix* by binding together the data arguments horizontally (column-wise) or vertically (row-wise), respectively. The supplied data arguments can be vectors (of any length) and/or matrices with the same columns size (i.e., the same number of rows) or row size (i.e., the same number of columns), respectively. Any supplied vector is cyclically extended to match the 'length' of the other data arguments if necessary. For example,

```
> cbind(1:3, 7:9)

     [,1] [,2]
[1,]    1    7
[2,]    2    8
[3,]    3    9

> rbind(1:11, 5:15)

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

Row and column names are created by supplying the vectors as *names* vectors – i.e., `VECTORname = vector`. For example,

```
> cbind(col1 = 1:3, col2 = 7:9)

     col1 col2
[1,]    1    7
[2,]    2    8
[3,]    3    9

> rbind(row1 = 1:11, row2 = 5:15)

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
row1    1    2    3    4    5    6    7    8    9    10    11
row2    5    6    7    8    9   10   11   12   13    14    15
```

The `array()` function can be used to construct an *array*. With the `array()` function, two formal arguments must be specified: (1) a vector specifying the elements to fill the array; and (2) a vector specifying the dimensions of the array. For example,

```
> array(1:24, dim = c(3, 4, 2))

, , 1

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

```
, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

The array is filled column-wise, and if the data vector is shorter than the number of elements defined by the dimensions vector, the data vector is recycled from its beginning to make up the needed size. Like the `matrix()` function, dimension names can be added using the `dimnames=` argument.

The `data.frame()` function can be used to construct a *data frame*. The data arguments, which will construct the columns of the data frame, can be specified with or without a corresponding column name – either in the form `value` or the form `COLname = value`. For example,

```
> ourdf <- data.frame(id = 101:110, sex = sample(c("M", "F"), size = 10,
+     replace = TRUE), age = sample(20:50, size = 10, replace = TRUE),
+     tx = sample(c("Drug", "Placebo"), size = 10, replace = TRUE), diabetes = sample(c(TRUE,
+         FALSE)))
> ourdf

    id sex age      tx diabetes
1  101   F  32    Drug     TRUE
2  102   M  30 Placebo    FALSE
3  103   F  37 Placebo     TRUE
4  104   M  44 Placebo    FALSE
5  105   M  22 Placebo     TRUE
6  106   F  48    Drug    FALSE
7  107   M  37    Drug     TRUE
8  108   F  50    Drug    FALSE
9  109   F  46    Drug     TRUE
10 110   F  21 Placebo    FALSE
```

Character vectors are automatically coerced into factors, whose levels are the unique values appearing in the vector. In the next lecture we will also cover how to construct a data frame by reading in a data file.

Lastly, the `list()` function can be used to construct a *list*. Like the `data.frame()` function, the components of a list can be named using the `COMPONENTname = component` construct. For example,

```
> list(logical.vector = c(TRUE, FALSE), numeric.vector = 1:4, matrix.2x5 = matrix(12:22,
+     nrow = 2, byrow = TRUE))

$logical.vector
[1]  TRUE FALSE

$numeric.vector
[1] 1 2 3 4

$matrix.2x5
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   12   13   14   15   16   17
[2,]   18   19   20   21   22   12
```

See the help files of each of the functions mentioned in this section for more details – see the 'Finding help' section.

3.3.4 Vectorization of functions & the recycling rule

Many R functions are *vectorized*, meaning that they operate on each of the elements of a vector (i.e., *element-by-element*). For example, the expression `log(y)` returns a vector that is the same length of `y`, where each element of the result is the natural logarithm of the corresponding element in `y`.

```
> y <- sample(1:100, size = 10)
> y

 [1]  1  9  4 83 65  7 71 74 30 82

> log(y)

 [1] 0.000000 2.197225 1.386294 4.418841 4.174387 1.945910 4.262680 4.304065 3.401197
[10] 4.406719
```

Other examples of vectorized functions include the `exp()` (exponentiation) function, the `sqrt()` (square root) function, the `round()` function, and the `casefold()` function.

In addition, any of the Arithmetic, Comparison, and Logic operators mentioned in the 'Operators' section will operate element-by-element. For example, adding two vectors of the same length:

```
> x <- sample(1:20, size = 10)
> x

 [1] 19 14 12  6 15  9  1  3  7 11

> x + y

 [1] 20 23 16 89 80 16 72 77 37 93
```

These operators and other specific functions incorporate what is known as the '*recycling rule*'. The rule is that shorter vectors in the expression are replicated (i.e., recycled; re-used) to be the length of the longer vector. The evaluated value of the expression is a vector of the same length as the longest vector which occurs in the expression.

The simplest illustration of the recycling rule is when the expression involves a 'constant' (e.g., a single numeric value):

```
> y + 2

 [1]  3 11  6 85 67  9 73 76 32 84

> y < 50

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

In each of these examples, the constant, which is technically a vector of length 1, is replicated to the length of `y`.

When the length of the shorter vector is greater than 1, the elements of the shorter vector are replicated *in order* until the result is the proper length. For example,

```
> x <- sample(1:20, size = 2)
> x

[1] 13 16

> x + y
```

```
 [1] 14 25 17 99 78 23 84 90 43 98
```

Here, the vector `x` was replicated 5 times to match the length of `y` (10 elements).

If the length of the longer vector is not a multiple of the shorter one, the shorter vector is *fractionally* replicated and a warning is given (`longer object length is not a multiple of shorter object length`). For example (warning is given, but not shown in output),

```
> x <- 1:9
> y <- 1:10
> x + y

 [1]  2  4  6  8 10 12 14 16 18 11
```

Here, to match the length of `y` only the first element of `x` (1) was re-used.

3.3.5 Object-oriented programming

Most objects in R have a *class*, which can be thought of as a special attribute. This includes those objects returned from an expression. In reality, some objects do not have a defined class, while others have more than one class. In general, the class of an object is a character vector of at least one element, which can be determined using the `class()` function. For example,

```
> class(1:10)

[1] "integer"

> class(TRUE)

[1] "logical"

> class(mean)

[1] "function"
```

In R, the behavior of some functions depends on the class of the object specified as its first argument. These are known as *generic* functions. For a generic function, there can be a number of different (*class specific*) *methods* where each method is a function that corresponds to the action to be taken for a particular class of objects.

These concepts of *class* and *methods* are central to any *object-oriented programming language*. Object-orientation simplifies the programming by accommodating the fact that you will have conceptually similar methods for different types of data, even though the implementations will have to be different. For example, it generally makes sense to print many kinds of data objects, but the print layout will depend on what the data object is. With object-orientation, if you want to print an object, you don't need to find out what type of object it is, then try to remember the proper function to use on that type of object, and then do it. You merely use the generic `print()` function and the right thing happens. In other words, the end result for the user is fewer function names to remember.

When a generic function is called, R uses a process called *method dispatch* to determine which method to use. Specifically, each element of the class of the object is examined in turn until one matches a method. If no element of the class matches a method or an object has no class, then the *default method* is used.

The `methods()` function can be used to print all the methods of a specific function.[3] For example,

```
> methods(mean)
```

[3] If a function is not generic, the message `no methods are found` is returned (e.g., `methods(paste)`).

```
[1] mean.data.frame mean.Date       mean.default    mean.difftime   mean.POSIXct
[6] mean.POSIXlt    mean.times*

   Non-visible functions are asterisked
```

In general, the name of a method is the name of the generic function followed by a period followed by the name of the class. So, the mean method for data frames (class `data.frame`) is `mean.data.frame` and the default method is `mean.default`. If you look at the body of a function, it is also apparent that the function is a generic function if you see the `UseMethod()` function. Take for example the `print()` function:

```
> print

function (x, ...)
UseMethod("print")
<environment: namespace:base>
```

The `UseMethod()` function notes the class of the object, and then calls the relevant methods for that class of object. The `summary()` and `plot()` functions are also well known generic functions.

If you need to consult a function's help file, understanding the concept of method dispatch is indispensable. For instance, if you weren't aware that the `plot()` function is a generic function, you would quickly become frustrated trying to use the plot help file (i.e., `help(plot)`) to determine which arguments you can specify when plotting various objects. As we can see, the plot function's arguments are not very specific:

```
> args(plot)

function (x, y, ...)
NULL
```

However, if you determined the class of the object you were trying to plot using the `class()` function and determined its corresponding class specific method using the `methods()` function, then you could consult the class specific method's help file. As an example, suppose I was trying to plot an object of class `stepfun`, an object generated by the step function `stepfun()`, then I would consult the class specific `plot.stepfun()` function's help file, and become aware of the following arguments:

```
> args(plot.stepfun)

function (x, xval, xlim, ylim = range(c(y, Fn.kn)), xlab = "x",
    ylab = "f(x)", main = NULL, add = FALSE, verticals = TRUE,
    do.points = TRUE, pch = par("pch"), col.points = par("col"),
    cex.points = par("cex"), col.hor = par("col"), col.vert = par("col"),
    lty = par("lty"), lwd = par("lwd"), ...)
NULL
```

Even looking at the help file of a function's default method (e.g., `plot.default()`) is more helpful than looking at the help file of the generic function:

```
> args(plot.default)

function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
    log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
    ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
    panel.last = NULL, asp = NA, ...)
NULL
```

3.4 Operators

As we saw in the 'Functions' section, functions are normally invoked by specifying their name and a set of arguments within parentheses – e.g., `seq(from = x, to = y)`. In contrast, *operators* are functions that (normally) accept only two arguments, which are specified on either side – e.g., `x:y` (the sequence operator). The parsed form of an operator is completely equivalent to a function invocation with the operator as the function name and the operands as the function arguments – e.g., `":"(x, y)`.

In addition to the `:` (*sequence*) operator, which is a shortcut for the `seq()` function, we have encountered the *assignment* operators (`<-` and `->`), which are shortcuts for the `assign()` function, and we have encountered the *arithmetic* and *comparison* operators:

Type	Operator	Action perfromed
Arithmetic	`+`	Addition
	`-`	Subtraction, sign
	`*`	Multiplication
	`/`	Division
	`^`	Raise to power
	`%/%`	Integer division
	`%%`	Modulus – remainder from integer division
Comparison	`<`	Less than
	`>`	Greater than
	`==`	Equal to
	`!=`	Not equal to
	`>=`	Greater than or equal to
	`<=`	Less than or equal to

Like other operators, these arithmetic operators are merely shortcuts for function invocations. For instance, `x + y` is a shortcut for `"+"(x, y)`.

It should be noted, in general, if you try to operate on a vector whose type is not appropriate for that kind of operation, R will automatically convert it to another kind, trying to lose the least possible amount of information in the process. For example, let's add a numeric vector to a logical one.

```
> c(TRUE, FALSE) + c(3, 4)

[1] 4 4
```

When logical vectors are involved in arithmetic operations, `TRUE` is converted to a value of 1, and `FALSE` is converted to a value of 0.

It should also be noted that the exponentiation operator `^` and the left assignment operator `<-` group right to left, whereas all other operators group left to right. That is, `2^2^3` is 2^8 not 4^3, whereas `1 - 1 - 1` is -1, not 1.

The `+` and `-` *binary* (i.e., use two arguments on either side) arithmetic operators, but they are also *unary* (i.e., use only one argument on the right) *sign* operators that are used to specify whether a numeric value is positive or negative. For example,

```
> +1

[1] 1

> -1

[1] -1
```

```
> -(1:10)

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

These unary operators are equivalent to `"+"(1)`, `"-"(1)`, and `"-"(c(1:10))`.

There is also a group of *logic* operators:

Operator	Action perfromed
`&`, `&&`	And (unary, binary)
`|`, `||`	Or (unary, binary)
`!`	Not

Like the `+` and `-` arithmetic operators, there are both unary and binary forms of the `&` (and) and `|` (or) logic operators. For the binary form of the operators (`&` and `|`), each element of the two arguments (e.g., `x & y`) is compared and a *vector* is returned. In contrast, `&&` and `||` only return a single `TRUE`/`FALSE` value for the outcome. For example,

```
> x <- 1:10
> x < 7 & x > 2

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

> x < 7 && x > 2

[1] FALSE
```

The unary forms of the logical operators (`&&` and `||`) are useful when specifying the condition in an `if` or `while` statement – see the 'Control flow statements' section.

Other useful operators include the `%in%` and the (`Hmisc` package's) `%nin%` value matching operators. These binary operators return a logical vector indicating whether elements of the vector specified as the left operand match any of the values in the vector specified as the right operand. For example,

```
> x <- sample(c("A", "B", "C", "D"), size = 10, replace = TRUE)
> x

 [1] "C" "D" "B" "D" "A" "C" "C" "C" "D" "D"

> x %in% c("A", "C")

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

> x %nin% c("A", "C")

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

The `%nin%` operator is the 'negative' of the `%in%` operator. ***DON'T FORGET*** to load the `Hmisc` package with the `library()` function (i.e., `library(Hmisc)`) if you haven't already done so. If you haven't previously installed the package, you must do this first.

There are also three operators that manipulate matrices and arrays: (1) `%*%` (matrix multiplication); (2) `%o%` (outer product of arrays); and (3) `%x%` (Kronecker product on arrays).

The arithmetic, comparison, logical (except the unary `&` and `|`), value matching, and matrix/array operators are all *vectorized*, as described in the 'Vectorization of functions' section.

In addition to all of the mentioned operators, the `~` (*tilde*) operator is used to specify model formulae (e.g., `y ~ x + z`), the `?` (*help*) operator is a shortcut of the `help()` function (see the 'Finding help' section), and there are two *indexing* operators, `[ ]` and `$`, which can be used to extract various elements of a data structure. These indexing operators will be discussed in detail in the next section.

All of these operators have an *order of precedence* (highest first):

```
$
^
- + (unary)
:
%/% %% %in% %nin% %*% %o% %x%
* /
- + (binary)
> >= < <= == !=
!
& &&
| ||
~
-> <-
```

Like all functions, each mentioned operator has a help file that you can access for more information – see the 'Finding help' section. Because all operators are nonstandard topic names, they need to be quoted – e.g., `help("+")` or `help("%in%")`.

3.4.1 Extracting elements of a data structure

As mentioned in the previous section, the indexing operators `[]` and `$` can be used to extract various elements from a data structure.

3.4.1.1 Vectors and factors

For a vector, we can use the `[ ]` indexing operator to extract specific subsets of the vector's elements. Specifically, for a vector `x`, we use the general form `x[index]`, where `index` is a vector that can be one of the following forms:

1. **A vector of positive integers.** The values in the `index` vector normally lie in the set `{1, 2, ..., length(x)}`. The corresponding elements of the vector are extracted, in that order, to form the result. The `index` vector can be of any length and the result is of the same length as the `index` vector. For example, `x[6]` is the sixth element of `x`, `x[1:10]` extracts the first ten elements of `x` (assuming `length(x)` $\geq$ 10), `x[length(x)]` extracts the last element of `x`, and `x[c(1, 5, 20)]` extracts the 1st, 5th, and 20th elements of x (assuming `length(x)` $\geq$ 20). We can use any of the functions that construct numeric vectors discussed in the 'Functions that construct data structures' section to define the `index` vector of positive integers – the `c()` (concatenate), the `rep()` (replicate), the `sample()`, and the `seq()` (sequence) functions or the `:` (colon) operator. `NA` is returned if the `index` vector contains an integer $>$ `x`, and an empty vector (`numeric(0)`) is returned if the `index` vector contains a `0`.

2. **A vector of negative integers.** This specifies the values to be *excluded* rather than *included* in the extraction. For example, `x[-(1:5)]` extracts all but the first five elements of `x` – we merely place a negative sign (the `-` unary operator) in front of the `index` vector. As seen, all elements of `x` except those that are specified in the `index` vector are extracted, in their original order, to form the result. The results is the `length(x)` minus the length of the `index` vector elements long.

3. **A logical vector.** The `index` vector must be of the same length as `x`. Values corresponding to `TRUE` in the `index` vector are extracted and those corresponding to `FALSE` or `NA` are not.[4] The logical `index`

[4]See the 'More on missing values' section for more details.

vector can be explicitly given (e.g., `x[c(TRUE, FALSE, TRUE)]`) or can be constructed as a *conditional* expression using any of the comparison, logic, and/or value matching operators. For example,

```
> x <- sample(10, size = 10, replace = TRUE)
> x

 [1] 2 8 6 6 1 6 6 3 6 2

> x[x == 4]

numeric(0)

> x[x > 2 & x < 5]

[1] 3

> x <- factor(sample(c("B", "W", "O"), size = 10, replace = TRUE))
> x

 [1] W O O O W O W B O W
Levels: B O W

> x[x == "B"]

[1] B
Levels: B O W

> x[x %in% c("B", "O")]

[1] O O O O B O
Levels: B O W
```

As seen, the result is the same length as the number of `TRUE` values in the `index` vector.

4. **A vector of character strings.** This possibility applies only when the elements of a vector have *names*. In that case, a subvector of the names vector may be used in the same way as the positive integers case in 1. That is, the strings in the `index` vector are matched against the names of the elements of `x` and the resulting elements are extracted. Alphanumeric names are often easier to remember than numeric indices of elements. For example,

```
> fruit <- c(oranges = 5, bananas = 10, apples = 1, peaches = 30)
> fruit

oranges bananas  apples peaches
      5      10       1      30

> names(fruit)

[1] "oranges" "bananas" "apples"  "peaches"

> fruit[c("apples", "oranges")]

 apples oranges
      1       5
```

MORE ON MISSING VALUES: The following results may seem surprising:

```
> x <- c(9, 5, 12, NA, 2, NA, NA, 1)
> x[x > 2]
```

```
[1]  9  5 12 NA NA NA
```

```
> x[x == NA]
```

```
[1] NA NA NA NA NA NA NA NA
```

In `x[x > 2]`, the `NA` (missing) elements are extracted from the vector, which is counter-intuitive. And in general, any operator (arithmetic, comparison, or logical) performed on an NA returns an NA. With this in mind, use the `is.na()` function to *explicitly* test whether a value is missing and, in turn, remove any `NA` (missing) values from the vector – the `is.na()` function returns a logical vector that indicates which elements are missing (`NA`). The following are equivalent:

```
> x[x > 2 & !is.na(x)]
```

```
[1]  9  5 12
```

```
> x[x > 2 & is.na(x) == FALSE]
```

```
[1]  9  5 12
```

Missing values (`NA`s) can cause problems in a number of functions. It is always a good idea to look at an individual function's help page to determine how it will handle `NA`s – see the 'Finding help' section.

IMPORTANT: As hinted to in the 'Assignment' section, extracting elements of a data structure, such as a vector, can be done on the left hand side of an assignment expression (e.g., to select parts of a vector to replace) as well as on the right-hand side. For example, we can assign the non-missing values of the `x` vector assigned above to `y` and we can replace the missing values in the `x` vector with zeros:

```
> x
```

```
[1]  9  5 12 NA  2 NA NA  1
```

```
> y <- x[!is.na(x)]
> y
```

```
[1]  9  5 12  2  1
```

```
> x[is.na(x)] <- 0
> x
```

```
[1]  9  5 12  0  2  0  0  1
```

As seen, the *recycling rule*, as discussed in the 'Vectorization of functions' section, was used in `x[!is.na(x)] <- 0`. That is, if a sub-vector extracted for replacement on the left-hand side of an assignment expression (e.g., `x[!is.na(x)]`) is longer than the replacement vector given on the right-hand side (e.g., `0`), the right-hand side is *recycled* as often as necessary.

Also, in a replacement, a positive `index` greater than the length of the vector being extracted extends the vector, assigning `NA`s to any gap. For example,

```
> x
```

```
[1]  9  5 12  0  2  0  0  1
```

```
> length(x)
```

```
[1] 8
```

```
> x[15] <- 1
> x
```

```
 [1]  9  5 12  0  2  0  0  1 NA NA NA NA NA NA  1
```

Lastly, a negative `index` less than `-length(x)` returns an error (`subscript out of bounds`) – e.g., `x[-20]`.

3.4.1.2 Other data structures

Elements (rows and/or columns) of a matrix may be extracted by giving two index vectors in the form `x[i, j]`, where `i` extracts rows and `j` extracts columns. The index vectors `i` and `j` can take any of the four forms shown for vectors. If character vectors are used as indices, they refer to row or column names, as appropriate. And, if either index vector is not specified (i.e., left empty), then the range of that subscript is taken. That is `x[i, ]` extracts the rows specified in `i` across all columns of `x`, and `x[ , j]` extracts the columns specified in `j` across all rows of `x`. Some examples:

```
> x <- matrix(1:12, ncol = 4, byrow = TRUE)
> x

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

> x[1, ]

[1] 1 2 3 4

> x[, 4]

[1]  4  8 12

> x[2, 3]

[1] 7

> x[-(1:2), 3:4]

[1] 11 12

> x[x[, 2] < 8 & x[, 4] < 10, ]

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

Remember, arrays are k-dimensional generalizations of matrices. Therefore, for a k-dimensional array we must give `k` indices from the four forms – `x[i, j, k, ...]`. As with matrices, if any index position is given an empty index vector, then the full range of that subscript is extracted. Some examples:

```
> x <- array(1:24, dim = c(3, 4, 2))
> x

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

```
> x[2, 3, 1]

[1] 8

> x[1:2, c(1, 4), -1]

     [,1] [,2]
[1,]   13   22
[2,]   14   23
```

Recall, each component of a list can be of any type of data structure, including a list. In a list, indexing applies just as it does for vectors using the `[ ]` operators, and there is an additional operator using double square brackets (`[[ ]]`) to provide the *contents* of an individual component of a list. For example, let's create a list and then extract a sub-list, consisting of the first two components of the list.

```
> z <- list("a", TRUE, c = 1:3)
> z

[[1]]
[1] "a"

[[2]]
[1] TRUE

$c
[1] 1 2 3

> z[1:2]

[[1]]
[1] "a"

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

Going one step further, the following code extracts just the contents of the third component of our list `z`, which is a vector.

```
> z[[3]]

[1] 1 2 3
```

Using single brackets would have produced a list of length 1 (i.e., `z[3]`).

If the components of the list have been named, the subset of a list can be specified as the names of the components of the list. For example, the last example could have been written `z[["c"]]`. There is also a short-hand equivalent of the double square brackets using the dollar sign operator (`$`). The `$` operator is used in the form `LISTname$COMPname`, where `LISTname` is the (case-sensitive) name of your list, and `COMPname` is the (case-sensitive) name of the component you want to extract the contents of. So, the third component of our list `z` could be extracted in three separate ways (all of which are equivalent): `z[[3]]`, `z[["c"]]`, and `z$c`.

Because data frames are a generalization of matrices and a special case of lists, elements (rows and/or columns) of a data frame can be extracted like matrices or like lists using the `[ ]`, `[[ ]]`, and/or `$` indexing operators. Some examples:

```
> ourdf <- data.frame(id = 101:110, sex = sample(c("M", "F"), size = 10,
+     replace = TRUE), age = sample(20:50, size = 10, replace = TRUE),
+     tx = sample(c("Drug", "Placebo"), size = 10, replace = TRUE), diabetes = sample(c(TRUE,
+         FALSE)))
> ourdf
```

```
    id sex age      tx diabetes
1  101   F  41 Placebo    FALSE
2  102   F  47 Placebo     TRUE
3  103   F  35 Placebo    FALSE
4  104   F  38 Placebo     TRUE
5  105   F  48    Drug    FALSE
6  106   F  33    Drug     TRUE
7  107   F  26 Placebo    FALSE
8  108   F  43    Drug     TRUE
9  109   M  37 Placebo    FALSE
10 110   F  27    Drug     TRUE
```

```
> ourdf[, 4]
```

```
 [1] Placebo Placebo Placebo Placebo Drug    Drug    Placebo Drug    Placebo Drug
Levels: Drug Placebo
```

```
> ourdf[["tx"]]
```

```
 [1] Placebo Placebo Placebo Placebo Drug    Drug    Placebo Drug    Placebo Drug
Levels: Drug Placebo
```

```
> ourdf$tx
```

```
 [1] Placebo Placebo Placebo Placebo Drug    Drug    Placebo Drug    Placebo Drug
Levels: Drug Placebo
```

```
> ourdf[ourdf$sex == "M", c("id", "tx")]
```

```
   id      tx
9 109 Placebo
```

As can be seen, the character column names are often used when specifying the index vectors.

REMEMBER: If the columns of the matrix or data frame you are using in the conditional selection of rows contains missing values, you will need to explicitly remove the `NA`s. For example, if the `sex` column of `ourdf` contained `NA`s, in order to extract the rows for which `sex == "M"`, we would have to specify `ourdf[ ourdf$sex == "M" & !is.na(ourdf$sex), ]`.

In future lectures, we will explore alternatives to using the `[ ]` and `$` operators to subset data frames, such as the `subset()` and `complete.cases()` functions.

3.5 Flow control statements

Until now, we have discussed components of the R language that involve the evaluation of single expressions. In actuality, *conditional* evaluation and *repetitive* evaluation are also possible in the R language, which incorporate *several* expressions. These two types of evaluation are governed by what are known as '*flow control statements*.'

3.5.1 Conditional evaluation

The `if` statement allows us to evaluate expressions based on a condition, and takes one of two forms. The first is the 'if-then' form:

```
if(condition) {
```

```
   expression(s) if condition is TRUE
}
```

The `condition` is a logical expression that, when evaluated, returns a *single* `TRUE` or `FALSE` value – an error is returned if the `condition` does not evaluate to a logical value. If the `condition` evaluates to `TRUE`, then any `expression(s)` between the braces (`{` and `}`) is/are evaluated. The second form is an 'if-then-else' form:

```
if(condition) {
   expression(s) if condition is TRUE
} else {
   expression(s) if condition is FALSE
}
```

In this form, the `condition` is evaluated, and any `expression(s)` between the *first set* of braces is/are evaluated if the `condition` evaluates to `TRUE`. If the `condition` evaluates to `FALSE`, then the `expression(s)` between the *second set* of braces is/are evaluated.

The 'if-then-else' form of the `if` statement can also be *nested*.

```
if(condition1) {
   expression(s) if condition1 is TRUE
} else if(condition2) {
   expression(s) if condition 1 is FALSE but condition2 is TRUE
} else if(condition3) {
   expression(s) if both condition1 and 2 are FALSE but condition3 is TRUE
} else {
   expression(s) if condition1, 2, and 3 are FALSE
}
```

In any form, the `if` statement returns the value of the expression evaluated, or `NULL` if no expression was, which may happen if there is no `else`.

When the `expression(s)` is/are not specified in a block involving braces (`{` and `}`), then the `else`, if present, *must* appear on the same line as the `if(condition)`. In general though, it is a good habit to *always* use braces to block the expressions with the appropriate part of the statement.

As mentioned, the `condition` of an `if` statement is expected to return a *single* `TRUE` or `FALSE` value when evaluated. If the `condition` returns a logical vector of more than one element, then a warning is given. For example,

```
> x <- c(2, 5, 7, NA, 10, NA, -1)
> if (x < 0) {
+     print("< 0")
+ } else {
+     print("> 0")
+ }

[1] "> 0"

Warning message:
the condition has length > 1 and only the first element will be used in:  if (x < 0) {
```

In this example, the `condition` `x < 0` actually returns a logical vector the same length as `x`:

```
> x < 0

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

When the `condition` evaluates to a logical vector of more than one element, as the warning suggests, only the first element of the evaluated `condition` (either `TRUE` or `FALSE`) is used – in this example, `FALSE`.

An alternative to the `if` statement is the `ifelse()` function, which is the vectorized version of the `if` statement. The `ifelse()` function has the form `ifelse(condition, expression1, expression2)` and returns a vector of the length of its longest argument, with elements `expression1[i]` if `condition` is `TRUE`, or `expression2[i]` otherwise. Here's an example:

```
> x <- c(6:-4)
> ifelse(x <= 0, "Negative", "Positive")

 [1] "Positive" "Positive" "Positive" "Positive" "Positive" "Positive" "Negative"
 [8] "Negative" "Negative" "Negative" "Negative"
```

The `ifelse()` function can also be nested:

```
> ifelse(x < 0, "< 0", ifelse(x >= 0 & x <= 3, "0 - 3", "> 3"))

 [1] "> 3"   "> 3"   "> 3"   "0 - 3" "0 - 3" "0 - 3" "0 - 3" "< 0"   "< 0"   "< 0"
[11] "< 0"
```

The `ifelse()` function is very useful when you want to create new columns in a data frame that are derived from existing ones. An alternative to the `if` statement and `ifelse()`) function is the `switch()` function.

See the `if` statement and `ifelse()` function help files for more details. Because it's a nonstandard topic name, the `if` statement must be quoted to access its help file – `help("if")`.

3.5.2 Repetitive evaluation

Repetitive evaluation, which involves the repeated evaluation of an expression or a block of expressions, is often called '*looping*.' There are three statements that will perform looping: (1) the `for` statement; (2) the `while` statement; and (3) the `repeat` statement. In general, the `while` and `repeat` statements are rarely used in R.

The standard `for` loop has the basic structure of

```
for( increment in sequence ) {
   expression(s)
}
```

The `increment` is a variable name – often the name of an indexer, such as `i`. The `sequence` can be any vector or list, and the `expression(s)` is/are evaluated for each `increment` (i.e., value) of `sequence`. When `sequence` is a vector, `increment` loops over each element of the vector (i.e., `sequence[increment]`). When `sequence` is a list, `increment` refers to each successive component in the list (i.e., `sequence[[increment]]`). A simple example is a countdown program:

```
> for (i in 5:1) {
+     print(i)
+ }

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

In this example, `increment` is the variable `i`, and `sequence` is the numeric vector of the set of numbers 10 through 1.

A `for` loop returns the value of the last expression evaluate (or `NULL` if none was), and sets `increment` to the last used element (component) of `sequence` (or to `NULL` if it was of length zero).

```
> i

[1] 1
```

As an alternative to using `for` loops, R provides the group of `apply()` functions (`lapply()`, `mapply()`, `sapply()`, `tapply()`, etc.), which perform *implicit* looping. And ***DON'T FORGET***, many functions and operators in R are *vectorized*, so you may not need to use a loop – see the 'Vectorization of functions' section.

Obviously, the `for` loop is great to use when you know ahead of time what sequence of values you want to loop over. However, sometimes you don't know this. In this case, you may want to repeat something as long as a condition is met (e.g., as long as a number is positive, or a number is larger than some tolerance). For this, use a `while` loop:

```
while( condition ) {
   expression(s)
}
```

Like the `if` statement, the `condition` is a logical expression that, when evaluated, returns *single* `TRUE` or `FALSE` value. If the `condition` evaluates to `TRUE`, then any `expression(s)` between the braces (`{` and `}`) is/are evaluated. This process continues until the `condition` evaluates to `FALSE`. Like the `for` loop, the `while` loop returns the value of the last evaluation of the `expression(s)`. For example,

```
> i <- 1
> while (i <= 5) {
+     cat("Iteration", i, "\n")
+     i <- i + 1
+ }

Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

If the `expression(s)` is/are never evaluated, then the `while()` function returns `NULL`. And like the `if` statement, a warning is returned and only the first logical element of the evaluated `condition` is used if the `condition` returns a logical vector of more than one element.

Lastly, a `repeat` loop causes repeated evaluation of expressions until a break is specifically requested. This means that you need to be careful when using `repeat` because of the danger of an infinite loop. The syntax of the `repeat` loop is

```
repeat {
   expression(s)
}
```

Unlike `if`, `for`, and `while`, when using `repeat`, the `expression(s)` must be a block of code denoted with braces (`{` and `}`). This is because you need to both perform some computation and test whether or not to break from the loop, which usually requires at least two expressions.

The additional flow control statements `next` and `break` can be used to skip the next value in a loop or to break out of a loop, respectively. More specifically, `break` breaks out of a `for`, `while` or `repeat` loop, and control is transferred to the first statement outside the inner-most loop. `next` halts the processing of the current iteration and advances the looping index. Both `break` and `next` apply only to the innermost of nested loops.

The following example is a contrast and comparison of the `repeat` and `for` loops:

```
> i <- 0
> repeat {
```

```
+     if (i > 20)
+         break
+     if (i > 5 && i < 15) {
+         i <- i + 1
+         next
+     }
+     print(i)
+     i <- i + 1
+ }
```

```
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 15
[1] 16
[1] 17
[1] 18
[1] 19
[1] 20
```

```
> for (i in 0:20) {
+     if (i > 5 && i < 15)
+         next
+     print(i)
+ }
```

```
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 15
[1] 16
[1] 17
[1] 18
[1] 19
[1] 20
```

See the appropriate help files for more details. All three looping statements must be quoted to access their help files – e.g., `help("for")`.

Chapter 4

Other Essentials

4.1 Object management

As we mentioned in the 'Assignment' section, it is important to know whether a variable name is already being used before assigning it to a new variable. The `exists()` function can be used to determine whether an object with a proposed name already exists. The `exists()` function will search for the name as a function or as the name of another assigned value. In the following examples, we defined `x` in previous sections, and `length` is a function, but `lngth` has not been assigned yet.

```
> exists("x")

[1] TRUE

> exists("length")

[1] TRUE

> exists("lngth")

[1] FALSE
```

During an R session, every object you assign a name is stored in what is known as your *workspace*. The `ls()` or `objects()` function can be used to display the names of all the variables currently assigned. If you have run all of the proceeding examples in this document, your workspace should include all of the following objects:

```
> ls()

 [1] "anotherFUN" "fruit"      "i"          "lastFUN"    "ourdf"      "ourFUN"
 [7] "race"       "stress"     "test"       "x"          "y"          "z"
```

As you can imagine, your workspace can quickly become cluttered, and it can become difficult to identify specific desired objects from among this clutter.

The `rm()` function can be used to remove any unnecessary and/or unwanted objects by specifying the names of these objects. The name(s) may or may not be quoted, and multiple names can be specified if separated by commas. For example, to remove the object `x`, we would type `rm(x)` at the command line. To remove both the `x` and `y` objects, we would type `rm(x, y)`.

Multiple names can also be given in a vector form as a quoted string to the `list=` argument. Therefore, the previous example could be typed as: `rm(list=c("x", "y"))`. We can use this `list=` argument in conjunction with the `ls()` function to easily remove all of the objects in your workspace by typing `rm(list =`

`ls()`). In the Windows version of R, the entire current workspace can also be cleared by selecting Remove all objects from the Misc drop-menu. We can also take this example one step further by incorporating the `setdiff()` function, which returns the difference between two sets of elements, to remove all objects except specified ones from your current workspace: `rm(list = setdiff(ls(), c(object1, object2)))`, where `object1` and `object2` are those objects you wish to keep.

There are also several ways to *save* desired objects. Recall, when you quit R, you are asked `Save your workspace image?` Accepting this offer will save all the objects in your current workspace to a *directory specific* hidden file name `.RData`. If you saved your workspace before quitting and start R from the same directory at a later time, R loads this saved workspace and all of the objects are restored to the current workspace. The `save.image()` function can also be explicitly used to save the current workspace. By default, the objects are saved to the `.RData` hidden file, but a different file name can be specified using the `file=` argument. Unfortunately both of these methods are very inefficient and use a lot of memory to save the workspace.

An alternative is to use the `save()` function, which has an option to compress the file the objects are saved to. This is extremely useful when dealing with large objects that take significant execution time to create and/or normally take up scarce memory. Specify `compress = TRUE` to store the specified file very compactly – `save(list = c("object1", "object2"), file = "objects.RData", compress = TRUE)`, where `object1` and `object2` are those objects you wish to save, and `objects.RData` is the file you wish to compactly save them too.

Use `load("objects.RData")` to reload the saved objects at a later time.

4.2 Finding help

Using R requires knowing a lot of different functions and other language specifics. All of this is more than most of us can keep in our head at any given time. Thankfully, R has an excellent built-in help system that allows you to access individual help files. Most of the help files pertain to specific functions, but there are some regarding language essentials.

There are several functions we can use to access and search the help files on a specific topic in R including (1) the `help()` function; (2) the `apropos()` function; and (3) the `help.search()` function.

The `help()` function will access the help file of a specific topic, but you need to know the exact topic name on which the help documentation is sought. In general, the `help()` function is most helpful when you want to learn more about a function you already know the name of, like the `mean()` function. The `help()` function can be called in several ways, which are all equivalent: `help(mean)`; `?mean`; `?"mean"`; and `help("mean")`. To use the `help()` function to access the help files on a topic specified by *special* or *non-conventional* characters, for example $*$ or `[[`, the argument must be enclosed in double or single quotes, making it a character string (i.e., `help("*")`). By default, the `help()` function only searches in the packages which are loaded in memory – specify `try.all.packages = TRUE` to search in all packages (whether installed or loaded; e.g., `help("bs", try.all.packages=TRUE)`).

The `apropos()` function returns a character vector containing all the objects (functions or assigned variables) whose name contains the character string given as the argument. Only the packages loaded in memory are searched. An example would be to search all objects for the character string `"mean"`:

```
> apropos("mean")

 [1] "smean.cl.boot"   "smean.cl.normal" "smean.sd"        "smean.sdl"
 [5] "wtd.mean"        "kmeans"          "weighted.mean"   "mean"
 [9] "mean.data.frame" "mean.Date"       "mean.default"    "mean.difftime"
[13] "mean.POSIXct"    "mean.POSIXlt"
```

An alternative to the `apropos()` function is the `find()` function.

The `help.search()` function searches the help system for documentation matching a given character string in the name, alias, title, concept or keyword entries (or any combination thereof) of a help files, using either fuzzy matching or regular expression matching – e.g., `help.search("regression")`. The `help.search()` function is a good function to use when you can't remember the specific name of the function or topic, or if you are trying to find a function to perform a specific task. The results are printed in two columns: the function name and corresponding package name in parentheses is shown in the left column, while the full function title is show in the right.

In the Windows version on R, the help pages can be accessed via the Help drop-down menu. In addition, help is available in HTML format on most R installations by calling the `help.start()` function, which will launch a Web browser that allows the help pages to be browsed with hyperlinks. The Search Engine and Keywords link in the page loaded by `help.start()` is particularly useful as it contains a high-level concept list which searches through available functions. It can be a great way to get your bearings quickly and to understand the breadth of what R has to offer.

In addition, as you will see in the next section, the help pages often include a series of examples and the `example()` function is useful for running these examples.

All of these mentioned help functions have many options, so it useful to access the help files of each of the help functions for details and examples.

ANATOMY OF A HELP FILE: It is helpful to understand the general layout of a help file. The first two lines of the help file contain general information, such as the name of the package where the documented function(s) or operator(s) is/are located, and the title of the topic. In addition, each help file usually contains the following sections:

- **Description**, which gives a brief description of the topic.
- **Usage**, which displays a function's arguments and their possible default values. For non-function topics, the Usage section gives the typical use.
- **Arguments**, which details each of a function's arguments.
- **Details**, which provides a more detailed description of the topic.
- **Value**, which, if applicable, describes the type of object returned by the function or topic.
- **See also**, which lists other help files similar to the present one.
- **Examples**, which contains some example code using the function.

For beginners, it is good to look at the `Examples` section. Generally, it is useful to read the `Arguments` section carefully. And it should be noted that, other sections may be encountered, such as `Note`, `References`, and/or `Author(s)`.

4.3 Good programming practices

1. Create a directory (folder) where you will keep all your data, code, and output related to a particular project. This can be thought of as your *'working directory'* whenever you use R for that particular project.
2. Start your R session from within the relevant working directory. This allows you to keep assigned objects separate for each project and not overload your workspace, and therefore memory. Specifically, under R for Windows, use Change dir... from the File to Browse for and select the relevant working directory after starting R. And under R for Linux/Unix, use the `cd` (i.e. change directory) command at the shell prompt to move to the relevant working directory before launching R.

3. Create and use an R *code file* to completely and cumulatively record your analysis. Doing so ensures reproducible results. It also allows you to 'cannibalize' your code for other projects.

4. Make your code files well readable and as much self-explaining as possible. This includes indenting and using spaces appropriately; explicitly specifying function arguments by name; wrapping long lines by explicitly breaking long expressions; and adding copious comments to your code.[1]

5. When naming and assigning new objects, make sure that the proposed name is not already assigned to an existing object.

6. Keep in mind that there are usually multiple ways of performing the same task in R.

7. Read, Read, Read – read the documentation, including help files and their examples; read other's code; and read the body of defined functions.

8. Like learning any new programming language, R has a steep learning curve, in part due to a number of fine points and common pitfalls which may surprise the user at first. However, taking the time to really learn R will be well worth it in the end - DON'T QUIT!

[1] Any text following a `#` on the command line, to the end of the line, is taken as a 'comment' and ignored by R. Comments can be put almost anywhere, except inside quoted strings, and within the argument list of a function definition. For example, `> 175*(8/5) # convert 175 miles to kms - correct comment`.